CakePHP Book

Version 5.x

Cake Software Foundation

29 juin 2025

Table des matières

1 CakePHP en un Coup d'Oeil **1**
Conventions plutôt que Configuration . . . 1
La Couche Model (Modèle) . . . 1
La Couche View (Vue) . . . 2
La Couche Controller (Contrôleur) . . . 2
Cycle de Requête CakePHP . . . 3
Que le Début . . . 3
Lectures Complémentaires . . . 3

2 Guide de Démarrage Rapide **13**
Tutoriel d'un système de gestion de contenu . . . 13
Tutoriel CMS - Création de la base de données . . . 15
Tutoriel CMS - Création du Controller Articles . . . 19

3 Guides de Migration **29**
Guide de mise à jour pour la version 4.0 . . . 29
Guide de migration vers la version 4.0 . . . 31
Guide de migration vers la version 4.1 . . . 39
Guide de migration vers la version 4.2 . . . 44
Guide de migration vers la version 4.3 . . . 48
Guide de migration vers la version 4.4 . . . 53

4 Tutoriels et exemples **59**
Tutoriel d'un système de gestion de contenu . . . 59
Tutoriel CMS - Création de la base de données . . . 61
Tutoriel CMS - Création du Controller Articles . . . 65
Tutoriel CMS - Tags et Users . . . 75
Tutoriel CMS - Authentification . . . 83
Tutoriel CMS - Autorisation . . . 89
Tutoriel de Bookmarker . . . 93
Tutoriel de Bookmarker Part 2 . . . 100
Tutoriel d'un Blog . . . 108
Tutoriel d'un Blog - Partie 2 . . . 111
Tutoriel d'un Blog - Partie 3 . . . 123

Tutoriel d'un Blog - Authentification . . . 130

5 Contribuer **139**
Documentation . . . 139
Tickets . . . 147
Code . . . 147
Normes de codes . . . 150
Guide de Compatibilité Rétroactive . . . 160

6 Installation **163**
Installer CakePHP . . . 164
Permissions . . . 165
Serveur de Développement . . . 165
Production . . . 166
A vous de jouer ! . . . 167
Réécriture d'URL . . . 167

7 Configuration **175**
Configurer votre Application . . . 175
Variables d'Environnement . . . 176
Chemins de Classe Supplémentaires . . . 178
Configuration de Inflection . . . 180
Classe Configure . . . 180
Lire et Ecrire les Fichiers de Configuration . . . 182
Désactiver les tables génériques . . . 184

8 Routing **185**
Tour Rapide . . . 185
Connecter les Routes . . . 187
Créer des Routes RESTful . . . 202
Arguments Passés . . . 205
Générer des URLs . . . 206
Générer des URL de ressources . . . 209
Routing de Redirection . . . 210
Routage des Entités . . . 210
Classes Route Personnalisées . . . 211
Créer des Paramètres d'URL Persistants . . . 212

9 Les Objets Request & Response **215**
ServerRequest . . . 215
Response . . . 225
Définir les En-têtes de Requête d'Origine Croisée (Cross Origin Request Headers = CORS) . . . 232
Erreurs Communes avec les Responses Immutables . . . 233
CookieCollections . . . 233

10 Controllers (Contrôleurs) **237**
Le Controller App . . . 237
Déroulement d'une Requête . . . 238
Les Actions du Controller . . . 238
Interactions avec les Vues . . . 239
Négociation de Contenu . . . 242
Rediriger vers d'Autres Pages . . . 242
Chargement des Modèles Supplémentaires . . . 243
Paginer un Modèle . . . 243
Configurer les Components à Charger . . . 244

Cycle de Vie des Callbacks de la Requête . . . 244
Middleware de Controller . . . 245
Plus sur les Controllers . . . 245

11 Views (Vues) 289
La View App . . . 289
Templates de Vues . . . 290
Utiliser les Blocks de Vues . . . 293
Layouts . . . 295
Elements . . . 297
Events de View . . . 300
Créer vos propres Classes de View . . . 300
En savoir plus sur les vues . . . 301

12 Accès Base de Données & ORM 395
Exemple Rapide . . . 395
Pour en savoir plus . . . 397

13 La mise en cache 571
Configuration de la classe Cache . . . 572
Ecrire dans un Cache . . . 574
Lire depuis un Cache . . . 575
Suppression d'un Cache . . . 576
Effacer les Données du Cache . . . 576
Utiliser le Cache pour Stocker les Compteurs . . . 577
Utiliser le Cache pour Stocker les Résultats de Requêtes Courantes . . . 577
Utilisation des Groupes . . . 578
Activer ou Désactiver Globalement le Cache . . . 579
Création d'un moteur de stockage pour le Cache . . . 579

14 Console Bake 581

15 Commandes sur la Console 583
La Console CakePHP . . . 583
Applications sur Console . . . 584
Renommer des Commandes . . . 585
Commandes . . . 585
Commandes Fournies par CakePHP . . . 607
Routage dans l'Environnement de la Console . . . 621

16 Debugger 623
Debug Basique . . . 623
Utiliser la Classe Debugger . . . 624
Affichage des Valeurs . . . 624
Logging With Stack Traces . . . 624
Generating Stack Traces . . . 625
Getting an Excerpt From a File . . . 625
Utiliser les Logs pour Debugger . . . 626
Kit de Debug . . . 626

17 Déploiement 627
Déplacer les Fichiers . . . 627
Ajuster la Configuration . . . 627
Vérifier Votre Sécurité . . . 628
Définir le Document Root . . . 628

Améliorer les Performances de votre Application . . . 628
Déployer une Mise à Jour . . . 629

18 Mailer **631**
Utilisation basique . . . 631
Configuration . . . 632
Envoyer des Pièces Jointes . . . 635
Envoyer des Messages Rapidement . . . 636
Envoyer des Emails en Ligne de Commande . . . 636
Créer des Emails Réutilisables . . . 637
Configurer les Transports . . . 638
Envoyer des Emails sans utiliser Mailer . . . 641
Tester les Mailers . . . 641

19 Gestion des Erreurs & Exceptions **645**
Configuration des Erreurs et des Exceptions . . . 645
Modifier la gestion des exceptions . . . 646
Templates d'erreur personnalisés . . . 647
ErrorController personnalisé . . . 647
ExceptionRenderer personnalisé . . . 648
Créer vos Propres Gestionnaires d'Erreurs . . . 649
Logging Personnalisé des Erreurs . . . 651
Créer vos propres Exceptions d'Application . . . 652
Exceptions Intégrées de CakePHP . . . 653

20 Événements système **657**
Exemple d'Utilisation d'Événement . . . 657
Accéder aux Gestionnaires d'Événements . . . 658
Events du Cœur . . . 659
Enregistrer les Listeners . . . 660
Dispatcher les Events . . . 663
Lecture Supplémentaire . . . 666

21 Internationalisation & Localisation **667**
Internationaliser Votre Application . . . 667
Utiliser les Fonctions de Traduction . . . 669
Créer Vos Propres Traducteurs . . . 673
Localiser les Dates et les Nombres . . . 676
Sélection Automatique de Locale Basée sur les Données de Requêtes . . . 678
Translate Content/Entities . . . 679

22 Journalisation (logging) **681**
Configuration des flux d'un log (journal) . . . 681
Journalisation des Erreurs et des Exception . . . 682
Ecrire dans les logs . . . 683
Scopes de Journalisation . . . 684
Utilisation de l'Adaptateur FileLog . . . 685
Logging vers Syslog . . . 685
l'API de Log . . . 687
Logging Trait . . . 688
Utiliser Monolog . . . 688

23 Formulaires Sans Models **691**
Créer un Formulaire . . . 691
Traiter les Données de Requêtes . . . 692

Définir des Valeurs pour le Formulaire . . . 693
Récupérer les Erreurs d'un Formulaire . . . 694
Invalider un Champ de Formulaire depuis un Controller . . . 694
Créer le HTML avec FormHelper . . . 694

24 Plugins **697**
Installer un Plugin Avec Composer . . . 697
Installer un Plugin Manuellement . . . 698
Charger un Plugin . . . 698
Configuration du Plugin . . . 699
Utiliser un Plugin . . . 700
Créer Vos Propres Plugins . . . 701
Plugin Classes . . . 702
Routes de Plugins . . . 703
Controllers du Plugin . . . 704
Models du Plugin . . . 705
Vues du Plugin . . . 706
Ressources de Plugin . . . 707
Components, Helpers et Behaviors . . . 708
Commands . . . 708
Tester votre Plugin . . . 709
Publier votre Plugin . . . 709
Fichier de Mappage de Plugin . . . 709
Gérer Vos Plugins avec Mixer . . . 709

25 REST **711**
Mise en place Simple . . . 711
Encodage des données de réponse . . . 713
Parsing des corps de requête . . . 713

26 Sécurité **715**
Security . . . 715
Protection CSRF . . . 717
Middleware Content Security Policy . . . 720
Middleware des Headers de Sécurité . . . 721
Middleware HTTPS Enforcer . . . 721
Ajouter Strict-Transport-Security . . . 722

27 Sessions **723**
Configuration de Session . . . 723
Gestionnaires de Session intégrés & Configuration . . . 724
Configurer les Directives ini . . . 726
Créer un Gestionnaire de Session Personnalisé . . . 727
Accéder à l'Objet Session . . . 728
Lire & Ecrire les Données de Session . . . 729
Détruire la Session . . . 729
Faire une Rotation des Identificateurs de Session . . . 730
Messages Flash . . . 730

28 Testing **731**
Installer PHPUnit . . . 731
Tester la Configuration de la Base de Données . . . 732
Vérifier la Configuration Test . . . 732
Conventions des Cas de Test (TestCase) . . . 733
Créer Votre Premier Cas de Test . . . 733

Lancer les Tests . . . 735
Callbacks du Cycle de Vie des Cas de Test . . . 737
Fixtures . . . 737
Tester les Classes De Tables . . . 744
Tests d'Intégration des Controllers . . . 746
Tests d'Intégration en Console . . . 759
Mocker les Injections de Dépendances . . . 759
Mocker les Réponses du Client HTTP . . . 759
Tester les Views . . . 759
Tester les Components . . . 760
Tester les Helpers . . . 761
Tester les Events . . . 763
Testing Email . . . 765
Créer des Suites de Test (Test Suites) . . . 765
Créer des Tests pour les Plugins . . . 765
Générer des Tests avec Bake . . . 766

29 Validation **767**
Créer les Validators . . . 767
Valider les Données . . . 774
Valider les Entities . . . 775
Règles de Validation du Cœur . . . 776

30 Classe App **777**
Trouver les Classes . . . 777
Trouver les Chemins vers les Namespaces . . . 778
Localiser les Plugins . . . 778
Localiser les Themes . . . 778
Charger les Fichiers de Vendor . . . 778

31 Collections **781**
Exemple Rapide . . . 781
Liste des Méthodes . . . 782
Faire une Itération . . . 782
Filtrer . . . 786
Agrégation . . . 788
Trier . . . 791
Utiliser des Données en Arbre . . . 792
Autres Méthodes . . . 794

32 Folder & File **803**
Utilisation Basique . . . 803
API de Folder . . . 804
L'API de File . . . 808

33 Hash **811**
Syntaxe de chemin Hash . . . 811

34 Client Http **827**
Faire des Requêtes . . . 827
Créer des Requêtes Multipart avec des Fichiers . . . 828
Envoyer des Corps de Requête . . . 829
Options de la Méthode Request . . . 829
Authentification . . . 830
Créer des Clients Délimités (Scoped Clients) . . . 831

Configurer et Gérer les Cookies . . . 832
Objets Response . . . 833
Changer les Adaptateurs de Transport . . . 835
Tests . . . 835

35 Inflector **837**
Résumé des Méthodes d'Inflector et de leurs Sorties . . . 837
Créer des Formes Pluriel et Singulier . . . 838
Créer des Formes en CamelCase et en Underscore . . . 839
Créer des Formes Lisibles par l'Homme . . . 839
Créer des Formes pour les Tables et les Noms de Classe . . . 839
Créer des Noms de Variable . . . 839
Configuration d'Inflexion . . . 840

36 Number **841**
Formatage des Devises . . . 842
Paramétrage de la Devise par Défaut . . . 842
Formatage Des Nombres A Virgules Flottantes . . . 843
Formatage Des Pourcentages . . . 843
Interagir Avec Des Valeurs Lisibles Par L'Homme . . . 843
Formatage Des Nombres . . . 844
Formatage Des Différences . . . 845
Configurer le Formatage . . . 846

37 Objets Registry **847**
Charger les Objets . . . 847
Attraper les Callbacks . . . 848
Désactiver les Callbacks . . . 848

38 Text **849**
Convertion des Chaînes de Caractères en ASCII . . . 850
Créer des chaînes saines pour les URL . . . 850
Générer des UUIDs . . . 851
Parseur de chaînes simples . . . 851
Formater une chaîne . . . 851
Fixer la largeur d'un texte . . . 852
Subrillance de Sous-Chaîne . . . 852
Retirer les Liens . . . 853
Tronquer le Texte . . . 853
Tronquer une chaîne par la fin . . . 854
Générer un Extrait . . . 855
Convertir un tableau sous la forme d'une phrase . . . 855

39 Date & Time **857**
Créer des Instances FrozenTime . . . 858
Manipulation . . . 858
Formatage . . . 859
Conversion . . . 863
Comparer Avec le Present . . . 863
Comparer Avec Des Intervalles . . . 864
FrozenDate . . . 864
Dates et Heures Modifiables . . . 865
Accepter des Données de Requête Localisées . . . 866
Timezones Supportés . . . 866

40 Xml **867**
Importer les données vers la classe Xml . . . 867
Transformer une Chaîne de Caractères XML en Tableau . . . 868
Transformer un tableau en une chaîne de caractères XML . . . 868

41 Globales & Fonctions **871**
Fonctions Globales . . . 871
Définitions des constantes du noyau . . . 873
Définition de Constantes de Temps . . . 874

42 Chronos **875**

43 Debug Kit **877**

44 Migrations **879**

45 ElasticSearch **881**

46 Annexes **883**
4.x Guide de Migration . . . 883
Mimer la Compatibilité Descendante . . . 886
Mimer la Compatibilité Ascendante . . . 886
Informations générales . . . 886

PHP Namespace Index **889**

Index **891**

CHAPITRE 1

CakePHP en un Coup d'Oeil

CakePHP est conçu pour faciliter et simplifier les tâches classiques du développement web. En fournissant une boite à outil tout-en-un pour vous aider à démarrer, les différentes parties de CakePHP fonctionnent aussi bien ensemble que séparément.

Le but de cette présentation est d'introduire les concepts généraux de CakePHP, et de vous donner un aperçu rapide de la façon dont ces concepts sont mis en œuvre dans CakePHP. Si vous êtes impatient de démarrer un projet, vous pouvez *commencer avec le tutoriel*, ou vous plonger dans la documentation.

Conventions plutôt que Configuration

CakePHP fournit une structure organisationnelle de base qui comprend les noms de classes, les noms de fichiers, les noms de table de base de données, et d'autres conventions. Même si les conventions peuvent paraître longues à apprendre, en suivant les conventions offertes par CakePHP, vous pouvez éviter les configurations inutiles et construire une structure d'application uniforme ce qui facilite le travail quand vous travaillez sur de multiples projets. Le *chapitre sur les conventions* couvre les différentes conventions utilisées par CakePHP.

La Couche Model (Modèle)

La couche Model représente la partie de l'application qui exécute la logique applicative. Elle est responsable de récupérer les données et de les convertir selon des concepts significatifs pour votre application. Cela inclut le traitement, la validation, l'association et beaucoup d'autres tâches concernant la manipulation des données.

Dans le cas d'un réseau social, la couche Model s'occuperait des tâches telles que sauvegarder les données utilisateur, sauvegarder les associations d'amis, enregistrer et récupérer les photos des utilisateurs, trouver des suggestions de nouveaux amis, etc … Tandis que les objets Models seraient « Friend », « User », « Comment », « Photo ». Si nous voulions charger des données depuis notre table `users`, nous pourrions faire :

```
use Cake\ORM\Locator\LocatorAwareTrait;
```

(suite sur la page suivante)

(suite de la page précédente)

```
$users = $this->getTableLocator()->get('Users');
$resultset = $users->find()->all();
foreach ($resultset as $row) {
    echo $row->username;
}
```

Vous remarquerez peut-être que nous n'avons pas eu à écrire de code avant que nous puissions commencer à travailler avec nos données. En utilisant les conventions, CakePHP utilisera des classes standards pour les classes table et entity qui n'ont pas encore été définies.

Si nous voulions créer un nouvel utilisateur et l'enregistrer (avec validation), nous ferions ceci :

```
use Cake\ORM\Locator\LocatorAwareTrait;

$users = $this->getTableLocator()->get('Users');
$user = $users->newEntity(['email' => 'mark@example.com']);
$users->save($user);
```

La Couche View (Vue)

La View retourne une présentation des données modélisées. Etant séparée des objets Model, elle est responsable de l'utilisation des informations dont elle dispose pour produire n'importe quelle interface de présentation nécessaire à votre application.

Par exemple, la view pourrait utiliser les données du model pour afficher un template de vue HTML contenant ces données, ou alors un résultat au format XML pour que d'autres l'utilisent :

```
// Dans un fichier de template de vue, nous afficherions un 'element' pour chaque user.
<?php foreach ($users as $user): ?>
    <li class="user">
        <?= $this->element('user_info', ['user' => $user]) ?>
    </li>
<?php endforeach; ?>
```

La couche View fournit un certain nombre d'extensions tels que les *Templates*, les *Elements* et les *Cells* pour vous permettre de réutiliser votre logique de présentation.

La couche View n'est pas seulement limitée au HTML ou à la représentation en texte de données. Elle peut aussi être utilisée pour offrir une grande variété de formats tels que JSON, XML et grâce à une architecture modulable tout autre format dont vous auriez besoin, comme CSV par exemple.

La Couche Controller (Contrôleur)

La couche Controller gère les requêtes des utilisateurs. Elle est responsable de retourner une réponse avec l'aide mutuelle des couches Model et View.

Les Controllers peuvent être imaginés comme des managers qui ont pour mission que toutes les ressources nécessaires pour accomplir une tâche soient déléguées aux bonnes personnes. Il attend des requêtes des clients, vérifie leur validité selon l'authentification et les règles d'autorisation, délègue la récupération et le traitement des données à la couche Model, puis sélectionne les types de présentation acceptés par le client pour finalement déléguer le processus de rendu à la couche View. Un exemple de controller d'enregistrement d'utilisateur serait :

```
public function add()
{
    $user = $this->Users->newEmptyEntity();
    if ($this->request->is('post')) {
        $user = $this->Users->patchEntity($user, $this->request->getData());
        if ($this->Users->save($user, ['validate' => 'registration'])) {
            $this->Flash->success(__('Vous êtes maintenant enregistré.'));
        } else {
            $this->Flash->error(__('Il y a eu un problème.'));
        }
    }
    $this->set('user', $user);
}
```

Notez que nous n'avons jamais explicitement rendu de view. Les conventions de CakePHP prendront soin de sélectionner la bonne view et de la rendre avec les données préparées avec `set()`.

Cycle de Requête CakePHP

Maintenant que vous êtes familier avec les différentes couches de CakePHP, voyons comment fonctionne le cycle d'une requête CakePHP :

Le cycle d'une requête CakePHP typique débute avec une requête utilisateur qui demande une page ou une ressource de votre application. À haut niveau chaque requête passe par les étapes suivantes :

1. Les règles de réécriture de votre serveur web dirigent la requête vers **webroot/index.php**.
2. Votre Application est chargée et liée à un `HttpServer`.
3. Le middleware de votre application est initialisé.
4. Une requête et une réponse sont dispatchées à travers le Middleware PSR-7 utilisé par votre application. Typiquement, il inclut l'interception d'erreurs et le routing.
5. Si aucune réponse n'est retournée à partir du middleware et que la requête contient des informations de routing, un controller et une action sont sélectionnés.
6. L'action du controller est appelée et le controller interagit avec les Models et Components nécessaires.
7. Le controller délègue la création de la response à la View pour générer le résultat obtenu à partir des données du model.
8. Le View utilise les Helpers et les Cells pour générer l'en-tête et le corps de la réponse.
9. La réponse est de nouveau envoyée à travers le /controllers/middleware.
10. `HttpServer` émet la réponse au serveur web.

Que le Début

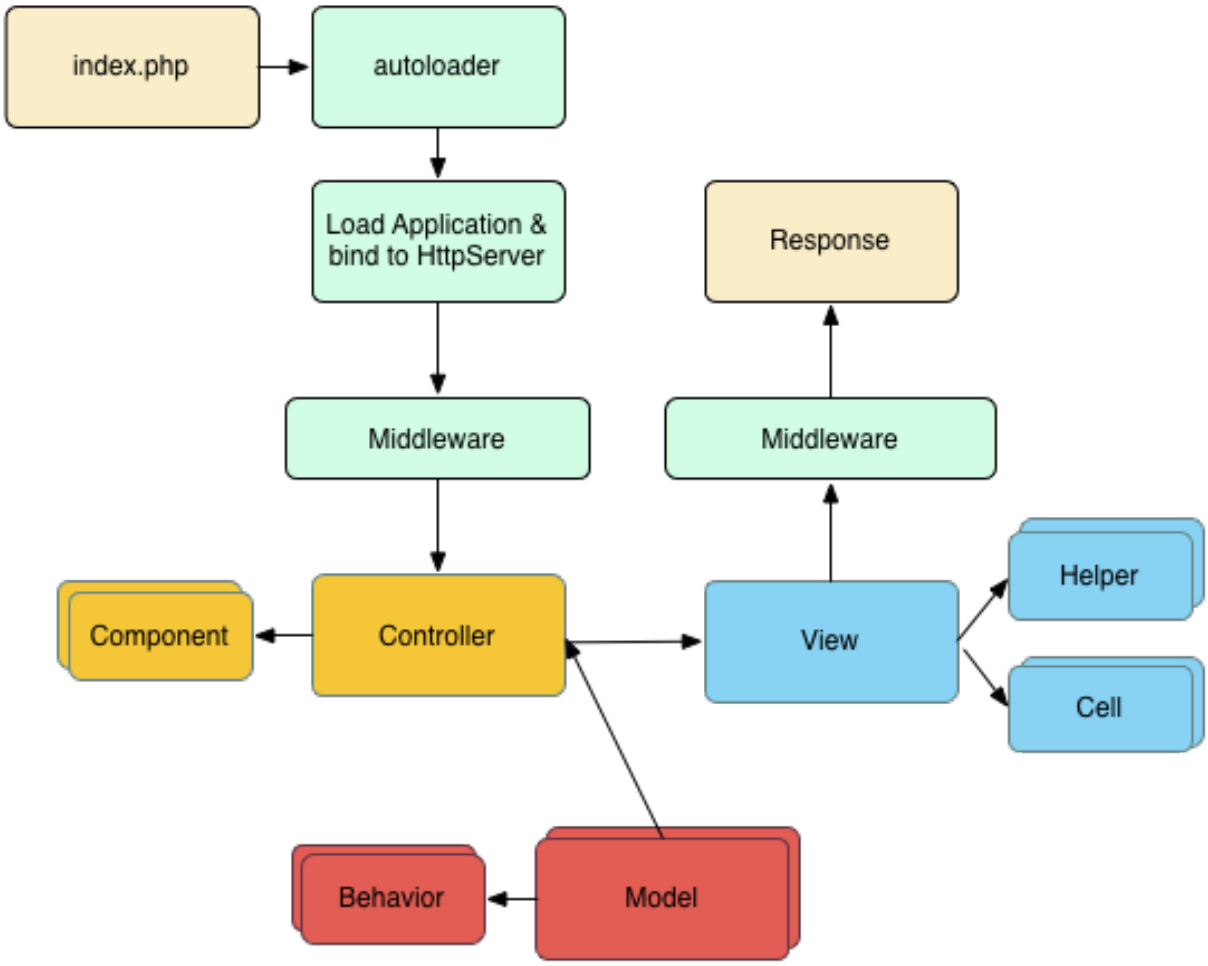
Espérons que ce bref aperçu ait éveillé votre intérêt. Quelques autres grandes fonctionnalités de CakePHP sont :

— Un framework de *cache* qui s'intègre à Memcached, Redis et d'autres moteurs de cache.
— Un outil de génération de code puissant pour partir sur les chapeaux de roue.
— Un *framework de tests intégré* pour vous assurer que votre code fonctionne correctement.

Les prochaines étapes évidentes sont de *télécharger CakePHP*, lire le *tutoriel et construire un truc génial*.

Lectures Complémentaires

Où obtenir de l'aide


index.php
autoloader
Load Application &
bind to HttpServer
Response
Middleware
Middleware
Helper
Component
Controller
View
Cell
Behavior
Model

Le Site Officiel de CakePHP

https://cakephp.org

Le site officiel de CakePHP est toujours un endroit épatant à visiter. Il propose des liens vers des outils fréquemment utilisés par le développeur, des didacticiels vidéo, des possibilités de faire un don et des téléchargements.

Le Cookbook

https://book.cakephp.org

Ce manuel devrait probablement être le premier endroit où vous rendre pour obtenir des réponses. Comme pour beaucoup d'autres projets open source, nous accueillons de nouvelles personnes régulièrement. Faîtes tout votre possible pour répondre à vos questions vous-même dans un premier temps. Les réponses peuvent venir lentement, mais elles resteront longtemps et vous aurez ainsi allégé notre charge de travail en support utilisateur. Le manuel et l'API ont tous deux une version en ligne.

La Boulangerie

https://bakery.cakephp.org

La Boulangerie (Bakery) est une chambre de compensation pour tout ce qui concerne CakePHP. Vous y trouverez des tutoriels, des études de cas et des exemples de code. Lorsque vous serez familiarisés avec CakePHP, connectez-vous pour partager vos connaissances avec la communauté et obtenez en un instant la gloire et la fortune.

L'API

https://api.cakephp.org/

Allez droit au but et atteignez le graal des développeurs, l'API CakePHP (Application Programming Interface) est la documentation la plus complète sur tous les détails essentiels au fonctionnement interne du framework. C'est une référence directe au code, donc apportez votre chapeau à hélice.

Les Cas de Test

Si vous avez toujours le sentiment que l'information fournie par l'API est insuffisante, regardez le code des cas de test fournis avec CakePHP. Ils peuvent servir d'exemples pratiques pour l'utilisation d'une fonction et de données membres d'une classe :

```
tests/TestCase/
```

Le Canal IRC

Canaux IRC sur irc.freenode.net :

- #cakephp – Discussion générale.
- #cakephp-docs – Documentation.
- #cakephp-bakery – Bakery.
- #cakephp-fr – Canal francophone.

Si vous êtes paumé, poussez un hurlement sur le canal IRC de CakePHP. Une personne de l'équipe de développement [4] s'y trouve habituellement, en particulier durant les heures du jour pour les utilisateurs d'Amérique du Nord et du Sud. Nous serions ravis de vous écouter, que vous ayez besoin d'un peu d'aide, que vous vouliez trouver des utilisateurs dans votre région ou que vous souhaitiez donner votre nouvelle marque de voiture sportive.

4. https://cakephp.org/team

Forum Officiel

Forum Officiel de CakePHP [5]

Notre forum officiel où vous pouvez demander de l'aide, suggérer des idées et discuter de CakePHP. C'est l'endroit idéal pour trouver rapidement des réponses et aider les autres. Rejoignez la famille de CakePHP en vous y inscrivant.

Stackoverflow

https://stackoverflow.com/ [6]

Taggez vos questions avec `cakephp` et la version spécifique que vous utilisez pour activer les utilisateurs existants de stackoverflow pour trouver vos questions.

Où Trouver de l'Aide dans Votre Langue

Portugais brésilien

— Communauté de CakePHP brésilienne [7]

Danoise

— Canal CakePHP danois sur Slack [8]

Française

— Communauté de CakePHP Francophone [9]

Allemande

— Canal CakePHP allemand sur Slack [10]
— Groupe Facebook CakePHP Allemand [11]

Iranienne

— Communauté CakePHP Iranienne [12]

Hollandaise

— Canal CakePHP hollandais sur Slack [13]

Japonaise

— Canal Slack CakePHP japonais [14]
— Groupe Facebook CakePHP japonais [15]

5. https://discourse.cakephp.org
6. https://stackoverflow.com/questions/tagged/cakephp/
7. https://cakephp-br.org
8. https://cakesf.slack.com/messages/denmark/
9. https://cakephp-fr.org
10. https://cakesf.slack.com/messages/german/
11. https://www.facebook.com/groups/146324018754907/
12. https://cakephp.ir
13. https://cakesf.slack.com/messages/netherlands/
14. https://cakesf.slack.com/messages/japanese/
15. https://www.facebook.com/groups/304490963004377/

Portugaise

— Groupe Google CakePHP protugais [16]

Espagnol

— Canal CakePHP espagnol sur Slack [17]
— Canal IRC de CakePHP espagnol
— Groupe Google de CakePHP espagnol [18]

Conventions de CakePHP

Nous sommes de grands fans des conventions plutôt que de la configuration. Bien que cela réclame un peu de temps pour apprendre les conventions de CakePHP, à terme vous gagnerez du temps. En suivant les conventions, vous aurez des fonctionnalités automatiques et vous vous libérerez du cauchemar de la maintenance du suivi des fichiers de configuration. Les conventions créent un environnement de développement uniforme, permettant à d'autres développeurs de s'investir d'avantage et d'aider.

Les Conventions des Controllers

Les noms des classes de controller sont au pluriel, en CamelCase et se terminent par `Controller`. `UsersController` et `MenuLinksController` sont des exemples respectant cette convention.

Les méthodes publiques des controllers sont souvent exposées comme des "actions" accessibles via un navigateur web. Par exemple `/users/view-me` correspond à la méthode `viewMe()` de `UsersController` sans rien modifier (si l'on utilise l'inflexion dashed par défaut dans le routage. Les méthodes privées ou protégées ne sont pas accessibles avec le routing.

Considérations concernant les URLs et les Noms des Controllers

Comme vous venez de voir, un controller dont le nom est constitué d'un seul mot renvoie vers un chemin URL en minuscules. Par exemple, `UsersController` (qui serait défini dans le fichier nommé **UsersController.php**) est accessible à l'adresse http://exemple.com/users.

Alors que vous pouvez router des controllers dont le nom est formé de plusieurs mots de la façon que vous souhaitez, la convention est que vos URLs soient en minuscules avec des tirets en utilisant la classe `DashedRoute`, donc `/menu-links/view-all` est la bonne forme pour accéder à l'action `MenuLinksController::viewAll()`.

Quand vous créez des liens en utilisant `this->Html->link()`, vous pouvez utiliser les conventions suivantes pour le tableau d'url :

```
$this->Html->link('link-title', [
    'prefix' => 'MyPrefix' // CamelCased
    'plugin' => 'MyPlugin', // CamelCased
    'controller' => 'ControllerName', // CamelCased
    'action' => 'actionName' // camelBacked
]
```

Pour plus d'informations sur les URLs de CakePHP et la gestion des paramètres, allez voir *Connecter les Routes*.

16. https://groups.google.com/group/cakephp-pt
17. https://cakesf.slack.com/messages/spanish/
18. https://groups.google.com/group/cakephp-esp

Conventions des Fichiers et des Noms de Classe

En général, les noms de fichiers correspondent aux noms des classes et suivent les standards PSR-4 pour l'autoloading (chargement automatique). Voici quelques exemples de noms de classes et de fichiers :

- La classe controller `LatestArticlesController` devra se trouver dans un fichier nommé **LatestArticlesController.php**.
- La classe Component (Composant) `MyHandyComponent` devra se trouver dans un fichier nommé **MyHandyComponent.php**.
- La classe Table `OptionValuesTable` devra se trouver dans un fichier nommé **OptionValuesTable.php**.
- La classe Entity `OptionValue` devra se trouver dans un fichier nommé **OptionValue.php**.
- La classe Behavior (Comportement) `EspeciallyFunkableBehavior` devra se trouver dans un fichier nommé **EspeciallyFunkableBehavior.php**.
- La classe View (Vue) `SuperSimpleView` devra se trouver dans un fichier nommé **SuperSimpleView.php**.
- La classe Helper (Assistant) `BestEverHelper` devra se trouver dans un fichier nommé **BestEverHelper.php**.

Chaque fichier sera situé dans le répertoire/namespace approprié dans le dossier de votre application.

Conventions pour les Models et les Bases de Données

Les noms de tables correspondant aux models CakePHP sont au pluriel et utilisent le caractère souligné (underscore). Les tables correspondantes aux models mentionnés ci-dessus seront donc respectivement : `users`, `menu_links` et `user_favorite_pages`. Si le nom de table contient plusieurs mots, seul le dernier doit être au pluriel, par exemple `menu_links`.

Les noms des champs avec deux mots ou plus doivent être avec des underscores comme ici : `first_name`.

Les clés étrangères des relations hasMany, belongsTo ou hasOne sont reconnues par défaut grâce au nom (singulier) de la table associée, suivi de `_id`. Donc, si Users hasMany Articles, la table `articles` se référera à la table `users` via une clé étrangère `user_id`. Pour une table avec un nom de plusieurs mots comme `menu_links`, la clé étrangère sera `menu_link_id`.

Les tables de jointure sont utilisées dans les relations BelongsToMany entre models. Elles doivent être nommées d'après le nom des tables qu'elles unissent. Les noms doivent être au pluriel et dans l'ordre alphabétique : `articles_tags` plutôt que `tags_articles` ou *àrticle_tags`. Si vous ne respectez pas ces conventions, la commande bake ne fonctionnera pas.* Si la table de jointure contient d'autres colonnes que les clés étrangères qui servent à l'association, vous devriez créer une une entité/table réelle pour cette table.

En plus de l'utilisation des clés auto-incrémentées en tant que clés primaires, vous pouvez aussi utiliser des colonnes UUID. CakePHP va créer un UUID unique de 36 caractères (`CakeUtilityText::uuid()`) à chaque fois que vous sauvegarderez un nouvel enregistrement en utlisant la méthode `Table::save()`.

Model Conventions

Les noms de classe de modèle (model) sont au pluriel, en CamelCase et finissent par `Table`. `UsersTable`, `MenuLinksTable` et `UserFavoritePagesTable` sont des exemples de nom de classes de table correspondant respectivement aux tables `users`, `menu_links` and `user_favorite_pages`.

Les noms de classe d'entités (entity) sont au singulier, en CamelCase et ne possèdent pas de suffixe. `User`, `MenuLink` et `UserFavoritePage` sont des exemples de noms d'entités correspondant respectivement aux tables `users`, `menu_links` and `user_favorite_pages`.

Conventions des Views

Les fichiers de template de view sont nommés d'après les fonctions du controller qu'elles affichent, sous une forme avec underscores. La fonction `viewAll()` de la classe `ArticlesController` cherchera un gabarit de view dans **templates/Articles/view_all.php**.

Le schéma classique est **templates/Controller/nom_de_fonction_avec_underscore.php**.

> **Note**
>
> Par défaut, CakePHP utilise des inflexions anglaises. Si vous avez une base de données tables/colonnes qui utilisent une autre langue, vous devrez ajouter une règle d'inflexion (du singulier au pluriel et vice-versa). Vous pouvez utiliser `Cake\Utility\Inflector` pour définir vos règles d'inflexion personnalisées. Voir la documentation sur : *Inflector* pour plus d'informations.

Conventions des Plugins

Il est utile de préfixer un plugin CakePHP avec « cakephp- » dans le nom du paquet. Cela rend le nom sémantiquement lié au framework dont il dépend.

N'utilisez **pas** l'espace de noms CakePHP (cakephp) comme nom de fournisseur car c'est réservé aux plugins appartenant à CakePHP. La convention est d'utiliser des lettres minuscules et des tirets comme séparateur :

```
// Mauvais
cakephp/foo-bar

// Bon
votre-nom/cakephp-foo-bar
```

Voir awesome list recommendations [19] pour plus de détails.

En résumé

En utilisant les conventions CakePHP dans le nommage des différentes parties de votre application, vous gagnerez des fonctionnalités sans les tracas et les affres de la configuration. Voici un exemple récapitulant les conventions abordées :

- Nom de la table de la base de données : « articles », « menu_links »
- Classe Table : `ArticlesTable` se trouvant dans **src/Model/Table/ArticlesTable.php**
- Classe Entity : `Article` se trouvant dans **src/Model/Entity/Article.php**
- Classe Controller : `ArticlesController` se trouvant dans **src/Controller/ArticlesController.php**
- Template de View se trouvant dans **templates/Articles/index.php**

En utilisant ces conventions, CakePHP sait qu'une requête de type http://exemple.com/articles/ sera liée à un appel à la fonction `index()` du Controller `ArticlesController`, dans lequel le model `Articles` est automatiquement disponible. Aucune de ces relations n'a été configurée par rien d'autre que la création des classes et des fichiers dont vous aviez besoin de toute façon.

19. https://github.com/FriendsOfCake/awesome-cakephp/blob/master/CONTRIBUTING.md#tips-for-creating-cakephp-plugins

Exempl	articles	menu_links	
Table en base de données	articles	menu_links	Les noms de tables correspondant au modèles de CakePHP sont au pluriel et utilisent le caractère souligné (underscore)
Fichier	ArticlesController.php	MenuLinksController.php	
Table	ArticlesTable.php	MenuLinksTable.php	Les noms de classes sont au pluriel, CamelCased et se terminent par Table
Entity	Article.php	MenuLink.php	Les nom des classes d'entités (Entity) sont au singulier, CamelCased : Article et MenuLink
Class	ArticlesController	MenuLinksController	
Controller	ArticlesController	MenuLinksController	Pluriel, CamelCased, se termine par Controller
Behavior	ArticlesBehavior.php	MenuLinksBehavior.php	
View	ArticlesView.php	MenuLinksView.php	Les fichiers de template de view sont nommés d'après les fonctions du controller qu'elles affichent, sous une forme avec underscores
Helper	ArticlesHelper.php	MenuLinksHelper.php	
Component	ArticlesComponent.php	MenuLinksComponent.php	
Plugin	Mauvais : cakephp/articles Bon : you/cakephp-articles	cakephp/menu-links vous/cakephp-menu-links	Il est utile de préfixer un plugin CakePHP avec « cakephp- » dans le nom du paquet. N'utilisez pas l'espace de noms CakePHP (cakephp) comme nom de fournisseur car c'est réservé aux plugins appartenant à CakePHP. La convention est d'utiliser des lettres minuscules et des tirets comme séparateur
Chaque fichier sera situé dans le dossier/espace de noms approprié dans le dossier de votre application.			

Résumé des conventions de Base de Données

Clés étrangères hasMany belongsTo/ hasOne BelongsToMany	Les relations sont reconnues par défaut comme le nom (au singulier) de la table associée suivi par `_id`. Users hasMany Articles, la table `articles` fera référence à la table `users` via la clé étrangère `user_id`.
Plusieurs mots	Pour `menu_links` dont le nom contient plusieurs mots, la clé étrangère serait `menu_link_id`.
Auto Increment	En plus d'utiliser un entier auto-incrémenté comme clés primaires, vous pouvez également utiliser des colonnes UUID. CakePHP créera automatiquement les valeurs UUID en utilisant (*`Cake\Utility\Text::uuid()`*) à chaque fois que vous sauvegarderez un nouvel enregistrement en utlisant la méthode `Table::save()`.
Tables jointes	Doivent être nommées d'après les tables du modèle qu'elles joindront dans l'ordre alphabétique (`articles_tags` plutôt que `tags_articles`) sinon la commande bake ne fonctionnera pas. En cas de besoin de colonne supplémentaire dans la table intermédiaire, vous devez créer une entité/table séparée pour cette table.

Maintenant que vous avez été initié aux fondamentaux de CakePHP, vous devriez essayer de dérouler *le tutoriel du Blog CakePHP* pour voir comment les choses s'articulent.

Structure du dossier de CakePHP

Après avoir téléchargé et extrait l'application CakePHP, voici les fichiers et répertoires que vous devriez voir :

- Le dossier *bin* contient les exécutables de la console Cake.
- Le dossier *config* contient les (quelques) fichiers de *Configuration* que CakePHP utilise. Les détails sur la connexion à la base de données, le bootstrapping, les fichiers de configuration du cœur et consorts doivent être stockés ici.
- Le dossier *logs* contient normalement vos fichiers de log selon la configuration par défaut des logs.
- Le dossier *plugins* est l'endroit où sont stockés les *Plugins* que votre application utilise.
- Le dossier *src* sera celui vous placerez les fichiers source de votre application.
- Le dossier *templates* contient les fichiers de présentation : éléments, pages d'erreur, mises en page (layout) et les fichiers de vues.
- Le dossier *resources* contient un sous-dossier pour différents types de fichiers de ressources. Le sous-dossier *locales* stocke les fichiers de langue pour l'internationalisation.
- Le dossier *tests* est l'endroit où vous mettez les cas de test pour votre application.
- Le dossier *tmp* est l'endroit où CakePHP stocke les données temporaires. Les données qu'il stocke dépendent de la façon dont vous avez configuré CakePHP mais ce dossier est généralement utilisé pour les stocker les traductions, les descriptions de model et parfois les informations de session.
- Le dossier *vendor* est l'endroit où CakePHP et d'autres dépendances de l'application vont être installés par Composer [20]. Modifier ces fichiers est déconseillé car composer écrasera vos changements lors du prochain update que vous ferez.
- Le répertoire *webroot* est la racine publique de votre application. Il contient tous les fichiers que vous souhaitez voir accessibles publiquement.

  Assurez-vous que les dossiers *tmp* et *logs* existent et qu'ils sont en écriture, autrement la performance de votre application sera sévèrement impactée. En mode debug, CakePHP vous avertira que ces dossiers ne peuvent pas être écrits.

Le Dossier Src

Le répertoire *src* de CakePHP est l'endroit où vous réaliserez la majorité du développement de votre application. Regardons d'un peu plus près les dossiers à l'intérieur de *src*.

Command

Contient les commandes de la console de votre application. Voir *Objets Command* pour en savoir plus.

20. https://getcomposer.org

Console
Contient le script d'installation exécuté par Composer.

Controller
Contient les *Controllers (Contrôleurs)* et les components de votre application.

Middleware
Contient les /controllers/middleware de votre application.

Model
Contient les tables, entity et behaviors de votre application.

Shell
Contient les commandes de la console et les tasks de la console pour votre application. Pour plus d'informations, regardez la section *Shells*.

View
Les classes de présentation sont placés ici : views, cells, helpers.

> **Note**
>
> Le dossier `Shell` n'est pas présent par défaut. Vous pouvez l'ajouter lorsque vous en avez besoin..

CHAPITRE 2

Guide de Démarrage Rapide

Le meilleur moyen de tester et d'apprendre CakePHP est de s'assoir et de construire une application simple de gestion de Contenu (CMS).

Tutoriel d'un système de gestion de contenu

Ce tutoriel vous accompagnera dans la création d'une application de type CMS (Content Management System). Pour commencer, nous installerons CakePHP, créerons notre base de données et construirons un système simple de gestion d'articles.

Voici les pré-requis :

1. Un serveur de base de données. Nous utiliserons MySQL dans ce tutoriel. Vous avez besoin de connaître assez de SQL pour créer une base de données et exécuter quelques requêtes SQL que nous fournirons dans ce tutoriel. CakePHP se chargera de construire les requêtes nécessaires pour votre application. Puisque nous allons utiliser MySQL, assurez-vous que `pdo_mysql` est bien activé dans PHP.
2. Les connaissances de base en PHP.

Avant de commencer, assurez-vous que votre version de PHP est à jour :

```
php -v
```

Vous devez avoir au minimum PHP 8.1 installé (en CLI). Votre version serveur de PHP doit au moins être aussi 8.1 et, dans l'idéal, devrait également être la même que pour votre version en ligne de commande (CLI).

Récupérer CakePHP

La manière la plus simple d'installer CakePHP est d'utiliser Composer. Composer est une manière simple d'installer CakePHP via votre terminal. Premièrement, vous devez télécharger et installer Composer si vous ne l'avez pas déjà fait. Si vous avez cURL installé, exécutez la commande suivante :

```
curl -s https://getcomposer.org/installer | php
```

Ou vous pouvez télécharger `composer.phar` depuis le site de Composer [21].

Ensuite, tapez la commande suivante dans votre terminal pour installer le squelette d'application CakePHP dans le dossier **cms** du dossier courant :

```
php composer.phar create-project --prefer-dist cakephp/app:5.* cms
```

Si vous avez téléchargé et utilisé l'Installer de Composer pour Windows [22], tapez la commande suivante dans votre terminal depuis le dossier d'installation (par exemple C :\wamp\www\dev) :

```
composer self-update && composer create-project --prefer-dist cakephp/app:4.* cms
```

Utiliser Composer a l'avantage d'exécuter automatiquement certaines tâches importantes d'installation, comme définir les bonnes permissions sur les dossiers et créer votre fichier **config/app.php**.

Il existe d'autres moyens d'installer CakePHP. Si vous ne pouvez pas (ou ne voulez pas) utiliser Composer, rendez-vous dans la section *Installation*.

Quelque soit la manière de télécharger et installer CakePHP, une fois que la mise en place est terminée, votre dossier d'installation devrait ressembler à ceci :

```
/cms
  /bin
  /config
  /logs
  /plugins
  /resources
  /src
  /templates
  /tests
  /tmp
  /vendor
  /webroot
  .editorconfig
  .gitignore
  .htaccess
  .travis.yml
  composer.json
  index.php
  phpunit.xml.dist
  README.md
```

C'est le bon moment pour en apprendre d'avantage sur le fonctionnement de la structure des dossiers de CakePHP : rendez-vous dans la section *Structure du dossier de CakePHP* pour en savoir plus.

Si vous vous perdez pendant ce tutoriel, vous pouvez voir le résultat final on GitHub [23].

Vérifier l'installation

Il est possible de vérifier que l'installation est terminée en vous rendant sur la page d'accueil. Avant de faire ça, vous allez devoir lancer le serveur de développement :

21. https://getcomposer.org/download/
22. https://getcomposer.org/Composer-Setup.exe
23. https://github.com/cakephp/cms-tutorial

```
cd /path/to/our/app

bin/cake server
```

> **Note**
>
> Pour Windows, la commande doit être `bin\cake server` (notez le backslash).

Cela démarrera le serveur embarqué de PHP sur le port 8765. Ouvrez **http ://localhost :8765** dans votre navigateur pour voir la page d'accueil. Tous les éléments de la liste devront être validés sauf le point indiquant si CakePHP arrive à se connecter à la base de données. Si d'autres points ne sont pas validés, vous avez peut-être besoin d'installer des extensions PHP supplémentaires ou définir les bonnes permissions sur certains dossiers.

Ensuite, nous allons créer notre *base de données et créer notre premier model*.

Tutoriel CMS - Création de la base de données

Maintenant que CakePHP est installé, il est temps d'installer la base de données pour notre application CMS. Si vous ne l'avez pas encore fait, créez une base de données vide qui servira pour ce tutoriel, avec le nom de votre choix (par exemple `cake_cms`). Si vous utilisez MySQL/MariaDB, vous pouvez exécuter le SQL suivant pour créer le tables nécessaires :

```
USE cake_cms;

CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    email VARCHAR(255) NOT NULL,
    password VARCHAR(255) NOT NULL,
    created DATETIME,
    modified DATETIME
);

CREATE TABLE articles (
    id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT NOT NULL,
    title VARCHAR(255) NOT NULL,
    slug VARCHAR(191) NOT NULL,
    body TEXT,
    published BOOLEAN DEFAULT FALSE,
    created DATETIME,
    modified DATETIME,
    UNIQUE KEY (slug),
    FOREIGN KEY user_key (user_id) REFERENCES users(id)
) CHARSET=utf8mb4;

CREATE TABLE tags (
    id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(191),
    created DATETIME,
    modified DATETIME,
    UNIQUE KEY (title)
```

(suite sur la page suivante)

(suite de la page précédente)

```
) CHARSET=utf8mb4;

CREATE TABLE articles_tags (
    article_id INT NOT NULL,
    tag_id INT NOT NULL,
    PRIMARY KEY (article_id, tag_id),
    FOREIGN KEY tag_key(tag_id) REFERENCES tags(id),
    FOREIGN KEY article_key(article_id) REFERENCES articles(id)
);

INSERT INTO users (email, password, created, modified)
VALUES
('cakephp@example.com', 'secret', NOW(), NOW());

INSERT INTO articles (user_id, title, slug, body, published, created, modified)
VALUES
(1, 'First Post', 'first-post', 'This is the first post.', 1, NOW(), NOW());
```

Si vous utilisez PostgreSQL, connectez-vous à la base de données `cake_cms` et exécutez le code SQL suivant à la place :

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    email VARCHAR(255) NOT NULL,
    password VARCHAR(255) NOT NULL,
    created TIMESTAMP,
    modified TIMESTAMP
);

CREATE TABLE articles (
    id SERIAL PRIMARY KEY,
    user_id INT NOT NULL,
    title VARCHAR(255) NOT NULL,
    slug VARCHAR(191) NOT NULL,
    body TEXT,
    published BOOLEAN DEFAULT FALSE,
    created TIMESTAMP,
    modified TIMESTAMP,
    UNIQUE (slug),
    FOREIGN KEY (user_id) REFERENCES users(id)
);

CREATE TABLE tags (
    id SERIAL PRIMARY KEY,
    title VARCHAR(191),
    created TIMESTAMP,
    modified TIMESTAMP,
    UNIQUE (title)
);

CREATE TABLE articles_tags (
    article_id INT NOT NULL,
```

(suite sur la page suivante)

(suite de la page précédente)

```
    tag_id INT NOT NULL,
    PRIMARY KEY (article_id, tag_id),
    FOREIGN KEY (tag_id) REFERENCES tags(id),
    FOREIGN KEY (article_id) REFERENCES articles(id)
);

INSERT INTO users (email, password, created, modified)
VALUES
('cakephp@example.com', 'secret', NOW(), NOW());

INSERT INTO articles (user_id, title, slug, body, published, created, modified)
VALUES
(1, 'First Post', 'first-post', 'This is the first post.', TRUE, NOW(), NOW());
```

Vous avez peut-être remarqué que la table `articles_tags` utilise une clé primaire composée. CakePHP supporte les clés primaires composées presque partout, vous permettant d'avoir des shémas plus simples qui ne nécessitent pas de colonnes `id` supplémentaires.

Les noms de tables et de colonnes utilisés ne sont pas arbitraires. En utilisant les *conventions de nommages* de CakePHP, nous allons bénéficier des avantages de CakePHP de manière plus efficace et allons éviter d'avoir trop de configuration à effectuer. Bien que CakePHP soit assez flexible pour supporter presque n'importe quel schéma de base de données, adhérer aux conventions va vous faire gagner du temps.

Configuration de la base de données

Ensuite, disons à CakePHP où est notre base de données et comment nous y connecter. Remplacez les valeurs dans le tableau `Datasources.default` de votre fichier **config/app.php** avec celle de votre installation de base de données. Un exemple de configuration complétée ressemblera à ceci :

```
<?php
return [
    // D'autres configurations au dessus
    'Datasources' => [
        'default' => [
            'className' => Connection::class,
            // Remplacez Mysql par Postgres si vous utilisez PostgreSQL
            'driver' => Mysql::class,
            'persistent' => false,
            'host' => 'localhost',
            'username' => 'cakephp',
            'password' => 'AngelF00dC4k3~',
            'database' => 'cake_cms',
            // Commentez la ligne ci-dessous si vous utilisez PostgreSQL
            'encoding' => 'utf8mb4',
            'timezone' => 'UTC',
            'cacheMetadata' => true,
        ],
    ],
    // D'autres configurations en dessous
];
```

Une fois que vous avez sauvegardé votre fichier **config/app.php**, vous devriez voir que CakePHP est capable de se connecter à la base de données sur la page d'accueil de votre projet.

> **Note**
>
> Si vous avez **config/app_local.php** dans votre dossier d'application, vous devez plutôt configurer votre connexion à la base de données dans ce fichier.

Création du premier Model

Les models font partie du coeur des applications CakePHP. Ils nous permettent de lire et modifier les données, de construire des relations entre nos données, de valider les données et d'appliquer les règles spécifiques à notre application. Les models sont les fondations nécessaires pour construire nos actions de controllers et nos templates.

Les models de CakePHP sont composés d'objets `Table` et `Entity`. Les objets `Table` nous permettent d'accéder aux collections d'entities stockées dans une table spécifique. Ils sont stockés dans le dossier **src/Model/Table**. Le fichier que nous allons créer sera sauvegardé dans **src/Model/Table/ArticlesTable.php**. Le fichier devra contenir ceci :

```
<?php
// src/Model/Table/ArticlesTable.php
namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{
    public function initialize(array $config): void
    {
        $this->addBehavior('Timestamp');
    }
}
```

Nous y avons attaché le behavior *Timestamp* qui remplira automatiquement les colonnes `created` et `modified` de notre table. En nommant notre objet Table `ArticlesTable`, CakePHP va utiliser les conventions de nommages pour savoir que notre model va utiliser la table `articles`. Toujours en utilisant les conventions, il saura que la colonne `id` est notre clé primaire.

> **Note**
>
> CakePHP créera dynamiquement un objet model s'il n'en trouve pas un qui correspond dans le dossier **src/Model/Table**. Cela veut dire que si vous faites une erreur lors du nommage du fichier (par exemple articles-table.php ou ArticleTable.php), CakePHP ne reconnaitra pas votre configuration et utilisera ce model généré à la place.

Nous allons également créer une classe Entity pour nos Articles. Les Entities représentent un enregistrement spécifique en base et donnent accès aux données d'une ligne de notre base. Notre Entity sera sauvegardée dans **src/Model/Entity/Article.php**. Le fichier devra ressembler à ceci :

```
<?php
// src/Model/Entity/Article.php
namespace App\Model\Entity;

use Cake\ORM\Entity;

class Article extends Entity
```

(suite sur la page suivante)

(suite de la page précédente)

```
{
    protected array $_accessible = [
        '*' => true,
        'id' => false,
        'slug' => false,
    ];
}
```

Notre entity est assez simple pour l'instant et nous y avons seulement défini la propriété `_accessible` qui permet de contrôler quelles propriétés peuvent être modifiées via *Assignement de Masse*.

Pour l'instant, nous ne pouvons pas faire grande chose avec notre model. Pour intéragir avec notre model, nous allons ensuite créer nos premiers *Controller et Template*.

Tutoriel CMS - Création du Controller Articles

Maintenant que notre model est créé, nous avons besoin d'un controller pour nos articles. Dans CakePHP, les controllers se chargent de gérer les requêtes HTTP et exécutent la logique métier des méthodes des models pour préparer une réponse. Nous placerons le code de ce controller dans un nouveau fichier **ArticlesController.php**, dans le dossier **src/Controller**. La base du controller ressemblera à ceci :

```
<?php
// src/Controller/ArticlesController.php

namespace App\Controller;

class ArticlesController extends AppController
{
}
```

Ajoutons maintenant une action à notre controller. Les actions sont les méthodes des controllers qui sont connectées aux routes. Par exemple, quand un utilisateur appelle la page **www.example.com/articles/index** (ce qui est la même chose qu'appeler **www.example.com/articles**), CakePHP appelera la méthode `index` de votre controller `ArticlesController`. Cette méthode devra à son tour faire appel à la couche Model et préparer une réponse en faisant le rendu d'un Template via la couche de View. Le code de notre action index sera le suivant :

```
<?php
// src/Controller/ArticlesController.php

namespace App\Controller;

class ArticlesController extends AppController
{
    public function index()
    {
        $this->loadComponent('Paginator');
        $articles = $this->Paginator->paginate($this->Articles->find());
        $this->set(compact('articles'));
    }
}
```

Maintenant que nous avons une méthode `index()` dans notre `ArticlesController`, les utilisateurs peuvent y accéder via **www.example.com/articles/index**. De la même manière, si nous définissions une méthode `foobar()`, les utilisa-

teurs pourraient y accéder via **www.example.com/articles/foobar**. Vous pourriez être tenté de nommer vos controllers et vos actions afin d'obtenir des URL spécifiques. Cependant, ceci est déconseillé. Vous devriez plutôt suivre les *Conventions de CakePHP* et créer des noms d'actions lisibles ayant un sens pour votre application. Vous pouvez ensuite utiliser le *Routing* pour obtenir les URLs que vous souhaitez et les connecter aux actions que vous avez créées.

Notre action est très simple. Elle récupère un jeu d'articles paginés dans la base de données en utilisant l'objet Model Articles qui est chargé automatiquement via les conventions de nommage. Elle utilise ensuite la méthode `set()` pour passer les articles récupérés au Template (que nous créerons par la suite). CakePHP va automatiquement rendre le Template une fois que notre action de Controller sera entièrement exécutée.

Création du Template de liste des Articles

Maintenant que notre controller récupère les données depuis le model et qu'il prépare le contexte pour la view, créons le template pour notre action index.

Les templates de view de CakePHP sont des morceaux de PHP qui sont insérés dans le layout de votre application. Bien que nous créerons du HTML ici, les Views peuvent générer du JSON, du CSV ou même des fichiers binaires comme des PDFs.

Un layout est le code de présentation qui englobe la view d'une action. Les fichiers de layout contiennent les éléments communs au site comme les headers, les footers et les éléments de navigation. Votre application peut très bien avoir plusieurs layouts et vous pouvez passer de l'un à l'autre. Mais pour le moment, utilisons seulement le layout par défaut.

Les fichiers de template de CakePHP sont stockés dans **templates** et dans un dossier au nom du controller auquel ils sont attachés. Nous devons donc créer un dossier nommé "Articles" dans notre cas. Ajoutez le code suivant dans ce fichier :

```
<!-- Fichier : templates/Articles/index.php -->

<h1>Articles</h1>
<table>
    <tr>
        <th>Titre</th>
        <th>Créé le</th>
    </tr>

    <!-- C'est ici que nous bouclons sur notre objet Query $articles pour afficher les␣
↪informations de chaque article -->

    <?php foreach ($articles as $article): ?>
    <tr>
        <td>
            <?= $this->Html->link($article->title, ['action' => 'view', $article->slug])␣
↪?>
        </td>
        <td>
            <?= $article->created->format(DATE_RFC850) ?>
        </td>
    </tr>
    <?php endforeach; ?>
</table>
```

Dans la précédente section, nous avons assigné la variable "articles" à la view en utilisant la méthode `set()`. Les variables passées à la view sont disponibles dans les templates de view comme des variables locales, comme nous l'avons fait ci-dessus.

Vous avez peut-être remarqué que nous utilisons un objet appelé `$this->Html`. C'est une instance du *HtmlHelper*.

CakePHP inclut plusieurs helpers de view qui peuvent créer des liens, des formulaires et des éléments de paginations. Vous pouvez en apprendre plus à propos des *Helpers (Assistants)* dans le chapitre de la documentation qui leur est consacré, mais le plus important ici est la méthode `link()`, qui générera un lien HTML avec le texte fourni (le premier paramètre) et l'URL (le second paramètre).

Quand vous spécifiez des URLs dans CakePHP, il est recommandé d'utiliser des tableaux ou des *routes nommées*. Ces syntaxes vous permettent de bénéficier du reverse routing fourni par CakePHP.

A partir de maintenant, si vous accédez à **http ://localhost :8765/articles/index**, vous devriez voir votre view qui liste les articles avec leur titre et leur lien.

Création de l'action View

Si vous cliquez sur le lien d'un article dans la page qui liste nos articles, vous tombez sur une page d'erreur vous indiquant que l'action n'a pas été implémentée. Vous pouvez corriger cette erreur en créant l'action manquante correspondante :

```
// Ajouter au fichier existant src/Controller/ArticlesController.php

public function view($slug = null)
{
    $article = $this->Articles->findBySlug($slug)->firstOrFail();
    $this->set(compact('article'));
}
```

Bien que cette action soit simple, nous avons utilisé quelques-unes des fonctionnalités de CakePHP. Nous commençons par utiliser la méthode `findBySlug()` qui est un *finder dynamique*. Cette méthode nous permet de créer une requête basique qui permet de récupérer des articles par un « slug » donné. Nous utilisons ensuite la méthode `firstOrFail()` qui nous permet de récupérer le premier enregistrement ou lancera une `NotFoundException` si aucun article correspondant n'est trouvé.

Notre action attend un paramètre `$slug`, mais d'où vient-il ? Si un utilisateur requête `/articles/view/first-post`, alors la valeur "first-post" sera passée à `$slug` par la couche de routing et de dispatching de CakePHP. Si nous rechargeons notre navigateur, nous aurons une nouvelle erreur, nous indiquant qu'il manque un template de View ; corrigeons cela.

Création du Template View

Créons le template de view pour notre action « view » dans **templates/Articles/view.php**.

```
<!-- Fichier : templates/Articles/view.php -->

<h1><?= h($article->title) ?></h1>
<p><?= h($article->body) ?></p>
<p><small>Créé le : <?= $article->created->format(DATE_RFC850) ?></small></p>
<p><?= $this->Html->link('Modifier', ['action' => 'edit', $article->slug]) ?></p>
```

Vous pouvez vérifier que tout fonctionne en essayant de cliquer sur un lien de `/articles/index` ou en vous rendant manuellement sur une URL de la forme `/articles/view/first-post`.

Ajouter des articles

Maintenant que les views de lecture ont été créées, il est temps de rendre possible la création d'articles. Commencez par créer une action `add()` dans le `ArticlesController`. Notre controller doit maintenant ressembler à ceci :

```php
// src/Controller/ArticlesController.php

namespace App\Controller;

use App\Controller\AppController;

class ArticlesController extends AppController
{

    public function initialize(): void
    {
        parent::initialize();

        $this->loadComponent('Paginator');
        $this->loadComponent('Flash'); // Inclusion du FlashComponent
    }

    public function index()
    {
        $articles = $this->Paginator->paginate($this->Articles->find());
        $this->set(compact('articles'));
    }

    public function view($slug)
    {
        $article = $this->Articles->findBySlug($slug)->firstOrFail();
        $this->set(compact('article'));
    }

    public function add()
    {
        $article = $this->Articles->newEmptyEntity();
        if ($this->request->is('post')) {
            $article = $this->Articles->patchEntity($article, $this->request->getData());

            // L'écriture de 'user_id' en dur est temporaire et
            // sera supprimée quand nous aurons mis en place l'authentification.
            $article->user_id = 1;

            if ($this->Articles->save($article)) {
                $this->Flash->success(__('Votre article a été sauvegardé.'));

                return $this->redirect(['action' => 'index']);
            }
            $this->Flash->error(__('Impossible d\'ajouter votre article.'));
        }
        $this->set('article', $article);
    }
}
```

Note

Vous devez inclure le *Flash* dans tous les controllers où vous avez besoin de l'utiliser. Il est souvent conseillé de le charger directement dans le `AppController`.

Voici ce que l'action `add()` fait :

- Si la méthode HTTP de la requête est un POST, cela tentera de sauvegarder les données en utilisant le model Articles.
- Si pour une quelconque raison la sauvegarde ne se fait pas, cela rendra juste la view. Cela nous donne ainsi une chance de montrer les erreurs de validation ou d'autres messages à l'utilisateur.

Toutes les requêtes de CakePHP incluent un objet request qui est accessible via `$this->request`. L'objet request contient des informations à propos de la requête qui vient d'être reçue. Nous utilisons la méthode *`Cake\Http\ServerRequest::is()`* pour vérifier que la requête possède bien le verbe HTTP POST.

Les données passées en POST sont disponibles dans `$this->request->getData()`. Vous pouvez utiliser les fonctions *`pr()`* ou *`debug()`* pour afficher les données si vous voulez voir à quoi elles ressemblent. Pour sauvegarder les données, nous devons tout d'abord « marshaller » les données du POST en une Entity Article. L'Entity sera ensuite persistée en utilisant la classe ArticlesTable que nous avons créée plus tôt.

Après la sauvegarde de notre article, nous utilisons la méthode `success()` du FlashComponent pour définir le message en Session. La méthode `success` est fournie via les méthodes magiques de PHP [24]. Les messages Flash seront affichés sur la page suivante après redirection. Dans notre layout, nous avons `<?= $this->Flash->render() ?>` qui affichera un message Flash et le supprimera du stockage dans la session. Enfin, après la sauvegarde, nous utilisons *`Cake\Controller\Controller::redirect`* pour renvoyer l'utilisateur à la liste des articles. Le paramètre `['action' => 'index']` correspond à l'URL `/articles`, c'est-à-dire l'action index du `ArticlesController`. Vous pouvez vous référer à la méthode `Cake\Routing\Router::url()` dans la documentation API [25] pour voir les formats dans lesquels vous pouvez spécifier une URL.

Création du Template Add

Voici le code de notre template de la view « add » :

```
<!-- File: templates/Articles/add.php -->

<h1>Ajouter un article</h1>
<?php
    echo $this->Form->create($article);
    // Hard code the user for now.
    echo $this->Form->control('user_id', ['type' => 'hidden', 'value' => 1]);
    echo $this->Form->control('title');
    echo $this->Form->control('body', ['rows' => '3']);
    echo $this->Form->button(__('Sauvegarder l\'article'));
    echo $this->Form->end();
?>
```

Nous utilisons le FormHelper pour générer l'ouverture du formulaire HTML. Voici le HTML que `$this->Form->create()` génère :

```
<form method="post" action="/articles/add">
```

Puisque nous appelons `create()` sans passer d'option URL, le `FormHelper` va partir du principe que le formulaire doit être soumis sur l'action courante.

La méthode `$this->Form->control()` est utilisée pour créer un élément de formulaire du même nom. Le premier paramètre indique à CakePHP à quel champ il correspond et le second paramètre vous permet de définir un très grand

24. https://php.net/manual/en/language.oop5.overloading.php#object.call
25. https://api.cakephp.org

nombre d'options - dans notre cas, le nombre de lignes (rows) pour le textarea. Il y a un peu d'instrospection et de conventions utilisées ici. La méthode `control()` affichera des éléments de formulaire différents en fonction du champ du model spécifié et utilisera une inflection automatique pour définir le label associé. Vous pouvez personnaliser le label, les inputs ou tout autre aspect du formulaire en utilisant les options. La méthode `$this->Form->end()` ferme le formulaire.

Retournons à notre template **templates/Articles/index.php** pour ajouter un lien « Ajouter un article ». Avant le `<table>`, ajoutons la ligne suivante :

```
<?= $this->Html->link('Ajouter un article', ['action' => 'add']) ?>
```

Ajout de la génération de slug

Si nous sauvons un article tout de suite, la sauvegarde échouerait car nous ne créons pas l'attribut « slug » et la colonne correspondante est définie comme `NOT NULL`. Un slug est généralement une version « URL compatible » du titre d'un article. Nous pouvons utiliser le *callback beforeSave()* de l'ORM pour créer notre slug :

```
<?php
// dans src/Model/Table/ArticlesTable.php
namespace App\Model\Table;

use Cake\ORM\Table;
// la classe Text
use Cake\Utility\Text;
// la classe EventInterface
use Cake\Event\EventInterface;

// Ajouter la méthode suivante

public function beforeSave($event, $entity, $options)
{
    if ($entity->isNew() && !$entity->slug) {
        $sluggedTitle = Text::slug($entity->title);
        // On ne garde que le nombre de caractère correspondant à la longueur
        // maximum définie dans notre schéma
        $entity->slug = substr($sluggedTitle, 0, 191);
    }
}
```

Ce code est simple et ne prend pas en compte les potentiels doublons de slug. Mais nous nous occuperons de ceci plus tard.

Ajout de l'action Edit

Notre application peut maintenant sauvegarder des articles, mais nous ne pouvons pas modifier les articles existants. Rectifions cela maintenant. Ajoutez l'action suivante dans votre `ArticlesController` :

```
// dans src/Controller/ArticlesController.php

// Ajouter la méthode suivante.

public function edit($slug)
{
    $article = $this->Articles
```

(suite sur la page suivante)

(suite de la page précédente)

```
        ->findBySlug($slug)
        ->firstOrFail();

    if ($this->request->is(['post', 'put'])) {
        $this->Articles->patchEntity($article, $this->request->getData());
        if ($this->Articles->save($article)) {
            $this->Flash->success(__('Votre article a été mis à jour.'));

            return $this->redirect(['action' => 'index']);
        }
        $this->Flash->error(__('Impossible de mettre à jour l\'article.'));
    }

    $this->set('article', $article);
}
```

Cette action va d'abord s'assurer que l'utilisateur essaie d'accéder à un enregistrement existant. Si vous n'avez pas passé de paramètre `$slug` ou que l'article n'existe pas, une `NotFoundException` sera lancée et le ErrorHandler de CakePHP rendra la page d'erreur appropriée.

Ensuite l'action va vérifier si la requête est une requête POST ou PUT. Si c'est le cas, nous utiliserons alors les données du POST/PUT pour mettre à jour l'entity de l'article en utilisant la méthode `patchEntity()`. Enfin, nous appelons la méthode `save()`, nous définissons un message Flash approprié et soit nous redirigeons, soit nous affichons les erreurs de validation en fonction du résultat de l'opération de sauvegarde.

Création du Template Edit

Le template edit devra ressembler à ceci :

```
<!-- Fichier : templates/Articles/edit.php -->

<h1>Modifier un article</h1>
<?php
    echo $this->Form->create($article);
    echo $this->Form->control('user_id', ['type' => 'hidden']);
    echo $this->Form->control('title');
    echo $this->Form->control('body', ['rows' => '3']);
    echo $this->Form->button(__('Sauvegarder l\'article'));
    echo $this->Form->end();
?>
```

Ce template affiche le formulaire de modification (avec les valeurs déjà remplies), ainsi que les messages d'erreurs de validation nécessaires.

Vous pouvez maintenant mettre à jour notre view index avec les liens pour modifier les articles :

```
<!-- Fichier : templates/Articles/index.php (liens de modification ajoutés) -->

<h1>Articles</h1>
<p><?= $this->Html->link("Ajouter un article", ['action' => 'add']) ?></p>
<table>
    <tr>
        <th>Titre</th>
        <th>Créé le</th>
```

(suite sur la page suivante)

(suite de la page précédente)

```
        <th>Action</th>
    </tr>

    <!-- C'est ici que nous bouclons sur notre objet Query $articles pour afficher les␣
↪informations de chaque article -->

<?php foreach ($articles as $article): ?>
    <tr>
        <td>
            <?= $this->Html->link($article->title, ['action' => 'view', $article->slug])␣
↪?>
        </td>
        <td>
            <?= $article->created->format(DATE_RFC850) ?>
        </td>
        <td>
            <?= $this->Html->link('Modifier', ['action' => 'edit', $article->slug]) ?>
        </td>
    </tr>
<?php endforeach; ?>

</table>
```

Mise à jour des règles de validation pour les Articles

Jusqu'à maintenant, nos Articles n'avaient aucune validation de données. Occupons-nous de ça en utilisant un *validator* :

```
// src/Model/Table/ArticlesTable.php

// Ajouter ce "use" juste sous la déclaration du namespace pour importer
// la classe Validator
use Cake\Validation\Validator;

// Ajouter la méthode suivante.
public function validationDefault(Validator $validator): Validator
{
    $validator
        ->notEmptyString('title')
        ->minLength('title', 10)
        ->maxLength('title', 255)

        ->notEmptyString('body')
        ->minLength('body', 10);

    return $validator;
}
```

La méthode `validationDefault()` indique à CakePHP comment valider les données quand la méthode `save()` est appelée. Ici, il est spécifié que les champs title et body ne peuvent pas être vides et qu'ils ont aussi des contraintes sur la longueur.

Le moteur de validation de CakePHP est à la fois puissant et flexible. Il vous fournit un jeu de règles sur des validations

communes comme les adresses emails, les adresses IP, etc. mais aussi la flexibilité d'ajouter vos propres règles de validation. Pour plus d'informations, rendez-vous dans la section *Validation* de la documentation.

Maintenant que nos règles de validation sont en place, utilisons l'application et essayons d'ajouter un article avec un title ou un body vide pour voir ce qu'il se passe. Puisque nous avons utiliser la méthode `Cake\View\Helper\FormHelper::control()` du FormHelper pour créer les éléments de formulaire, nos messages d'erreurs de validation seront affichés automatiquement.

Ajout de l'Action de Suppression

Donnons maintenant la possibilité à nos utilisateurs de supprimer des articles. Commencez par créer une action `delete()` dans `ArticlesController` :

```
// src/Controller/ArticlesController.php

// Ajouter la méthode suivante.

public function delete($slug)
{
    $this->request->allowMethod(['post', 'delete']);

    $article = $this->Articles->findBySlug($slug)->firstOrFail();
    if ($this->Articles->delete($article)) {
        $this->Flash->success(__('L\'article {0} a été supprimé.', $article->title));

        return $this->redirect(['action' => 'index']);
    }
}
```

Ce code va supprimer l'article ayant le slug `$slug` et utilisera la méthode `$this->Flash->success()` pour afficher un message de confirmation à l'utilisateur après l'avoir redirigé sur `/articles`. Si l'utilisateur essaie d'aller supprimer un article avec une requête GET, la méthode `allowMethod()` lancera une exception. Les exceptions non capturées sont récupérées par le gestionnaire d'exception de CakePHP qui affichera une belle page d'erreur. Il existe plusieurs *Exceptions* intégrées qui peuvent être utilisées pour remonter les différentes erreurs HTTP que votre application aurait besoin de générer.

> **Avertissement**
>
> Permettre de supprimer des données via des requêtes GET est très dangereux, car il est possible que des crawlers suppriment accidentellement du contenu. C'est pourquoi nous utilisons la méthode `allowMethod()` dans notre controller.

Puisque nous exécutons seulement de la logique et redirigeons directement sur une autre action, cette action n'a pas de template. Vous devez ensuite mettre à jour votre template index pour ajouter les liens qui permettront de supprimer les articles :

```
<!-- Fichier : templates/Articles/index.php (ajout des liens de suppression) -->

<h1>Articles</h1>
<p><?= $this->Html->link("Add Article", ['action' => 'add']) ?></p>
<table>
    <tr>
        <th>Titre</th>
```

(suite sur la page suivante)

(suite de la page précédente)

```
        <th>Créé le</th>
        <th>Action</th>
    </tr>

    <!-- C'est ici que nous bouclons sur notre objet Query $articles pour afficher les␣
↪informations de chaque article -->

<?php foreach ($articles as $article): ?>
    <tr>
        <td>
            <?= $this->Html->link($article->title, ['action' => 'view', $article->slug])␣
↪?>
        </td>
        <td>
            <?= $article->created->format(DATE_RFC850) ?>
        </td>
        <td>
            <?= $this->Html->link('Modifier', ['action' => 'edit', $article->slug]) ?>
            <?= $this->Form->postLink(
                'Supprimer',
                ['action' => 'delete', $article->slug],
                ['confirm' => 'Êtes-vous sûr ?'])
            ?>
        </td>
    </tr>
<?php endforeach; ?>

</table>
```

Utiliser `postLink()` va créer un lien qui utilisera du JavaScript pour faire une requête POST et supprimer notre article.

Note

Ce code de view utilise également le `FormHelper` pour afficher à l'utilisateur une boîte de dialogue de confirmation en JavaScript avant la suppression effective de l'article.

Maintenant que nous avons un minimum de gestion sur nos articles, il est temps de créer des actions basiques pour nos tables *Tags et Users*.

CHAPITRE 3

Guides de Migration

Les guides de migration contiennent des informations à propos des nouvelles fonctionnalités introduites dans chaque version et le chemin de migration entre 3.x et 4.x.

Guide de mise à jour pour la version 4.0

Tout d'abord, vérifiez que votre application fonctionne sur la dernière version de CakePHP 3.x.

> **Note**
>
> L'outil de mise à jour ne fonctionne que sur les applications exécutées sur la dernière version de CakePHP 3.x. Vous ne pouvez pas exécuter l'outil de mise à jour après la mise à jour vers CakePHP 4.0.

Corrigez les avertissements de dépréciation

Une fois que votre application s'exécute sur la dernière version de CakePHP 3.x, activez les avertissements d'obsolescence dans **config/app.php** :

```
'Error' => [
    'errorLevel' => E_ALL,
]
```

Maintenant que vous pouvez voir tous les avertissements, assurez-vous qu'ils sont corrigés avant de procéder à la mise à niveau.

Passez à PHP 7.2

Si vous n'utilisez pas **PHP 7.2 or higher**, vous devrez mettre à jour PHP avant de mettre à jour CakePHP.

> **Note**
>
> CakePHP 4.0 nécessite **a minimum of PHP 7.2**.

Utilisez l'outil de mise à niveau

Parce que CakePHP 4 adopte le mode strict et utilise plus de typehinting, il existe de nombreuses modifications incompatibles avec les versions précédentes concernant les signatures de méthode et les renommages de fichiers. Pour accélérer la résolution de ces modifications fastidieuses, il existe un outil CLI de mise à niveau :

> **Avertissement**
>
> La commande `file_rename` et les règles de l'outil rector for cakephp40, et phpunit80 sont destinés à être exécutés **avant** que vous n'effectuiez la mise à jour des dépendances de votre application vers 4.0. Les règles du recteur `cakephp40` ne fonctionneront pas correctement si votre application a déjà ses dépendances mises à jour vers 4.x ou PHPUnit8.

```
# Installe l'outil d'upgrade
git clone https://github.com/cakephp/upgrade
cd upgrade
git checkout 4.x
composer install --no-dev
```

Une fois l'outil de mise à niveau installé, vous pouvez maintenant l'exécuter sur votre application ou votre plugin :

```
# Renomme les fichiers locaux
bin/cake upgrade file_rename locales <path/to/app>

# Renomme les fichiers de template
bin/cake upgrade file_rename templates <path/to/app>
```

Une fois que vous avez renommé vos fichiers de template et de traduction, assurez-vous de mettre à jour les chemins `App.paths.locales` et `App.paths.templates` (dans **/config/app.php**) avec les bonnes valeurs. Si besoin, référez-vous à la [configuration du squelette d'une application](https://github.com/cakephp/app/blob/master/config/app.php)

Appliquez le Refactorings de Rector

Ensuite, utilisez la commande `rector` pour corriger automatiquement de nombreux appels de méthodes dépréciés dans CakePHP et PHPUnit. Il est important d'appliquer rector **avant** la mise à niveau vos dépendances :

```
bin/cake upgrade rector --rules phpunit80 <path/to/app/tests>
bin/cake upgrade rector --rules cakephp40 <path/to/app/src>
```

Vous pouvez également utiliser l'outil de mise à niveau pour appliquer de nouvelles règles de rector pour chaque version mineure de :

```
# Exécutez les règles du recteur pour la mise à niveau 4.0 -> 4.1.
bin/cake upgrade rector --rules cakephp41 <path/to/app/src>
```

Mettez à jour la dépendance de CakePHP

Après avoir appliqué les refactorisations de rector, mettez à niveau CakePHP et PHPUnit en utilisant les commandes suivantes de composer :

```
php composer.phar require --dev --update-with-dependencies "phpunit/phpunit:^8.0"
php composer.phar require --update-with-dependencies "cakephp/cakephp:4.0.*"
```

Application.php

Ensuite, assurez-vous que votre `src/Application.php` a été mis à jour pour avoir les mêmes signatures de méthodes que celles trouvées dans cakephp/app. vous trouverez la dernière version de Application.php [26] sur GitHub.

Si vous fournissez une API de type REST, n'oubliez pas d'inclure le body-parser-middleware. Enfin, vous devriez envisager de passer aux nouveaux AuthenticationMiddleware et AuthorizationMiddleware, si vous utilisez encore le composant `AuthComponent`.

Guide de migration vers la version 4.0

CakePHP 4.0 contient des changements non rétro-compatibles et n'est pas rétrocompatible avec 3.x communiqués. Avant d'essayer de mettre à niveau vers la version 4.0, effectuez d'abord la mise à niveau vers la version 3.8 et résolvez tous les avertissements d'obsolescence.

Référez-vous à *Guide de mise à jour pour la version 4.0* afin de suivre pas à pas les instructions concernant la façon de migrer vers la version 4.0.

Fonctionnalités obsolètes supprimées

Toutes les méthodes, propriétés et fonctionnalités qui émettaient des avertissements d'obsolescence à partir de 3.8 ont été supprimés.

La fonctionnalité d'authentification a été divisée en plugins autonomes Authentication [27] et Authorization [28]. L'ancien RssHelper -avec des fonctionnalités similaires- se trouve a présent dans le plugin Feed plugin [29] .

Dépréciations

Voici une liste des méthodes, propriétés et comportements obsolètes. Ces fonctionnalités continueront de fonctionner dans 4.x et seront supprimées dans la version 5.0.0.

Component

- `AuthComponent` et les classes associées sont devenues obsolètes et seront supprimées dans 5.0.0. Vous devez utiliser les bibliothèques d'authentification et d'autorisation mentionnées ci-dessus à la place.
- `SecurityComponent` est déprécié. A sa place, utilisez le `FormProtectionComponent` pour la protection contre la falsification de formulaire et le *Middleware HTTPS Enforcer* pour la fonctionnalité `requireSecure`.

Filesystem

- Ce package est obsolète et sera supprimé dans la version 5.0. Il pose un certain nombre de problèmes de conception et la maintenance de ce package rarement utilisé ne semble pas valoir l'effort alors qu'il existe une grande variété d'autres packages.

26. https://github.com/cakephp/app/blob/4.x/src/Application.php
27. https://github.com/cakephp/authentication
28. https://github.com/cakephp/authorization
29. https://github.com/dereuromark/cakephp-feed

ORM

- L'utilisation `Entity::isNew()` comme mutateur est dépréciée. Utilisez `setNew()` à la place.
- `Entity::unsetProperty()` a été renommée en `Entity::unset()` pour être conforme avec les autres méthodes.
- `TableSchemaInterface::primaryKey()` a été renommée en `TableSchemaInterface::getPrimaryKey()`.

View

- Les variables de la vue `JsonView` : `_serialize`, `_jsonOptions` and `_jsonp` sont dépréciées. A la place, vous devez utiliser `viewBuilder()->setOption($optionName, $optionValue)` pour définir ces options.
- Les variables de la vue `XmlView` : `_serialize`, `_rootNode` and `_xmlOptions` sont dépréciées. A la place, vous devez utiliser `viewBuilder()->setOption($optionName, $optionValue)` pour définir ces options.
- `HtmlHelper::tableHeaders()` préfère désormais que les cellules d'en-tête avec des attributs soient défini comme une liste imbriquée. Par exemple `['Title', ['class' => 'special']]`.
- `ContextInterface::primaryKey()` a été renommée en `ContextInterface::getPrimaryKey()`.

Mailer

- La classe `Cake\Mailer\Email` a été dépréciée. Utilisez `Cake\Mailer\Mailer` à sa place.

App

- `App::path()` a été dépréciée pour les chemins de classes (class paths). Utilisez `\Cake\Core\App::classPath()` à sa place.

Changements non rétro-compatibles

En plus de la suppression des fonctionnalités obsolètes, des changements non rétro-compatibles ont été effectués :

Cache

- `Cake\Cache\Cache::read()` renvoie `null` au lieu de `false` si les données n'existent pas.
- `Cake\Cache\CacheEngine::gc()` et toutes les implémentations de cette méthode ont été supprimées. Cette méthode était interdite dans la plupart des pilotes de cache et n'était utilisée que pour la mise en cache de fichiers.

Controller

- `Cake\Controller\Controller::referer()` met par défaut le paramètre `local` à true, au lieu de false. Cela rend l'utilisation des referer dans les headers plus sûre car ils seront limités au domaine de votre application par défaut.
- La mise en correspondance du nom de la méthode du contrôleur lors de l'appel d'actions est désormais sensible à la casse. Par exemple, si votre méthode de contrôleur est `ForgotPassword()`, utiliser la chaîne de caractères `Forgotpassword` dans l'URL ne correspondra pas au nom de l'action.

Console

- `ConsoleIo::styles()` a été séparée en `getStyle()` et `setStyle()`. Cela se reflète également dans `ConsoleOutput`.

Component

- `Cake\Controller\Component\RequestHandlerComponent` assigne à présent `isAjax` a un attribut de la requête à la place d'un paramètre de la requête. Par conséquent, vous devez maintenant utiliser `$request->getAttribute('isAjax')` à la place de `$request->getParam('isAjax')`.

- Les fonctionnalités de parsing du corps de requête de `RequestHandlerComponent` ont été supprimées. Vous devez utiliser body-parser-middleware à la place.
- `Cake\Controller\Component\PaginatorComponent` définit maintenant les informations des paramètres de pagination comme attribut de la requête à la place d'un paramètre de la requête. Par conséquent, vous devez à présent utiliser `$request->getAttribute('paging')` à la place de `$request->getParam('paging')`.

Database

- Les classes permettant le mapping de type dans `Cake\Database\TypeInterface` n'héritent plus de `Type`, et tirent à présent partie des fonctionnalités de `BatchCastingInterface`.
- `Cake\Database\Type::map()` s'uitlise uniquement comme un setter maintenant. Vous devez utiliser `Type::getMap()` pour inspecter le type des instances.
- Les types de colonnes Date, Time, Timestamp, et Datetime retournent à présent des objets de temps immuables (immutable time) par défaut.
- `BoolType` ne transforme plus les valeurs de chaînes de caractères non vides à `true` et les valeurs des chaines vides à `false`. Au lieu de cela, les valeurs de chaînes de caractères non booléennes sont converties à `null`.
- `DecimalType` utilise désormais des chaînes de caractères pour représenter des valeurs décimales au lieu de flottants. L'utilisation de flottants entrainait une perte de précision..
- `JsonType` préserve desormais `null` dans le contexte de préparation des valeurs pour l'écriture en base de données. Dans la version 3.x il envoyait la chaîne `'null'`.
- `StringType` transforme à présent les tableaux en `null` à la place d'une chaîne de caractère vide.
- `Cake\Database\Connection::setLogger()` n'accepte plus `null` pour désactiver la journalisation. Passez plutôt une instance de `Psr\Log\NullLogger` pour désactiver la journalisation.
- Les implémentations internes de `Database\Log\LoggingStatement`, `Database\QueryLogger` et `Database\Log\LoggedQuery` ont changé. Si vous étendez ces classes, vous devrez mettre à jour votre code.
- Les implémentations internes de `Cake\Database\Log\LoggingStatement`, `Cake\Database\QueryLogger` et `Cake\Database\Log\LoggedQuery` ont changé. Si vous étendez ces classes, vous devrez mettre à jour votre code.
- Les implémentations internes de `Cake\Database\Schema\CacheCollection` et `Cake\Database\SchemaCache` ont changé. Si vous étendez ces classes, vous devrez mettre à jour votre code.
- `Cake\Database\QueryCompiler` ne place plus les clauses `SELECT` entre quotes que lorsque l'auto-quoting est activé. La mise entre quotes est conservée pour Postgres afin d'éviter que les identifiers ne soient castés automatiquement en minuscules.
- Le shéma de base de donnée fait à présent correspondre les colonnes de type `CHAR` au nouveau type `char` à la place de du type `string`.
- Dans SqlServer le type de colonne datetime correspond à présent au type "datetime" plutôt qu'au type "timestamp".
- Les shémas pour les bases de données de type MySQL, PostgreSQL and SqlServer font correspondrent les colonnes supportant les secondes fractionnaires (fractional seconds) au nouveau type abstrait fractionnaire.
  - **MySQL**
    1. `DATETIME(1-6) => datetimefractional`
    2. `TIMESTAMP(1-6) => timestampfractional`
  - **PostgreSQL**
    1. `TIMESTAMP => timestampfractional`
    2. `TIMESTAMP(1-6) => timestampfractional`
  - **SqlServer**
    1. `DATETIME2 => datetimefractional`
    2. `DATETIME2(1-7) => ``datetimefractional`
- Le schéma PostgreSQL mappe désormais les colonnes prenant en charge les fuseaux horaires avec le nouveau types abstrait de fuseaux horaires. Spécifier (0) comme précision ne modifie pas le mappage de type comme il le fait avec les types fractionnaires réguliers ci-dessus.
  - **PostgreSQL**

1. `TIMESTAMPTZ` => `timestamptimezone`
2. `TIMESTAMPTZ(0-6)` => `timestamptimezone`
3. `TIMESTAMP WITH TIME ZONE` => `timestamptimezone`
4. `TIMESTAMP(0-6) WITH TIME ZONE` => `timestamptimezone`

Datasources

— `ModelAwareTrait::$modelClass` est a présent protégé.

Error

— **Les implémentations internes des classes de gestionnaire d'erreurs `BaseErrorHandler`,** `ErrorHandler` et `ConsoleErrorHandler` ont changées. Si vous avez étendu ces classes vous devez les mettre à jour en conséquence.
— `ErrorHandlerMiddleware` prend maintenant un nom de classe ou une instance de gestionnaire d'erreurs (error handler) comme argument de constructeur au lieu du nom ou de l'instance de la classe d'exception (exception render class) à rendre.

Event

— Appeler `getSubject()` sur un évènement (event) qui ne possède pas d'attribut *subject* ` provoquera à présent une exception.

Http

— `Cake\Controller\Controller::referer()` met par défaut le paramètre `local` à true, au lieu de false. Cela rend l'utilisation des referer dans les headers plus sûre car ils seront limités au domaine de votre application par défaut.
— La valeur par défaut de `Cake\Http\ServerRequest::getParam()` quand un paramètre est manquant est maintenant `null` et non `false`.
— `Cake\Http\Client\Request::body()` a été supprimée. Utilisez `getBody()` ou `withBody()` à la place.
— `Cake\Http\Client\Response::isOk()` retourne à présent `true` pour les codes de response 2xx and 3xx.
— `Cake\Http\Cookie\Cookie::getExpiresTimestamp()` retourne à présent un entier. Cela fait correspondre le type à celui utilisé dans `setcookie()`.
— `Cake\Http\ServerRequest::referer()` retourne à présent `null` quand la requête courante ne possède pas de referer. Auparavant, elle retournait `/`.
— `Cake\Cookie\CookieCollection::get()` lève maintenant une exception lors de l'accès à un cookie qui n'existe pas. Utilisez `has()` pour vérifier l'existence des cookies.
— La signature de `Cake\Http\ResponseEmitter::emit()` a changé, elle ne possède plus de 2nd argument.
— La valeur par défaut de `App.uploadedFilesAsObjects` est à présent `true`. Si votre application utilise l'upload de fichiers vous pouvez mettre ce flag à `false` afin de préserver la compatibilité avec le comportement de la version 3.x.
— Les clés retournées par `Cake\Http\Response::getCookie()` ont changé. `expire` est remplacé par `expires` et `httpOnly` par `httponly`.

HttpSession

— Le nom du cookie de session n'est plus défini comme `CAKEPHP` par défaut. A la place, le nom de cookie par défaut est celui défini dans votre fichier ``php.ini`. Vous pouvez utiliser l'option de configuration `Session.cookie` pour définir le nom du cookie.
— Les cookies de session ont désormais l'attribut `SameSite` défini comme `Lax` par défaut. Jetez un oeil à *Configuration de Session* pour d'avantage d'informations.

I18n

— L'encodage JSON des objets `Cake\I18n\Date` et `Cake\I18n\FrozenDate` produit maintenant des chaînes de caractères qui possèdent uniquement la partie concernant la date au format `yyyy-MM-dd` au lieu du `yyyy-MM-dd'T'HH:mm:ssxxx` précédemment.

Mailer

— `Email::set()` a été supprimée. Utilisez `Email::setViewVars()` à la place.
— `Email::createView()` a été supprimée.
— `Email::viewOptions()` a été supprimée. Utilisez `$email->getRenderer()->viewBuilder()->setOptions()` à la place.

ORM

— `Table::newEntity()` nécessite maintenant un tableau en entrée et applique la validation pour empêcher des sauvegardes accidentelles sans que la validation ne soit déclenchée. Cela signifie que vous devez utiliser `Table::newEmptyEntity()` pour créer des entités vides.
— Utiliser des conditions semblables à `['name' => null]` pour `Query::where()` va maintenant lever une exception. Dans 3.x, cela générerait une condition SQL `name = NULL` qui correspond toujours à 0 ligne, renvoyant ainsi des résultats incorrects. Pour comparer avec `null` vous devez utiliser l'opérateur `IS` de la façon suivante `['name IS' => null]`.
— Stopper l'évènement `Model.beforeSave` en renvoyant un résultat non nul ou qui n'est pas une entité (entity)va maintenant lever une exception. Ce changement garantit que `Table::save()` renverra toujours une entité ou false.
— Les objets Table lèveront désormais une exception lorsque les alias générés pour les noms et la colonne de la table seraient tronqués par la base de données. Cela avertit l'utilisateur avant que des erreurs cachées (hidden errors) ne se produisent lorsque CakePHP ne peut pas faire correspondre l'alias dans le résultat.
— `TableLocator::get()` et `TableRegistry::get()` s'attendent maintenant à ce que les alias des noms soient toujours **CamelCased** dans votre code. Passer des alias avec la mauvaise casse entraînera un chargement incorrect des classes de table et d'entité.
— La règle `IsUnique` n'accepte plus l'option `allowMultipleNulls` qui était activée par défaut. Ceci a été réajouté dans 4.2 mais désactivé par défaut.

Router

— Les préfixes de routage créés via `Router::prefix()` et `$routes->prefix()` sont à présent CamelCased et non plus under_scored. A la place de `my_admin`, vous devez utiliser `MyAdmin`. Ce changement normalise les préfixes avec les autres paramètres de routage et supprime la surcharge causée par l'inflexion.
— `RouteBuilder::resources()` infléchit maintenant les noms de ressources à une forme dasherized au lieu d'être souligné par défaut dans les URL. Vous pouvez conserver la forme soulignée en utilisant `'inflect' => 'underscore'` dans l'argument `$options`.
— `Router::plugin()` et `Router::prefix()` utilisent à présent le nom dasherized du plugin/prefix par défaut dans l'URL. Vous pouvez conserver la forme soulignée (ou toute autre forme de chemin personnalisée) en utilisantla clé `'path'` dans l'argument `$options`.
— `Router` maintient à présent la référence à une seule instance de requête à la place d'une pile des demandes. `Router::pushRequest()`, `Router::setRequestInfo()` et `Router::setRequestContext()` ont été supprimées, utilisez `Router::setRequest()` à la place. `Router::popRequest()` a été supprimée. `Router::getRequest()` ne possède plus d'argument `$current`.
— `Router::url()` et toutes les méthodes de génération de routes (`HtmlHelper::link()`, `UrlHelper::build()`, ...) ne déplaceront plus automatiquement les variables inconnues dans le `?`. `Router::url(['_name' => 'route', 'c' => 1234])` doit être réécrit en `Router::url(['_name' => 'route', '?' => ['c' => 1234]])`.

TestSuite

— `Cake\TestSuite\TestCase::$fixtures` ne peut pas être une chaîne séparée par des virgules plus. Ce doit être un tableau..

Utility

— `Cake\Utility\Xml::fromArray()` nécessite maintenant un tableau pour le paramètre `$options`.
— `Cake\Filesystem\Folder::copy($to, array $options = [])` et `Cake\Filesystem\Folder::move($to, array $options = [])` ont maintenant le chemin cible extrait comme premier argument..
— L'option `readFile` de `Xml::build()` n'a plus la valeur true par défaut. Au lieu de cela, vous devez activer `readFile` pour lire les fichiers locaux.
— `Hash::sort()` accepte désormais les constantes `SORT_ASC` et `SORT_DESC` comme paramètre de direction.
— `Inflector::pluralize()` infléchit maintenant `index` à `indexes` au lieu de``indices``. Cela reflète l'utilisation technique de ce pluriel dans le noyau ainsi que dans l'écosystème.

View

— Les modèles (Templetes) ont été déplacés du dossier `src/Template/` vers le dossier `templates/` à la racine de l'application et des plugins. Avec cette modification, le dossier `src` ne contient plus que des fichiers avec des classes qui sont chargées automatiquement via l'autoloader de composer.
— Les dossiers de modèles spéciaux comme `Cell`, `Element`, `Email`, `Layout` et `Plugin` ont été renommés en minuscules `cell`, `element`, `email`, `layout` and `plugin` respectivement. Cela permet une meilleure distinction visuelle entre les dossiers spéciaux et les dossiers correspondant aux noms de contrôleurs de votre application qui eux sont exprimés sous la forme `CamelCase`.
— L'extension des fichiers de Template a été modifiée `.ctp` à `.php`. L'extension spéciale n'a fourni aucun avantage réel et a plutôt nécessité que les éditeurs/IDE soient configurés pour reconnaître les fichiers avec l'extension `.ctp` en tant que fichiers PHP.
— Vous ne pouvez plus utiliser `false` comme argument pour `ViewBuilder::setLayout()` ou `View::setLayout()` pour définir la propriété `View::$layout` à `false`. Utilisez plutôt `ViewBuilder::disableAutoLayout()` et `View::disableAutoLayout()` pour rendre un modèle de vue sans mise en page.
— `Cake\View\View` re-rendra les vues au lieu de retourner `null` si `render()` est appelée plusieurs fois.
— Les constantes `View::NAME_ELEMENT` et `View::NAME_LAYOUT` ont été supprimées. Vous pouvez utiliser `View::TYPE_ELEMENT` et `View::TYPE_LAYOUT`.

Helper

— Les arguments de `Cake\View\Helper\PaginatorHelper::hasPage()` ont été intervertis. Cela la rend cohérente avec les autres méthodes de pagination pour lesquelles le "modèle" est le deuxième argument.
— `Cake\View\Helper\UrlHelper::build()` n'accepte plus un booléen pour le deuxième paramètre. Vous devez utiliser `['fullBase' => true]` à la place.
— Vous devez maintenant utiliser uniquement `null` comme 1er argument de `FormHelper::create()` pour créer un formulaire sans contexte. Passer toute autre valeur pour laquelle le contexte ne peut pas être déduit entraînera la levée d'une exception.
— `Cake\View\Helper\FormHelper` et `Cake\View\Helper\HtmlHelper` utilisent à présent l'attribut de donnée HTML `data-confirm-message` afin de conserver le message de confirmation pour les méthodes qui ont l'option `confirm`.
— `Cake\View\Helper\FormHelper::button()` encode à présent par défaut sous forme d'entités HTML le texte des boutons ainsi que les attrinuts HTML. Une nouvelle option `escapeTitle` a été ajouté pour permettre de contrôler l'échappement du titre séparément des autres attributs HTML.
— `Cake\View\Helper\SecureFieldTokenTrait` a été supprimé. Sa fonctionnalité permettant de construire des jetons de formulaires à partir des données est désormais incluse dans la classe interne `FormProtector`.

— La méthode `HtmlHelper::docType()` a été supprimée. HTML4 et XHTML sont maintenant obsolètes et doctype pour HTML5 est court à taper directement.
— L'option `safe` pour `HtmlHelper::scriptBlock()` et `HtmlHelper::scriptStart()` a été retiré. Lorsqu'il était activé, il générait des tags `CDATA` qui ne sont nécessaires que pour XHTML qui est maintenant obsolète..

Log

— Les méthodes relatives au Logging comme `Cake\Log\LogTrait::log()`, `Cake\Log\Log::write()` etc. n'acceptent désormais plus que des chaînes de caractère comme argument `$message`. Ce changement était nécessaire pour aligner l'API avec le standard PSR-3 [30].

Miscellaneous

— Le fichier `config/bootstrap.php` de votre application doit maintenant contenir un appel à `Router::fullBaseUrl()`. Consultez le dernier squelette d'application `bootstrap.php` et mettez le votre à jour en conséquence.
— `App::path()` utilise mainetant `$type` et `templates` à la place de `Template` pour obtenir le chemin d'accès des templates. De même, `locales` est utilisé au lieu de `Locale` pour obtenir le chemin diu dossier contenant les traductions.
— `ObjectRegistry::get()` lève maintenant une exception si l'objet avec le nom fourni n'est pas chargé. Vous devez utiliser `ObjectRegistry::has()` pour vous assurer que l'objet existe dans le registre. Le getter magique `ObjectRegistry::__get()` continuera à retourner `null` si l'objet correspondant au nom n'est pas chargé.
— Les fichiers de traduction (Locale) ont été déplacés de `src/Locale` vers `resources/locales`.
— Le fichier `cacert.pem` qui était fourni dans CakePHP a été remplacé par une dépendance vers composer/ca-bundle [31].

Nouvelles fonctionnalités

Console

— Les classes de commande peuvent implémenter la méthode `defaultName()` pour remplacer le nom CLI basé sur les conventions.

Core

— `InstanceConfigTrait::getConfigOrFail()` et `StaticConfigTrait::getConfigOrFail()` ont été ajoutées. Comme les autres `orFail` méthodes ces méthodes lèveront une exception lorsque la clé demandée n'existe pas ou possède la valeur `null`.

Database

— Si le fuseau horaire de votre base de données ne correspond pas au fuseau horaire PHP, vous pouvez utiliser `DateTime::setDatabaseTimezone()`. Référez-vous à *datetime-type* pour plus de détails.
— `DateTime::setKeepDatabaseTimezone()` vous permet de conserver le fuseau horaire de la base de données dans les objets DateTime créés par des requêtes..
— `Cake\Database\Log\LoggedQuery` implémente à présent `JsonSerializable`.
— `Cake\Database\Connection` permet désormais d'utiliser n'importe quel logger PSR-3. Par conséquent ceux qui utilisent le package de base de données autonome ne sont plus obligés d'utiliser le paquet `cakephp/log` pour la journalisation.
— `Cake\Database\Connection` permet désormais d'utiliser n'importe quel cache PSR-16. Par conséquent ceux qui utilisent le package de base de données autonome ne sont plus obligés d'utiliser le paquet `cakephp/cache` pour la mise en cache. Les nouvelles méthodes `Cake\Database\Connection::setCacher()` et `Cake\Database\Connection::getCacher()` ont été ajoutées.

30. https://www.php-fig.org/psr/psr-3/
31. https://packagist.org/packages/composer/ca-bundle

— `Cake\Database\ConstraintsInterface` a été extraite de `Cake\Datasource\FixtureInterface`. Cette interface doit être implémentée par les les implémentations de fixture qui supportent les contraintes, ce qui d'après notre expérience est généralement le cas des bases de données relationelles.
— Le type abstrait `char` a été ajouté. Ce type gère les colonnes de types "caractères de longueur fixe".
— Les types abstraits `datetimefractional` et `timestampfractional` ont été ajoutés. Ces types gèrent les colonnes de types "secondes décimales".
— Les schémas SqlServer prennent désormais en charge les valeurs par défaut avec des fonctions comme SYSDATETIME().
— Les types abstraits `datetimetimezone` et `timestamptimezone` ont été ajoutés. Ces types gèrent les colonnes de types supportant la gestion du fuseau horaire (time zone).

Error

— Si une erreur est déclenchée par une action du contrôleur dans une route préfixée, `ErrorController` tentera d'utiliser un modèle d'erreur préfixé s'il y en a un disponible. Ce comportement n'est appliqué que lorsque `debug` est désactivé.

Http

— Vous pouvez utiliser `cakephp/http` sans inclure le framework complet.
— CakePHP supporte désormais la spécification PSR-15: HTTP Server Request Handlers [32]. En conséquence, les middlewares implémentent désormais `Psr\Http\Server\MiddlewareInterface`. Les middlewares invocables à double passe de CakePHP 3.x sont toujours pris en charge afin d'assurer la compatibilité ascendante.
— `Cake\Http\Client` suit à présent la spécification PSR-18: HTTP Client [33].
— `Cake\Http\Client\Response::isSuccess()` a été ajoutée. Cette méthode renvoie true si le code d'état de la réponse est 2xx
— `CspMiddleware` a été ajouté afin de simplifier la définition de la stratégie de sécurité des contenus dans les en-têtes (Content Security Policy headers).
— `HttpsEnforcerMiddleware` a été ajouté. Il remplace la fonction `requireSecure` du composant `SecurityComponent`.
— Les cookies prennent désormais en charge l'attribut `SameSite`.

I18n

— `Date` et `FrozenDate` respectent désormais le paramètre de fuseau horaire pour divers assistants comme `today('Asia/Tokyo')`.

Mailer

— La responsabilité de la génération des e-mails a maintenant été transférée à `Cake\Mailer\Renderer`. Il s'agit principalement d'un changement architectural et n'a pas d'impact sur la façon dont la classe `Email` est utilisée. La seule différence est que vous avez maintenant besoin d'utiliser `Email::setViewVars()` au lieu de `Email::set()` pour définir les variables de templates.

ORM

— La méthode `Table::saveManyOrFail()` qui a été ajoutée lèvera une exception `PersistenceFailedException` mentionnant en cas d'erreur l'entité dont l'enregistrement a échoué. Les entités sont enregistrées au sein d'une transaction.
— Les méthodes `Table::deleteMany()` et `Table::deleteManyOrFail()` ont été ajoutées pour permettre la suppression de plusieurs entités à la fois en incluant des callbacks. Les entités sont supprimées au sein d'une transaction.

32. https://www.php-fig.org/psr/psr-15/
33. https://www.php-fig.org/psr/psr-18/

— `Table::newEmptyEntity()` a été ajoutée pour créer une nouvelle entité vide. Cela ne déclenche aucune validation de champ. L'entité peut être persistée sans erreur de validation en tant qu'enregistrement vide.
— `Cake\ORM\RulesChecker::isLinkedTo()` et `isNotLinkedTo()` ont été ajoutées. Ces nouvelles règles d'application vous permettent de vous assurer qu'une association possède ou non des enregistrements connexes.
— Une nouvelle classe de type `DateTimeFractionalType` a été ajoutée pour les types datetime avec une précision de l'ordre de la microseconde. Vous pouvez choisir d'utiliser ce type en l'ajoutant au `TypeFactory` comme type par défaut pour le type `datetime` ou en re-mappant chaque type de colonne. Voir les notes de migration de base de données pour savoir comment ce type est automatiquement mappé aux types de base de données.
— Une nouvelle classe de type `DateTimeTimezoneType` a été ajoutée pour les types datetime supportant la prise en charge du fuseau horaire. Vous pouvez choisir d'utiliser ce type en l'ajoutant au `TypeFactory` comme type par défaut pour le type `datetime` ou en re-mappant chaque type de colonne. Voir les notes de migration de base de données pour savoir comment ce type est automatiquement mappé aux types de base de données.

Routing

— `Cake\Routing\Asset` a été ajoutée. Cette classe expose la génération d'URL de ressources via une interface statique similaire à `Router::url()`. Voir *Générer des URL de ressources* pour plus d'information.

TestSuite

— `TestSuite\EmailTrait::assertMailContainsAttachment()` a été ajouté.

Validation

— `Validation::dateTime()` accepte désormais les valeurs en microsecondes.

View

— `FormHelper` génère désormais des messages de validation HTML5 pour les champs marqués comme « notEmpty » dans la classe Table correspondant à l'entité. Cette fonction peut être activée grâce à l'option de configuration `autoSetCustomValidity` de la classe.
— `FormHelper` génère désormais des balises d'entrée HTML5 natives pour les champs datetime. Consultez la page *Form Helper <create-datetime-controls>* pour plus de détails. Si vous devez conserver l'ancien balisage, un FormHelper calé peut être trouvé dans Shim plugin [34] avec l'ancien behavior/generation (4.x branch).
— `FormHelper` définit maintenant la taille de l'incrément par défaut en secondes pour les widgets `datetime` qui possèdent une composante de temps. La valeur par défaut est de millisecondes si le champ provient des nouveaux types de données `datetimefractional` ou `timestampfractional`.

Guide de migration vers la version 4.1

CakePHP 4.1 est une mise à jour de l'API compatible à partir de la version 4.0. Cette page présente les dépréciations et fonctionnalités ajoutées dans la version 4.1.

Mettre à jour vers la version 4.1.0

Vous pouvez utiliser composer pour mettre à jour vers CakePHP 4.1.0 :

```
php composer.phar require --update-with-dependencies "cakephp/cakephp:4.1.x"
```

34. https://github.com/dereuromark/cakephp-shim

Dépréciations

4.1 introduit quelques dépréciations. Toutes ces fonctionnalités continueront d'exister dans les versions 4.x mais seront supprimées dans la version 5.0. Vous pouvez utiliser l'' *outil de mise à niveau* pour automatiser la mise à jour des fonctionnalités obsolètes :

```
bin/cake upgrade rector --rules cakephp41 <path/to/app/src>
```

> **Note**
>
> Cela ne met à jour que les changements de CakePHP 4.1. Assurez-vous d'appliquer d'abord les modifications de CakePHP 4.0.

Controller

- L'option `sortWhitelist` du composant `PaginatorComponent` a été dépréciée. Utilisez `sortableFields` à sa place.
- L'option `whitelist` du composant `PaginatorComponent` a été dépréciée. Utilisez `allowedParameters` à sa place.

Database

- `TableSchema::getPrimary()` a été dépréciée. Utilisez `getPrimaryKey()` à sa place.
- `Cake\Database\Schema\BaseSchema` a été renommée en `Cake\Database\Schema\SchemaDialect`.
- `Cake\Database\Schema\MysqlSchema` a été renommée en `Cake\Database\Schema\MysqlSchemaDialect` et marquée comme interne.
- `Cake\Database\Schema\SqliteSchema` a été renommée en `Cake\Database\Schema\SqliteSchemaDialect` et marquée comme interne.
- `Cake\Database\Schema\SqlserverSchema` a été renommée en `Cake\Database\Schema\SqlserverSchemaDialect` et marquée comme interne.
- `Cake\Database\Schema\PostgresSchema` a été renommé en `Cake\Database\Schema\PostgresSchemaDialect` et marquée comme interne.
- `DateTimeType::setTimezone()` a été dépréciée. Utilisez `setDatabaseTimezone()` à sa place.
- La signature la méthode magique pour `FunctionBuilder::cast([...])` est dépréciée. Utilisez `FunctionBuilder::cast('field', 'type')` à sa place.
- `Cake\Database\Expression\Comparison` a été renommé en `Cake\Database\Expression\ComparisonExpression`.

Datasource

- L'option `sortWhitelist` de la classe `Paginator` a été dépréciée. Utilisez `sortableFields` à sa place.
- L'option `whitelist` de la classe `Paginator` a été dépréciée. Utilisez `allowedParameters` à sa place.

Form

- `Form::schema()` a été déprécié. Utilisez `Form::getSchema()` or `Form::setSchema()` à sa place.

Http

- `CsrfProtectionMiddleware::whitelistCallback()` a été dépréciée. Utilisez `skipCheckCallback()` à sa place.
- `ServerRequest::input()` est dépréciée. Utilisez `(string)$request->getBody()` pour obtenir les données d'entrées brutes venant de PHP sous forme de chaîne de caractères ; utilisez `BodyParserMiddleware` pour décoder le corps de la requête et ainsi le rendre disponible sous la forme d'un array/object au travers de la méthode `$request->getParsedBody()`

— L'option `httpOnly` pour le middleware `CsrfProtectionMiddleware` se nomme à présent `httponly` afin d'améliorer la cohérence avec la création des cookie que l'on trouve par ailleurs dans le framework.

ORM

— `QueryExpression::or_()` et `QueryExpression::and_()` ont été dépréciées. Utilisez `or()` and `and()` à leur place.

Routing

— `Cake\Routing\Exception\RedirectException` est dépréciée. Utilisez `Cake\Http\Exception\RedirectException` à sa place.

View

— `Form/ContextInterface::primaryKey()` a été dépréciée. Utilisez `getPrimaryKey()` à sa place.

Changement pour les Behavior

Bien que les modifications suivantes ne modifient pas la signature des méthodes, elles changent la sémantique ou le comportement de certaines méthodes.

Database

— MySQL : Les largeurs d'affichage des entiers sont désormais ignorées sauf pour `TINYINT(1)` qui correspond toujours au type booléen. Les largeurs d'affichage sont obsolètes dans MySQL 8.

Http

— La normalisation des fichiers téléchargés a été déplacée de `ServerRequest` vers `ServerRequestFactory`. Cela pourrait avoir un impact sur vos tests si vous créez objets de requête qui utilisent des tableaux de téléchargement de fichiers imbriqués. Les tests utilisant `IntegrationTestCaseTrait` n'ont pas à être modifiés.

ORM

— `Cake\ORM\TableRegistry` a été dépréciée. Utilisez `Cake\ORM\Locator\LocatorAwareTrait::getTableLocator()` ou `Cake\Datasource\FactoryLocator::get('Table')` afin d'obtenir une instance du "table locator". Les classes comme `Controller`, `Command`, `TestCase` utilisent déjà `Cake\ORM\Locator\LocatorAwareTrait` ainsi dans ces classes vous pouvez simplement utiliser `$this->getTableLocator()->get('ModelName')`.
— Les associations BelongsToMany respectent désormais le "bindingKey" fourni dans la table de jointure de l'association BelongsTo. Auparavant, la clé primaire de la table cible était toujours utilisée à la place.
— Les noms d'association sont désormais correctement sensibles à la casse et doivent correspondre lorsqu'ils sont référencés dans des fonctions telles que `Query::contain()` et `Table::hasMany()`.
— `Cake\ORM\AssociationCollection` ne transfome désormais plus les noms des associations en minuscule quand il génère les clés pour les tableaux d'objets (map) qu'il maintien en interne.

TestSuite

— `TestCase::setAppNamespace()` retourne maintenant l'espace de noms précédent de l'application afin de simplifier sa sauvegarde et sa restauration.
— GroupsFixture a été renommé en SectionsFixture à cause d'un changement des mots clés réservés de MySQL.

View

- Les sources de données par défaut du helper `FormHelper``sont à présent ``data, context` au lieu de `context`. Si vous utilisez `setValueSources()` pour changer les valeurs des sources, vous pourriez avoir besoin de mettre votre code à jour.
- Les classes de contexte `FormHelper` fournies par CakePHP ne prennent désormais plus un objet `$request` dans leur constructeur.

Nouvelles fonctions

Datasource

- `EntityInterface::getAccessible()` a été ajoutée.

Console

- Lorsque la variable d'environnement `NO_COLOR` est définie toutes les sorties n'inclueront pas les codes d'échappement ANSII correspondant aux couleurs. Voyez no-color.org [35] pour plus d'informations.
- Les commandes ont désormais la même possibilité de désactiver le mode interactif que les shells possédaient en utilisant `$io->setInteractivate(false)`. Ici les invites seront évitées le cas échéant et les valeur par défaut seront utilisées. L'utilisation de `--quiet/-q` permet également de réaliser cela pour toutes les commandes existantes.

Database

- MySQL 8 est pris en charge et testé.
- `AggregateExpression` a été ajouté pour représenter les fonctions SQL d'agrégation. `FunctionsBuilder::aggregate()` peut être utilisé pour encapsuler de nouvelles fonctions SQL agrégées.
- La prise en charge des fonctions "Window" a été ajoutée pour n'importe quelle expression agrégée. `AggregateExpression` enveloppe l'expression de fenêtrage (window expression) pour le chaînage des appels.
- Les fonctions d'agrégation prennent désormais en charge les clauses `FILTER (WHERE ...)`.
- Postgres et SQLServer prennent désormais en charge les conditions `HAVING` sur les fonctions d'aggrégation avec alias.
- `FunctionsBuilder::cast()` a été ajoutée.
- Le support des Common Table Expression (CTE) a été ajouté. Les CTE peuvent être attachées en utilisant `Query::with()`.
- `Query::orderAsc()` et `Query::orderDesc()` acceptent désormais des closures comme champs vous permettant ainsi de construire des expressions de tri (order) complexes utilisant l'objet `QueryExpression`.

Error

- `debug()` et `Debugger::printVar()` émettent maintenant du HTML dans les contextes Web, et des sories formattées selon le style ANSI dans le contexte de ligne de commande CLI. L'affichage de structures cycliques et des objets répétés est plus simple. Les objects cycliques ne sont affichés en entier qu'une fois et utilisent des id de référence pour pointer vers la valeur complète de l'objet.
- `Debugger::addEditor()` et `Debugger::setEditor()` ont été ajoutées. Ces méthodes vous permettent respectivement d'ajouter des formats supplémentaires à l'éditeur et de définir votre éditeur préféré.
- La valeur de configuration `Debugger.editor` a été ajoutée. Cette valeur est utilisée pour définir le format des liens préféré pour l'éditeur.
- `ErrorHandlerMiddleware` supporte à présent `Http\Exception\RedirectException` et convertit ces exceptions en redirection HTTP.
- `BaseErrorHandler` utilise maintenant le logger configuré pour les erreurs afin d'enregistrer les avertissement de PHP ainsi que les erreurs.

35. https://no-color.org/

— `ErrorLoggerInterface` a été ajouté pour formaliser l'interface requise pour les loggers d'erreurs personnalisés.

Form

— `Form::set()` a été ajoutée. Cette méthode vous permet d'ajouter des données supplémentaires au objets de formulaires de la même façon que `View::set()` ou `Controller::set()`.

Http

— `BaseApplication::addOptionalPlugin()` a été ajoutée. Cette méthode gère chargement des plugins et gestion des erreurs pour les plugins qui peuvent ne pas exister car ce sont des dépendances de développement.
— `Cake\Http\Exception\RedirectException` a été ajoutée. Cette exception remplace `RedirectException` dans le package de routage et peut être déclenchée n'importe où dans votre application pour signaler au middleware de gestion des erreurs de créer une réponse de redirection.
— `CsrfProtectionMiddleware` peut désormais créer des cookies avec l'attribut `samesite`.
— Le second paramètre de `Session::read()` permet maintenant de définir des valeurs par défaut.
— `Session::readOrFail()` a été ajouté comme wrapper permettant le déclenchement d'exceptions pour les opérations `read()` pour lesquelles la clé manque.

I18n

— La méthode `setJsonEncodeFormat` pour les classes `Time`, `FrozenTime`, `Date` et `FrozenDate` accepte désormais une fonction de rappel (callable) qui peut être utilisée pour retourner une chaîne de caractères personnalisée.
— Le parsing indulgent (Lenient) pour `parseDateTime()` et `parseDate()` peut être désactivé en utilisant `disableLenientParsing()`. Par défaut il est activé - idem pour IntlDateFormatter.

Log

— Les messages de Log peuvent désormais contenir des placeholders du type `{foo}`. Ces placeholders seront remplacés par les valeurs du paramètre `$context` le cas échant.

ORM

— L'ORM déclenche maintenant un événement `Model.afterMarshal` après que chaque entité ait été marshalée à partir des données de la requête.
— Vous pouvez utiliser l'option `locale` du finder option pour modifier la locale d'une recherche pour une table qui a le comportement `TranslateBehavior`.
— `Query::clearResult()` a été ajoutée. Cette méthode vous permet de supprimer le résultat d'une requête afin que vous puissiez la réexécuter.
— `Table::delete()` abandonnera désormais une opération de suppression et retournera false si une association dépendante ne parvient pas à être supprimée pendant les opérations de cascadeCallback.
— `Table::saveMany()` déclenchera maintenant l'événement `Model.afterSaveCommit` sur les entités qui sont enregistrées.

Routing

— Une fonction pratique `urlArray()` a été introduite pour générer rapidement des tableaux d'URL à partir d'une chaîne de chemin de route..

TestSuite

- `FixtureManager::unload()` ne tronque plus les tables à la *fin* d'un test tandis que les fixtures sont déchargés. Les tables seront toujours tronquées pendant le chargement des fixturage. Vous devriez voir une suite de tests plus rapide car moins d'opérations de troncature seront réalisées.
- Les assertions concernant le corps des email incluent désormais le contenu de l'email rendant les tests plus faciles à déboguer.
- `TestCase::addFixture()` a été ajouté pour permettre la configuration en chaîne des fixtures, ceci est également auto-completable dans les IDEs.

View

- La méthode `TextHelper::slug()` a été ajoutée. Elle délègue le travail à `Cake\Utility\Text::slug()`.
- La nouvelle méthode `ViewBuilder::addHelper()` permet de créer des helpers en chaîne.
- Les nouvelles méthodes `HtmlHelper::linkFromPath()` et `UrlHelper::urlFromPath()` permettent de créer des liens et des URLs à partir des chemins de routes et offrent le support de l'IDE dans les fichiers de vue.

Utility

- `Hash::combine()` accepte maintenant `null` pour le paramètre `$keyPath`. Fournir la valeur null produira un tableau de sortie indexé numériquement.

Guide de migration vers la version 4.2

CakePHP 4.2 est une mise à jour de l'API compatible à partir de la version 4.0. Cette page présente les dépréciations et fonctionnalités ajoutées dans la version 4.2.

Mettre à jour vers la version 4.2.0

Vous pouvez utiliser composer pour mettre à jour vers CakePHP 4.2.0 :

```
php composer.phar require --update-with-dependencies "cakephp/cakephp:4.2.x"
```

Dépréciations

4.2 introduit quelques dépréciations. Toutes ces fonctionnalités continueront d'exister dans les versions 4.x mais seront supprimées dans la version 5.0. Vous pouvez utiliser l'*outil de mise à niveau* pour automatiser la mise à jour des fonctionnalités obsolètes :

```
bin/cake upgrade rector --rules cakephp42 <path/to/app/src>
```

> **Note**
>
> Cela ne met à jour que les changements de CakePHP 4.2. Assurez-vous d'appliquer d'abord les modifications de CakePHP 4.1.

Une nouvelle option de configuration a été ajoutée pour désactiver les dépéréciations chemin par chemin. Cf. *Avertissements de dépréciation* pour plus d'informations.

Core

- L'`Exception::responseHeader()` est maintenant dépréciée. Les utilisateurs doivent utiliser `HttpException::setHeaders()` pour définir les en-têtes de la réponse HTTP. Les exceptions d'applications et de plugins qui définissent des en-têtes de réponse devraient être mises à jour pour hériter de `HttpException`.
- `Cake\Core\Exception\Exception` a été renommée en `Cake\Core\Exception\CakeException`.

Database

- `Cake\Database\Exception` a été renommée en `Cake\Database\Exception\DatabaseException`.

ORM

- `TableLocator::allowFallbackClass()` a été ajoutée. Cette méthode vous permet de désactiver les classes de table fallback générées automatiquement. La désactivation est actuellement une option, mais deviendra à l'avenir le comportement par défaut.
- `ORM\Behavior::getTable()` a été dépréciée. Utilisez `table()` à la place. Ce changement marque une différence des noms de méthodes par rapport à `ORM\Table`, car les valeurs de retour de ces méthodes sont différentes.

Changement pour les Behavior

Bien que les changements qui suivent ne modifient pas la signature des méthodes, ils changent la signification ou le comportement des méthodes.

Collection

- `Collection::groupBy()` et `Collection::indexBy()` lèvent maintenant une exception quand le chemin n'existe pas ou quand le chemin contient une valeur null. Les utilisateurs qui ont besoin de null devraient utiliser un callback pour renvoyer plutôt une valeur par défaut.

Controller

- `Controller::$components` a été marquée protected. Elle était auparavant documentée comme protected. Cela ne devrait pas impacter la plupart des codes d'application puisque les implémentations peuvent augmenter la visibilité à public.

Component

- `FlashComponent::set()` définit maintenant l'option `element` à `error` par défaut quand elle est utilisée avec une instance `Exception`.

Database

- Le `TimeType` sérialisera désormais correctement les valeurs dans le format `H:i`. Auparavant ces valeurs étaient castées en `null` après validation.
- Le pilote `Sqlserver` réessayera de se connecter après une erreur « Azure Sql Database paused ».

Error

- `ExceptionRenderer` utilise maintenant le code de l'exception comme statut HTTP uniquement pour `HttpException`. Les autres exceptions qui sont censées renvoyer un code HTTP différent de 500 sont contrôlées par `ExceptionRenderer::$exceptionHttpCodes`.

> **Note**
>
> Si vous avez besoin de restaurer le comportement précédent jusqu'à ce que vos exceptions soient mises à jour, vous pouvez créer un ExceptionRenderer personnalisé et réécrire la function `getHttpCode()`. Cf. *ExceptionRenderer personnalisé* pour plus d'informations.

— `ConsoleErrorHandler` utilise désormais le code de l'exception comme code de sortie uniquement pour `ConsoleException`.

Validation

— `Validation::time()` rejettera désormais une chaîne de texte s'il manque les minutes. Auparavant, elle acceptait des chiffres correspondant uniquement aux heures alors que la documentation de l'API disait que les minutes étaient exigées.

Changements entraînant une rupture

Derrière l'API, certains changements sont nécessaires pour avancer. Ils n'affectent généralement pas les tests.

I18n

— La dépendance envers le paquet Aura.Intl [36] a été supprimée car il n'est plus maintenu. Si votre application/plugin a des traducteurs génériques (*custom translation loaders*) alors elle doit retourner désormais une instance `Cake\I18n\Package` à la place de `Aura\Intl\Package`. Les deux classes ont des API compatibles donc vous n'avez rien besoin de changer d'autre.

Testing

— Les noms de fixtures autour des UUIDs ont été consolidés (`UuidItemsFixture`, `BinaryUuidItemsFixture`). Si vous utilisez l'une d'entre elles, assurez-vous d'avoir mis à jour ces noms. La `UuidportfoliosFixture` n'était pas utilisée dans le cœur et a maintenant été retirée.

Nouvelles fonctionnalités

Nous sommes en train de mettre en place un nouveau process pour nous permettre de lancer de nouvelles fonctionnalités, de recevoir des feedbacks de la communauté et de faire évoluer ces fonctionnalités. Nous appelons ce process *experimental-features*.

Core

— Un support expérimental pour un /development/dependency-injection a été ajouté.

Console

— `ConsoleIo::comment()` a été ajouté. Cette méthode formate le texte en bleu comme des commentaires dans le texte d'aide généré.
— `TableHelper` supporte maintenant un tag de formatage `<text-right>`, qui aligne le contenu de la cellule par rapport au côté droit plutôt que le gauche.

36. https://github.com/auraphp/Aura.Intl

Database

— `SqlServer` crée maintenant par défaut des curseurs en tampon côté client pour les requêtes préparées. Cela a été modifié pour résoudre des problèmes de performance significatifs avec les curseurs SCROLL côté serveur. Les utilisateurs devraient constater des performances boostées pour la plupart des results sets.

> **Avertissement**
>
> Pour les utilisateurs qui ont des requêtes avec de grands résultats, cela peut causer une erreur d'allocation du tampon côté client, si `Query::disableBufferedResults()` n'est pas invoquée. La taille maximum du tampon peut être configurée dans `php.ini` avec `pdo_sqlsrv.client_buffer_max_kb_size`. Cf. https://docs.microsoft.com/en-us/sql/connect/php/cursor-types-pdo-sqlsrv-driver?view=sql-server-ver15#pdo_sqlsrv-and-client-side-cursors pour plus d'informations.

— `Query::isResultsCastingEnabled()` a été ajoutée pour obtenir le mode actuel de cast du résultat en cours.
— `StringExpression` a été ajoutée pour utiliser des string literals avec collation.
— `IdentifierExpression` support maintenant la collation.

Http

— `Cake\Http\Middleware\SessionCsrfProtectionMiddleware` a été ajouté. Plutôt que de stocker les jetons CSRF dans un cookie, ce middleware stocke les jetons en session. Cela limite la portée des jetons CSRF à l'utilisateur et les relie à l'heure de la session, offrant une sécurité accrue par rapport aux jetons basés sur des cookies. Ce middleware est un substitut à `CsrfProtectionMiddleware`.
— Les types `hal+json`, `hal+xml`, et `jsonld` ont été ajoutés à `Response`, les rendant utilisables avec `withType()`.
— `Client::createFromUrl()` a été ajoutée. Cette méthode peut être utilisée pour créer des clients HTTP limités à des domaines incluant une base d'adresse spécifique.
— Une nouvelle classe utilitaire `Cake\Http\FlashMessage` a été ajoutée, dont l'instance est disponible par `ServerRequest::getFlash()`. La classe similaire à `FlashComponent` vous permet de définir des messages flash. Elle peut être particulièrement utile pour définir des messages flash depuis les middlewares.

ORM

— `Table::subquery()` et `Query::subquery()` ont été ajoutées. Ces méthodes vous permettent de créer des objets qui n'ont pas d'aliasing automatique. Cela aide à réduire l'empilement et la complexité de la construction de sous-requêtes et d'expressions de tables communes.
— La règle `IsUnique` accepte maintenant l'option `allowMultipleNulls` qui était disponible dans la version 3.x. Elle est désactivée par défaut, contrairement à ce qui se faisait dans la version 3.x.

TestSuite

— `EmailTrait::assertMailSubjectContains()` et `assertMailSubjectContainsAt()` ont été ajoutées.
— `mockService()` a été ajoutée à `ConsoleIntegrationTestTrait` et `IntegrationTestCaseTrait`. Cette méthode permet de remplacerr les services injectés avec le conteneur d"/development/dependency-injection par des Mocks ou des stubs.

View

— Les classes de contexte incluent maintenant les options de métadonnées `comment`, `null`, and `default` dans les résultats de `attributes()`.
— `ViewBuilder::addHelper()` accepte maintenant un paramètre `$options` pour passer des options dans le constructeur de l'Helper.
— L'option `assetUrlClassName` a été ajoutée à `UrlHelper`. Cette option vous permet de remplacer l'asset URL resolver par défaut par un autre qui soit spécifique à l'application. Cela peut être utile si vous avez besoin de personnaliser les paramètres de l'asset cache busting.

Guide de migration vers la version 4.3

CakePHP 4.3 est une mise à jour de l'API compatible à partir de la version 4.0. Cette page présente les dépréciations et fonctionnalités ajoutées dans la version 4.3.

Mettre à jour vers la version 4.3.0

Vous pouvez utiliser composer pour mettre à jour vers CakePHP 4.3.0 :

```
php composer.phar require --update-with-dependencies "cakephp/cakephp:^4.3"
```

Dépréciations

4.3 introduit quelques dépréciations. Toutes ces fonctionnalités continueront d'exister dans les versions 4.x mais seront supprimées dans la version 5.0. Vous pouvez utiliser l'*outil de mise à niveau* pour automatiser la mise à jour des fonctionnalités obsolètes :

```
bin/cake upgrade rector --rules cakephp43 <path/to/app/src>
```

> **Note**
>
> Cela ne met à jour que les changements de CakePHP 4.3. Assurez-vous d'appliquer d'abord les modifications de CakePHP 4.2.

Une nouvelle option de configuration a été ajoutée pour désactiver les dépréciations chemin par chemin. Cf. *Avertissements de dépréciation* pour plus d'informations.

Connection

— `Connection::supportsDynamicConstraints()` a été dépréciée car les fixtures ne tentent plus de supprimer ou créer des contraintes dynamiquement.

Controller

— Le callback de l'événement `Controller.shutdown` des controllers a été renommé de `shutdown` à `afterFilter` pour correspondre à celui du controller. Cela rend les callbacks plus cohérents.

Base De Données

— L'utilisation de classes de date et heure mutables avec `DateTimeType` et les autres classes de types relatifs aux heures est déprécié. De ce fait, les méthodes `DateTimeType::useMutable()`, `DateTimeType::useImmutable()` et les méthodes similaires dans d'autres classes de types sont dépréciées.
— `DriverInterface::supportsQuoting()` et `DriverInterface::supportSavepoints()` sont maintenant dépréciées au profit de `DriverInterface::supports()` qui accepte des constantes de feature définies dans `DriverInterface`.
— `DriverInterface::supportsDynamicConstraints()` a été dépréciée dès lors que les fixtures ne tentent plus de supprimer ou créer des contraintes dynamiquement.

I18n

— Les classes de date et heure `Time` et `Date` sont dépréciées. À la place, utilisez leurs alternatives immutables `FrozenTime` et `FrozenDate`.

Log

- Dans `FileLog` l'option de configuration `dateFormat` a été déplacée vers `DefaultFormatter`.
- Dans `ConsoleLog` l'option de configuration `dateFormat` a été déplacée vers `DefaultFormatter`.
- Dans `SyslogLog` l'option de configuration `format` a été déplacée vers `LegacySyslogFormatter`. Par défaut, c'est maintenant `DefaultFormatter` qui est utilisé.

Middleware

- Les middlewares « double pass », c'est-à-dire les classes avec une méthode `__invoke($request, $response, $next)`, sont dépréciés. À la place, utilisez `Closure` avec la signature `function($request, $handler)` o des classes qui implémentent `Psr\Http\Server\MiddlewareInterface`.

Network

- `Socket::$connected` est déprécié. Utilisez `isConnected()` à la place.
- `Socket::$description` est déprécié.
- `Socket::$encrypted` est déprécié. Utilisez `isEncrypted()` à la place.
- `Socket::$lastError` est déprécié. Utilisez `lastError()` à la place.

ORM

- `ModelAwareTrait::loadModel()` est dépréciée. Utilisez la nouvelle méthode `LocatorAwareTrait::fetchTable()` à la place. Par exemple, dans les controllers vous pouvez faire `$this->fetchTable()` pour obtenir l'instance de la table par défaut, ou utiliser `$this->fetchTable('Foos')` pour une table autre que celle par défaut. Vous pouvez définir la propriété `LocatorAwareTrait::$defaultTable` pour spécifier l'alias de la table par défaut pour `fetchTable()`.
- L'usage de requêtes pour intercepter toutes les méthodes de `ResultSetInterface` (y compris `` `CollectionInterface` ``), forcer la récupération des résultats et appeler la méthode sous-jacente sur ces résultats est maintenant déprécié. Un exemple de cet usage est `$query->combine('id', 'title');`. Ceci doit être remplacé par `$query->all()->combine('id', 'title');`.
- Passer un object validator à `Table::save()` via l'option `validate` est déprécié. Définissez le validator dans la classe de table ou utilisez `setValidator()` à la place.
- `Association::setName()` est dépréciée. Les noms d'associations doivent être définis en même temps que l'association.
- `QueryExpression::addCase()` est dépréciée. Utilisez `case()` à la place. Les syntaxes `['value' => 'literal']` et `['column' => 'identifier']` ne sont pas supportées dans le nouveau case builder. L'insertion de SQL brut ou d'identifiants nécessite d'utiliser des expressions explicitement. Vous pouvez définir la propriété `LocatorAwareTrait::$defaultTable` pour spécifier l'alias de la table par défaut.

Routing

- Les placeholders de routes préfixés par des doubles points tels que `:controller` sont dépréciés. Remplacez-les par des placeholders entre accolades tels que `{controller}`.

TestSuite

- `TestFixture::$fields` et `TestFixture::$import` sont dépréciés. Il est conseillé de convertir votre application vers le *nouveau système de fixture*.
- `TestCase::$dropTables` est déprécié. La suppression de tables pendant l'exécution d'un test est incompatible avec les nouvelles fixtures basées sur le dump d'une migration/schéma. La fonctionnalité sera supprimée dans 5.0.

View

- Les options non associatives des méthodes de FormHelper (par exemple `['disabled']`) sont maintenant dépréciées.
- Le second argument `$merge` de `ViewBuilder::setHelpers()` a été déprécié au profit de la méthode dédiée `ViewBuilder::addHelpers()` qui sépare proprement l'ajout et le remplacement de helpers.

Changements de comportements

Bien que les changements qui suivent ne changent la signature d'aucune méthode, ils en changent la sémantique ou le comportement.

Collection

- Le paramètre `$preserveKeys` a été renommé en `$keepKeys` avec la même implémentation.

Command

- `cake i18n extract` n'a plus d'option `--relative-paths`. Cette option est maintenant activée par défaut.

Core

- `Configure::load()` soulèvera désormais une exception en cas d'utilisation d'un moteur de configuration invalide.

Database

- `ComparisonExpression `` n'entoure plus le SQL de ``IdentifierExpression` entre des parenthèses. Cela affecte `Query::where()` et tous les autres endroits où une `ComparisonExpression` est générée.
- L'implémentation SQLite de `listTables()` renvoie maintenant les tables **et** les vues. Ce changement aligne SQLite avec les autres dialectes de bases de données.

Datasource

- Les noms des paramètres `$alias` et `$source` de `ConnectionManager::alias()` ont été modifiés pour correspondre à ce qu'ils sont. Cela affecte uniquement la documentation et les paramètres nommés.

Http

- `Http\Client` utilise maintenant `ini_get('user_agent')` avec "CakePHP" en tant que valeur de repli pour son user-agent.

ORM

- `Entity::isEmpty()` et `Entity::hasValue()` ont été alignées pour traiter "0" comme une valeur non-empty. Cela aligne le behavior avec la documentation et l'intention originelle.
- Les erreurs de validation d'entity de `TranslateBehavior` sont maintenant définies dans le chemin `_translations.{lang}` au lieu de `{lang}`. Cela normalise le chemin des erruers d'entities pour les données de la requête. Si vous avez des formulaires qui modifient plusieurs tranductions à la fois, vous aurez vraisemblablement besoin de mettre à jour la façon dont sont rendues les erreurs de validation.
- Les types spécifiés dans des expressions de fonctions ont maintenant la préséance sur les ensembles de types par défaut pour les colonnes, quand des colonnes sont sélectionnées. Par exemple, pour utiliser `$query->select(['id' => $query->func()->min('id')])` la valeur pour *id* dans l'entity récupérée sera un *float* au lieu d'un *integer*.

Routing

- `Router::connect()`, `Router::prefix()`, `Router::plugin()` et `Router::scope()` sont dépréciées. Utilisez les méthodes non statiques correspondantes de `RouteBuilder` à la place.
- `RouteBuilder::resources()` génère maintenant des routes qui utilisent des placeholders entre accolades.

TestSuite

- `TestCase::deprecated()` vérifie (*asserts*) maintenant qu'au moins un avertissement de dépréciation ait été déclenché par le callback.

Validation

- `Validator::setProvider()` lève maintenant une exception quand un nom de provider fourni n'est ni un objet ni une chaîne de caractères. Auparavant cela n'était pas une erreur, mais le provider ne fonctionnait pas.

View

- Le paramètre `$vars` de `ViewBuilder::build()` est déprécié. Utilisez `setVar()` à la place.
- `HtmlHelper::script()` et `HtmlHelper::css()` échappent désormais les URLs absolues qui incluent un scheme.

Changements entraînant une rupture

Derrière l'API, certains changements sont nécessaires pour avancer. Ils n'affectent généralement pas les tests.

Log

- Les configurations de `BaseLog::_getFormattedDate()` et `dateFormat` ont été supprimées puisque la logique de formatage du message a été déplacée vers les formatters de logs.

View

- `TimeHelper::fromString()` renvoie maintenant une instance de `FrozenTime` au lieu de `Time`.

Nouvelles fonctionnalités

Controller

- `Controller::middleware()` a été ajoutée. Elle vous permet de définir un middleware pour un seul contrôleur. Reportez-vous à *Middleware de Controller* pour plus d'informations.
- Les controllers supportent maintenant des paramètres d'actions avec des types déclarés `float`, `int`, `bool` ou `array`. Les booléens passés doivent être soit `0` soit `1`.

Core

- `deprecationWarning()` n'émet plus de notices en doublon. Au lieu de cela, seule la permière instance de dépréciation sera affichée. Cela améliore la lisibilité de la sortie de test, et le bruit visuel dans un contexte HTML. Vous pouvez restaurer la sortie de notices en doublon en définissant `Error.allowDuplicateDeprecations` à `true` dans votre `app_local.php`.
- La dépendance de CakePHP envers `league/container` a été mise à niveau à `^4.1.1`. Le conteneur DI étant marqué comme expérimental, cette mise à niveau peut nécessiter que vous mettiez à niveau les définitions de vos service providers.

Database

- Les types de mappage de bases de données peuvent maintenant implémenter `Cake\Database\Type\ColumnSchemaAwareInterface` pour spécifier la génération de colonne SQL et la réflexivité du schéma de colonne. Cela permet au types personnalisés de prendre en charge des colonnes non standard.
- Les queries loguées utilisent maintenant `TRUE` et `FALSE` pour les pilotes postgres, sqlite et mysql. Cela facilite la copie de queries et leur exécution dans un prompt interactif.
- Le `DateTimeType` peut maintenant convertir les données de la requête du fuseau horaire de l'utilisateur vers le fuseau horaire de l'application. Reportez-vous à *Convertir des Données de Requêtes du Fuseau Horaire de l'Utilisateur* pour plus d'informations.
- Ajout de `DriverInterface::supports()` qui consolide toutes les vérifications de feature en une seule fonction. Les pilotes peuvent supporter les nommages personnalisés de feature ou n'importe quelle constante de feature :
  - `FEATURE_CTE`
  - `FEATURE_JSON`
  - `FEATURE_QUOTE`
  - `FEATURE_SAVEPOINT`
  - `FEATURE_WINDOW`
- Ajout de `DriverInterface::inTransaction()` qui reflète le statut renvoyé par `PDO::inTranaction()`.
- Ajout d'un builder fluide pour les instructions `CASE, WHEN, THEN`.
- La méthode `listTablesWithoutViews()` a été ajoutée à `SchemaCollection` et aux dialectes des pilotes. Elle renvoie la liste des tables en excluant les vues. Ceci est principalement utilisé pour tronquer les tables dans les tests.

Form

- `Form::execute()` now accepts an `$options` parameter. This parameter can be used to choose which validator is applied or disable validation.
- `Form::validate()` now accepts a `$validator` parameter which chooses the validation set to be applied.

Http

- Le `CspMiddleware` définit maintenant les attributs de la requête `cspScriptNonce` et `cspStyleNonce` qui rationalise l'adoption de content-security-policy strict.
- `Client::addMockResponse()` et `clearMockResponses()` ont été ajoutées.

Log

- Les moteurs de log utilisent maintenant des formatters pour formater le texte du message avant de l'écrire. Cela peut être configuré avec l'option de configuration `formatter`. Consultez la section *logging-formatters* pour plus de détails.
- `JsonFormatter` a été ajouté et peut être défini comme option `formatter` pour n'importe quel moteur de log.

ORM

- Les queries qui font appel à des associations HasMany et BelongsToMany par `contain()` propagent le statut de cast du résultat. Cela assure que les résultats de toutes les associations sont soit castés avec des objets de types de mappage, soit pas du tout.
- `Table` inclut maintenant `label` dans la liste des champs qui peuvent candidater comme champs par défaut dans `displayField`.
- `Query::whereNotInListOrNull()` et `QueryExpression::notInOrNull()` ont été ajoutés pour les colonnes nullable puisque `null != value` est toujours false et le test `NOT IN` échoue toujours quand la colonne est null.
- `LocatorAwareTrait::fetchTable()` a été ajoutée. Elle vous permet d'utiliser `$this->fetchTable()` pour obtenir une instance de table depuis les classes qui utilisent ce trait, telles que les controllers, les commands, mailers et cells.

TestSuite

- `IntegrationTestTrait::enableCsrfToken()` permet maintenant l'utilisation de noms de clés personnalisés pour les cookies/sessions CSRF.
- `HttpClientTrait` a été ajouté pour faciliter l'écriture de mocks HTTP. Cf. *Tests* pour plus d'information.
- Un nouveau système de fixture a été introduit. Ce système de fixture sépare le schéma et les données, ce qui vous permet de réutiliser vos migrations existantes pour définir un schéma de test. Le guide *Mise à Niveau des Fixtures* explique comment mettre à niveau.

View

- `HtmlHelper::script()` et `HtmlHelper::css()` ajoutent maintenant l'attribut `nonce` pour générer des balises quand les attributs de requête `cspScriptNonce` et `cspStyleNonce` sont présents.
- `FormHelper::control()` complète maintenant les attributs `aria-invalid`, `aria-required` et `aria-describedby` à partir des métadonnées depuis le validator. L'attribut `aria-label` sera défini si vous désactivez l'élement automatique label et fournissez un placeholder.
- `ViewBuilder::addHelpers()` a été ajoutée pour séparer proprement les opérations d'ajout et de redéfinition de helpers.

Guide de migration vers la version 4.4

CakePHP 4.4 est une mise à jour de l'API compatible à partir de la version 4.0. Cette page présente les dépréciations et fonctionnalités ajoutées dans la version 4.4.

Mettre à jour vers la version 4.4.0

Vous pouvez utiliser composer pour mettre à jour vers CakePHP 4.4.0 :

```
php composer.phar require --update-with-dependencies "cakephp/cakephp:^4.4"
```

> **Note**
>
> CakePHP 4.4 nécessite PHP 7.4 ou supérieur.

Dépréciations

4.4 introduit quelques dépréciations. Toutes ces fonctionnalités continueront d'exister dans les versions 4.x mais seront supprimées dans la version 5.0.

Vous pouvez utiliser l'*outil de mise à niveau* pour automatiser la mise à jour des fonctionnalités obsolètes :

```
bin/cake upgrade rector --rules cakephp44 <path/to/app/src>
```

> **Note**
>
> Cela ne met à jour que les changements de CakePHP 4.4. Assurez-vous d'appliquer d'abord les modifications de CakePHP 4.3.

Une nouvelle option de configuration a été ajoutée pour désactiver les dépréciations chemin par chemin. Cf. *Avertissements de dépréciation* pour plus d'informations.

Controller

- L'option `paginator` pour `Controller::paginate()` est dépréciée. Utilisez l'option `className` à la place.
- L'option `paginator` pour `PaginatorComponent` est dépréciée. Utilisez l'option `className` à la place.

Datasource

- `FactoryLocator::add()` n'accepte plus de fonction de génération de closures. À la place, vous devez passer une instance de `LocatorInterface`.
- `Cake\Datasource\Paging\Paginator` a été renommé en `Cake\Datasource\Paging\NumericPaginator`.

ErrorHandler & ConsoleErrorHandler

Les classes `ErrorHandler` et `ConsoleErrorHandler` sont maintenant dépréciées. Elles ont été remplacées par les nouvelles classes `ExceptionTrap` et `ErrorTrap`. Les classes *trap* fournissent des outils plus extensibles et cohérents pour gérer les erreurs et exceptions. Pour mettre à niveau vers le nouveau système, vous pouvez remplacer l'utilisation de `ErrorHandler` et `ConsoleErrorHandler` (notamment dans votre `config/bootstrap.php`) par :

```
use Cake\Error\ErrorTrap;
use Cake\Error\ExceptionTrap;

(new ErrorTrap(Configure::read('Error')))->register();
(new ExceptionTrap(Configure::read('Error')))->register();
```

Si vous avez défini la valeur de configuration de `Error.errorLogger`, vous devrez le modifier en `Error.logger`.

Consultez la page *Gestion des Erreurs & Exceptions* pour une documentation plus détaillée (Ndt : sa traduction française n'est pas à jour. Appel aux bonnes volontés !). De plus, les méthodes suivantes liées au système déprécié de gestion des erreurs sont elles-mêmes dépréciées :

- `Debugger::outputError()`
- `Debugger::getOutputFormat()`
- `Debugger::setOutputFormat()`
- `Debugger::addFormat()`
- `Debugger::addRenderer()`
- `ErrorLoggerInterface::log()`. Implémentez `logException()` à la place.
- `ErrorLoggerInterface::logMessage()`. Implémentez `logError()` à la place.

RequestHandlerComponent

Le RequestHandlerComponent est déprécié « soft ». Comme pour `AuthComponent`, l'usage de `RequestHandler` ne déclenchera pas d'avertissements à l'exécution mais il **sera** supprimé dans 5.0.

- Remplacez `accepts()` par `$this->request->accepts()`.
- Remplacez `requestedWith()` par un détecteur personnalisé de requête (par exemple, `$this->request->is('json')`).
- Remplacez `prefers()` par `ContentTypeNegotiation`. Consultez *Négociation de Contenu*.
- Remplacez `renderAs()` par des fonctionnalités de négociation de contenu dans `Controller`.
- Remplacez l'option `checkHttpCache` par *Checking HTTP Cache*.
- Utilisez les *Négociation de Contenu* au lieu de définir des mappings de classes de vues dans `RequestHandlerComponent`.

PaginationComponent

Le `PaginationComponent` est déprécié et sera supprimé dans 5.0. Utilisez la propriété `Controller::$paginate` ou le paramètre `$settings` de la méthode `Controller::paginate()` pour spécifier les réglages de pagination nécessaires.

ORM

- `SaveOptionsBuilder` a été déprécié. Utilisez un tableau pour les options à la place.

Plugins

- Les noms de classes de plugin correspondent désormais au nom du plugin avec le suffixe « Plugin ». Par exemple, la classe de plugin pour « ADmad/I18n » serait `ADmad\I18n\I18nPlugin` au lieu de `ADmad\I18n\Plugin`, comme c'était le cas pour CakePHP 4.3 et antérieurs. L'ancien style de noms sera toujours supporté pour des raisons de compatibilité descendante.

Routing

- Les fichiers de route mis en cache ont été dépréciés. Cela soulevait de nombreux cas de figure impossibles à résoudre avec des routes en cache. Comme la fonctionnalité des routes en cache n'est pas fonctionnelles dans de nombeux cas d'utilisation, elle sera supprimée dans 5.x

Suite de Test

- `ConsoleIntegrationTestTrait` a été déplacé dans le package de la console, au même endroit que les autres dépendances, pour permettre de tester les applications en console sans avoir besoin de tout le package cakephp/cakephp.
  - `Cake\TestSuite\ConsoleIntegrationTestTrait` a été déplacé vers `Cake\Console\TestSuite\ConsoleIntegrationTestTrait`
  - `Cake\TestSuite\Constraint\Console\*` a été déplacé vers `Cake\Console\TestSuite\Constraint\*`
  - `Cake\TestSuite\Stub\ConsoleInput` a été déplacé vers `Cake\Console\TestSuite\StubConsoleInput`
  - `Cake\TestSuite\Stub\ConsoleOutput` a été déplacé vers `Cake\Console\TestSuite\StubConsoleOutput`
  - `Cake\TestSuite\Stub\MissingConsoleInputException` a été déplacé `Cake\Console\TestSuite\MissingConsoleInputException`
- `ContainerStubTrait` a été déplacé vers le package du cœur pour permettre le test des applications en console sans avoir besoin de tout le package cakephp/cakephp.
  - `Cake\TestSuite\ContainerStubTrait` a été déplacé vers `Cake\Core\TestSuite\ContainerStubTrait`
- `HttpClientTrait` a été déplacé vers le package http pour permettre de tester les applications http sans avoir besoin de tout le package cakephp/cakephp.
  - `Cake\TestSuite\HttpClientTrait` a été déplacé vers `Cake\Http\TestSuite\HttpClientTrait`

Changements de comportement

Bien que les changements suivants ne changent pas la signature des méthodes, ils changent la sémantique ou le comportement de certaines d'entre elles.

ORM

— `Table::saveMany()` now triggers the `Model.afterSaveCommit` event with entities that are still "dirty" and contain the original field values. This aligns the event payload for `Model.afterSaveCommit` with `Table::save()`.

Routing

— `Router::parseRequest()` soulève maintenant une `BadRequestException` au lieu d'une `InvalidArgumentException` lorsque le client utilise une méthode HTTP invalide.

Nouvelles Fonctionnalités

Cache

— `RedisEngine` supporte désormais les méthodes `deleteAsync()` et `clearBlocking()`. Ces méthodes utilisent l'opération `UNLINK` dans redis pour marquer les données en vue d'une suppression ultérieure par Redis.

Command

— `bin/cake routes` met maintenant en valeurs les collisions dans les templates de routes.
— `Command::getDescription()` vous permet de définir une description personnalisée.Cf. *Définir la Description de la Commande*

Controller

— `Controller::viewClasses()` a été ajoutée. Cette méthode devraient être implémentée par les contrôleurs qui ont besoin d'effectuer des négociations sur le content-type. Les classes de vue auront besoin d'implémenter la méthode statique `contentType()` pour participer à la négociation du content-type.

Database

— Le pilote `SQLite` supporte à présent les base de données partagées en mémoire sous PHP8.1+.
— `Query::expr()` a été ajoutée comme alternative à `Query::newExpr()`.
— Le builder `QueryExpression::case()` supporte maintenant l'inférence de types à partir d'expressions passées à `then()` et à `else()` qui implémentent `\Cake\Database\TypedResultInterface`.

Error

— `ErrorTrap` et `ExceptionTrap` ont été ajoutées. Ces classes forment la fondation d'un système de gestion d'erreur mis à jour pour les applications. Pour en savoir plus, rendez-vous sur *Gestion des Erreurs & Exceptions*.

Http

— `Response::checkNotModified()` a été dépréciée. Utilisez `Response::isNotModified()` à la place.
— `BaseApplication::handle()` ajoute désormais systématiquement `$request` dans le conteneur de service.
— `HttpsEnforcerMiddleware` a maintenant une option `hsts` qu vous permet de configurer le header `Strict-Transport-Security`.

Mailer

— `Mailer` accepte désormais une clé de configuration `autoLayout` qui qui désactive le layout automatique dans le `ViewBuilder` si elle est définie à `false`.

ORM

— L'option `cascadeCallbacks` a été ajoutée `TreeBehavior`. Lorsqu'elle est activée, `TreeBehavior` itérera un résultat de `find()` et effacera les enregistrements individuellement. Cela permet d'utiliser les callbacks de l'ORM lors de l'effacement de nœuds.

Routing

— `RoutingMiddleware` définit désormais l'attribut « route » de la requête avec l'instance `Route` qui correspond.

View

— `View::contentType()` a été ajoutée. Les vues peuvent implémenter cette méthode pour participer à une négociation du content-type.
— `View::TYPE_MATCH_ALL` a été ajoutée. Ce content-type spécial vous permet de construire des vues de repli pour les cas où la négociation du content-type ne fournit aucune correspondance.

CHAPITRE 4

Tutoriels et exemples

Dans cette section, vous pourrez découvrir des applications CakePHP typiques afin de voir comment toutes les pièces s'assemblent.

Sinon, vous pouvez vous référer au dépôt de plugins non-officiels de CakePHP CakePackages [37] ainsi que la Boulangerie [38] (Bakery) pour des applications et components existants.

Tutoriel d'un système de gestion de contenu

Ce tutoriel vous accompagnera dans la création d'une application de type CMS. Pour commencer, nous installerons CakePHP, créerons notre base de données et construirons un système simple de gestion d'articles.

Voici les pré-requis :

1. Un serveur de base de données. Nous utiliserons MySQL dans ce tutoriel. Vous avez besoin de connaître assez de SQL pour créer une base de données et exécuter quelques requêtes SQL que nous fournirons dans ce tutoriel. CakePHP se chargera de construire les requêtes nécessaires pour votre application. Puisque nous allons utiliser MySQL, assurez-vous que `pdo_mysql` est bien activé dans PHP.
2. Les connaissances de base en PHP.

Avant de commencer, assurez-vous que votre version de PHP est à jour :

```
php -v
```

Vous devez avoir au minimum PHP 8.1 installé (en CLI). Votre version serveur de PHP doit au moins être aussi 8.1 et, dans l'idéal, devrait également être la même que pour votre version en ligne de commande (CLI).

37. https://plugins.cakephp.org/
38. https://bakery.cakephp.org/

Récupérer CakePHP

La manière la plus simple d'installer CakePHP est d'utiliser Composer. Composer est une manière simple d'installer CakePHP via votre terminal. Premièrement, vous devez télécharger et installer Composer si vous ne l'avez pas déjà fait. Si vous avez cURL installé, exécutez la commande suivante :

```
curl -s https://getcomposer.org/installer | php
```

Ou vous pouvez télécharger `composer.phar` depuis le site de Composer [39].

Ensuite, tapez la commande suivante dans votre terminal pour installer le squelette d'application CakePHP dans le dossier **cms** du dossier courant :

```
php composer.phar create-project --prefer-dist cakephp/app:5.* cms
```

Si vous avez téléchargé et utilisé l'Installer de Composer pour Windows [40], tapez la commande suivante dans votre terminal depuis le dossier d'installation (par exemple C :\wamp\www\dev) :

```
composer self-update && composer create-project --prefer-dist cakephp/app:4.* cms
```

Utiliser Composer a l'avantage d'exécuter automatiquement certaines tâches importantes d'installation, comme définir les bonnes permissions sur les dossiers et créer votre fichier **config/app.php**.

Il existe d'autres moyens d'installer CakePHP. Si vous ne pouvez pas (ou ne voulez pas) utiliser Composer, rendez-vous dans la section *Installation*.

Quelque soit la manière de télécharger et installer CakePHP, une fois que la mise en place est terminée, votre dossier d'installation devrait ressembler à ceci :

```
/cms
  /bin
  /config
  /logs
  /plugins
  /resources
  /src
  /templates
  /tests
  /tmp
  /vendor
  /webroot
  .editorconfig
  .gitignore
  .htaccess
  .travis.yml
  composer.json
  index.php
  phpunit.xml.dist
  README.md
```

C'est le bon moment pour en apprendre d'avantage sur le fonctionnement de la structure des dossiers de CakePHP : rendez-vous dans la section *Structure du dossier de CakePHP* pour en savoir plus.

Si vous vous perdez pendant ce tutoriel, vous pouvez voir le résultat final on GitHub [41].

39. https://getcomposer.org/download/
40. https://getcomposer.org/Composer-Setup.exe
41. https://github.com/cakephp/cms-tutorial

Vérifier l'installation

Il est possible de vérifier que l'installation est terminée en vous rendant sur la page d'accueil. Avant de faire ça, vous allez devoir lancer le serveur de développement :

```
cd /path/to/our/app

bin/cake server
```

> **Note**
>
> Pour Windows, la commande doit être `bin\cake server` (notez le backslash).

Cela démarrera le serveur embarqué de PHP sur le port 8765. Ouvrez **http ://localhost :8765** dans votre navigateur pour voir la page d'accueil. Tous les éléments de la liste devront être validés sauf le point indiquant si CakePHP arrive à se connecter à la base de données. Si d'autres points ne sont pas validés, vous avez peut-être besoin d'installer des extensions PHP supplémentaires ou définir les bonnes permissions sur certains dossiers.

Ensuite, nous allons créer notre *base de données et créer notre premier model*.

Tutoriel CMS - Création de la base de données

Maintenant que CakePHP est installé, il est temps d'installer la base de données pour notre application CMS. Si vous ne l'avez pas encore fait, créez une base de données vide qui servira pour ce tutoriel, avec le nom de votre choix (par exemple `cake_cms`). Si vous utilisez MySQL/MariaDB, vous pouvez exécuter le SQL suivant pour créer le tables nécessaires :

```
USE cake_cms;

CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    email VARCHAR(255) NOT NULL,
    password VARCHAR(255) NOT NULL,
    created DATETIME,
    modified DATETIME
);

CREATE TABLE articles (
    id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT NOT NULL,
    title VARCHAR(255) NOT NULL,
    slug VARCHAR(191) NOT NULL,
    body TEXT,
    published BOOLEAN DEFAULT FALSE,
    created DATETIME,
    modified DATETIME,
    UNIQUE KEY (slug),
    FOREIGN KEY user_key (user_id) REFERENCES users(id)
) CHARSET=utf8mb4;

CREATE TABLE tags (
    id INT AUTO_INCREMENT PRIMARY KEY,
```

(suite sur la page suivante)

(suite de la page précédente)

```
    title VARCHAR(191),
    created DATETIME,
    modified DATETIME,
    UNIQUE KEY (title)
) CHARSET=utf8mb4;

CREATE TABLE articles_tags (
    article_id INT NOT NULL,
    tag_id INT NOT NULL,
    PRIMARY KEY (article_id, tag_id),
    FOREIGN KEY tag_key(tag_id) REFERENCES tags(id),
    FOREIGN KEY article_key(article_id) REFERENCES articles(id)
);

INSERT INTO users (email, password, created, modified)
VALUES
('cakephp@example.com', 'secret', NOW(), NOW());

INSERT INTO articles (user_id, title, slug, body, published, created, modified)
VALUES
(1, 'First Post', 'first-post', 'This is the first post.', 1, NOW(), NOW());
```

Si vous utilisez PostgreSQL, connectez-vous à la base de données `cake_cms` et exécutez le code SQL suivant à la place :

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    email VARCHAR(255) NOT NULL,
    password VARCHAR(255) NOT NULL,
    created TIMESTAMP,
    modified TIMESTAMP
);

CREATE TABLE articles (
    id SERIAL PRIMARY KEY,
    user_id INT NOT NULL,
    title VARCHAR(255) NOT NULL,
    slug VARCHAR(191) NOT NULL,
    body TEXT,
    published BOOLEAN DEFAULT FALSE,
    created TIMESTAMP,
    modified TIMESTAMP,
    UNIQUE (slug),
    FOREIGN KEY (user_id) REFERENCES users(id)
);

CREATE TABLE tags (
    id SERIAL PRIMARY KEY,
    title VARCHAR(191),
    created TIMESTAMP,
    modified TIMESTAMP,
    UNIQUE (title)
```

(suite sur la page suivante)

(suite de la page précédente)

```
);

CREATE TABLE articles_tags (
    article_id INT NOT NULL,
    tag_id INT NOT NULL,
    PRIMARY KEY (article_id, tag_id),
    FOREIGN KEY (tag_id) REFERENCES tags(id),
    FOREIGN KEY (article_id) REFERENCES articles(id)
);

INSERT INTO users (email, password, created, modified)
VALUES
('cakephp@example.com', 'secret', NOW(), NOW());

INSERT INTO articles (user_id, title, slug, body, published, created, modified)
VALUES
(1, 'First Post', 'first-post', 'This is the first post.', TRUE, NOW(), NOW());
```

Vous avez peut-être remarqué que la table `articles_tags` utilise une clé primaire composée. CakePHP supporte les clés primaires composées presque partout, vous permettant d'avoir des shémas plus simples qui ne nécessitent pas de colonnes `id` supplémentaires.

Les noms de tables et de colonnes utilisés ne sont pas arbitraires. En utilisant les *conventions de nommages* de CakePHP, nous allons bénéficier des avantages de CakePHP de manière plus efficace et allons éviter d'avoir trop de configuration à effectuer. Bien que CakePHP soit assez flexible pour supporter presque n'importe quel schéma de base de données, adhérer aux conventions va vous faire gagner du temps.

Configuration de la base de données

Ensuite, disons à CakePHP où est notre base de données et comment nous y connecter. Remplacez les valeurs dans le tableau `Datasources.default` de votre fichier **config/app.php** avec celle de votre installation de base de données. Un exemple de configuration complétée ressemblera à ceci :

```
<?php
return [
    // D'autres configurations au dessus
    'Datasources' => [
        'default' => [
            'className' => Connection::class,
            // Remplacez Mysql par Postgres si vous utilisez PostgreSQL
            'driver' => Mysql::class,
            'persistent' => false,
            'host' => 'localhost',
            'username' => 'cakephp',
            'password' => 'AngelF00dC4k3~',
            'database' => 'cake_cms',
            // Commentez la ligne ci-dessous si vous utilisez PostgreSQL
            'encoding' => 'utf8mb4',
            'timezone' => 'UTC',
            'cacheMetadata' => true,
        ],
    ],
    // D'autres configurations en dessous
```

(suite sur la page suivante)

(suite de la page précédente)

```
];
```

Une fois que vous avez sauvegardé votre fichier **config/app.php**, vous devriez voir que CakePHP est capable de se connecter à la base de données sur la page d'accueil de votre projet.

> **Note**
>
> Si vous avez **config/app_local.php** dans votre dossier d'application, vous devez plutôt configurer votre connexion à la base de données dans ce fichier.

Création du premier Model

Les models font partie du coeur des applications CakePHP. Ils nous permettent de lire et modifier les données, de construire des relations entre nos données, de valider les données et d'appliquer les règles spécifiques à notre application. Les models sont les fondations nécessaires pour construire nos actions de controllers et nos templates.

Les models de CakePHP sont composés d'objets `Table` et `Entity`. Les objets `Table` nous permettent d'accéder aux collections d'entities stockées dans une table spécifique. Ils sont stockés dans le dossier **src/Model/Table**. Le fichier que nous allons créer sera sauvegardé dans **src/Model/Table/ArticlesTable.php**. Le fichier devra contenir ceci :

```php
<?php
// src/Model/Table/ArticlesTable.php
namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{
    public function initialize(array $config): void
    {
        $this->addBehavior('Timestamp');
    }
}
```

Nous y avons attaché le behavior *Timestamp* qui remplira automatiquement les colonnes `created` et `modified` de notre table. En nommant notre objet Table `ArticlesTable`, CakePHP va utiliser les conventions de nommages pour savoir que notre model va utiliser la table `articles`. Toujours en utilisant les conventions, il saura que la colonne `id` est notre clé primaire.

> **Note**
>
> CakePHP créera dynamiquement un objet model s'il n'en trouve pas un qui correspond dans le dossier **src/Model/Table**. Cela veut dire que si vous faites une erreur lors du nommage du fichier (par exemple articlestable.php ou ArticleTable.php), CakePHP ne reconnaitra pas votre configuration et utilisera ce model généré à la place.

Nous allons également créer une classe Entity pour nos Articles. Les Entities représentent un enregistrement spécifique en base et donnent accès aux données d'une ligne de notre base. Notre Entity sera sauvegardée dans **src/Model/Entity/Article.php**. Le fichier devra ressembler à ceci :

```php
<?php
// src/Model/Entity/Article.php
namespace App\Model\Entity;

use Cake\ORM\Entity;

class Article extends Entity
{
    protected array $_accessible = [
        '*' => true,
        'id' => false,
        'slug' => false,
    ];
}
```

Notre entity est assez simple pour l'instant et nous y avons seulement défini la propriété `_accessible` qui permet de contrôler quelles propriétés peuvent être modifiées via *Assignement de Masse*.

Pour l'instant, nous ne pouvons pas faire grande chose avec notre model. Pour intéragir avec notre model, nous allons ensuite créer nos premiers *Controller et Template*.

Tutoriel CMS - Création du Controller Articles

Maintenant que notre model est créé, nous avons besoin d'un controller pour nos articles. Dans CakePHP, les controllers se chargent de gérer les requêtes HTTP et exécutent la logique métier des méthodes des models pour préparer une réponse. Nous placerons le code de ce controller dans un nouveau fichier **ArticlesController.php**, dans le dossier **src/Controller**. La base du controller ressemblera à ceci :

```php
<?php
// src/Controller/ArticlesController.php

namespace App\Controller;

class ArticlesController extends AppController
{
}
```

Ajoutons maintenant une action à notre controller. Les actions sont les méthodes des controllers qui sont connectées aux routes. Par exemple, quand un utilisateur appelle la page **www.example.com/articles/index** (ce qui est la même chose qu'appeler **www.example.com/articles**), CakePHP appelera la méthode `index` de votre controller `ArticlesController`. Cette méthode devra à son tour faire appel à la couche Model et préparer une réponse en faisant le rendu d'un Template via la couche de View. Le code de notre action index sera le suivant :

```php
<?php
// src/Controller/ArticlesController.php

namespace App\Controller;

class ArticlesController extends AppController
{
    public function index()
    {
        $this->loadComponent('Paginator');
```

(suite sur la page suivante)

(suite de la page précédente)

```
        $articles = $this->Paginator->paginate($this->Articles->find());
        $this->set(compact('articles'));
    }
}
```

Maintenant que nous avons une méthode `index()` dans notre `ArticlesController`, les utilisateurs peuvent y accéder via **www.example.com/articles/index**. De la même manière, si nous définissions une méthode `foobar()`, les utilisateurs pourraient y accéder via **www.example.com/articles/foobar**. Vous pourriez être tenté de nommer vos controllers et vos actions afin d'obtenir des URL spécifiques. Cependant, ceci est déconseillé. Vous devriez plutôt suivre les *Conventions de CakePHP* et créer des noms d'actions lisibles ayant un sens pour votre application. Vous pouvez ensuite utiliser le *Routing* pour obtenir les URLs que vous souhaitez et les connecter aux actions que vous avez créées.

Notre action est très simple. Elle récupère un jeu d'articles paginés dans la base de données en utilisant l'objet Model Articles qui est chargé automatiquement via les conventions de nommage. Elle utilise ensuite la méthode `set()` pour passer les articles récupérés au Template (que nous créerons par la suite). CakePHP va automatiquement rendre le Template une fois que notre action de Controller sera entièrement exécutée.

Création du Template de liste des Articles

Maintenant que notre controller récupère les données depuis le model et qu'il prépare le contexte pour la view, créons le template pour notre action index.

Les templates de view de CakePHP sont des morceaux de PHP qui sont insérés dans le layout de votre application. Bien que nous créerons du HTML ici, les Views peuvent générer du JSON, du CSV ou même des fichiers binaires comme des PDFs.

Un layout est le code de présentation qui englobe la view d'une action. Les fichiers de layout contiennent les éléments communs au site comme les headers, les footers et les éléments de navigation. Votre application peut très bien avoir plusieurs layouts et vous pouvez passer de l'un à l'autre. Mais pour le moment, utilisons seulement le layout par défaut.

Les fichiers de template de CakePHP sont stockés dans **templates** et dans un dossier au nom du controller auquel ils sont attachés. Nous devons donc créer un dossier nommé "Articles" dans notre cas. Ajoutez le code suivant dans ce fichier :

```
<!-- Fichier : templates/Articles/index.php -->

<h1>Articles</h1>
<table>
    <tr>
        <th>Titre</th>
        <th>Créé le</th>
    </tr>

    <!-- C'est ici que nous bouclons sur notre objet Query $articles pour afficher les␣
↪informations de chaque article -->

    <?php foreach ($articles as $article): ?>
    <tr>
        <td>
            <?= $this->Html->link($article->title, ['action' => 'view', $article->slug])␣
↪?>
        </td>
        <td>
            <?= $article->created->format(DATE_RFC850) ?>
```

(suite sur la page suivante)

(suite de la page précédente)

```
        </td>
    </tr>
    <?php endforeach; ?>
</table>
```

Dans la précédente section, nous avons assigné la variable "articles" à la view en utilisant la méthode `set()`. Les variables passées à la view sont disponibles dans les templates de view comme des variables locales, comme nous l'avons fait ci-dessus.

Vous avez peut-être remarqué que nous utilisons un objet appelé `$this->Html`. C'est une instance du *HtmlHelper*. CakePHP inclut plusieurs helpers de view qui peuvent créer des liens, des formulaires et des éléments de paginations. Vous pouvez en apprendre plus à propos des *Helpers (Assistants)* dans le chapitre de la documentation qui leur est consacré, mais le plus important ici est la méthode `link()`, qui générera un lien HTML avec le texte fourni (le premier paramètre) et l'URL (le second paramètre).

Quand vous spécifiez des URLs dans CakePHP, il est recommandé d'utiliser des tableaux ou des *routes nommées*. Ces syntaxes vous permettent de bénéficier du reverse routing fourni par CakePHP.

A partir de maintenant, si vous accédez à **http ://localhost :8765/articles/index**, vous devriez voir votre view qui liste les articles avec leur titre et leur lien.

Création de l'action View

Si vous cliquez sur le lien d'un article dans la page qui liste nos articles, vous tombez sur une page d'erreur vous indiquant que l'action n'a pas été implémentée. Vous pouvez corriger cette erreur en créant l'action manquante correspondante :

```
// Ajouter au fichier existant src/Controller/ArticlesController.php

public function view($slug = null)
{
    $article = $this->Articles->findBySlug($slug)->firstOrFail();
    $this->set(compact('article'));
}
```

Bien que cette action soit simple, nous avons utilisé quelques-unes des fonctionnalités de CakePHP. Nous commençons par utiliser la méthode `findBySlug()` qui est un *finder dynamique*. Cette méthode nous permet de créer une requête basique qui permet de récupérer des articles par un « slug » donné. Nous utilisons ensuite la méthode `firstOrFail()` qui nous permet de récupérer le premier enregistrement ou lancera une `NotFoundException` si aucun article correspondant n'est trouvé.

Notre action attend un paramètre `$slug`, mais d'où vient-il ? Si un utilisateur requête `/articles/view/first-post`, alors la valeur "first-post" sera passée à `$slug` par la couche de routing et de dispatching de CakePHP. Si nous rechargeons notre navigateur, nous aurons une nouvelle erreur, nous indiquant qu'il manque un template de View ; corrigeons cela.

Création du Template View

Créons le template de view pour notre action « view » dans **templates/Articles/view.php**.

```
<!-- Fichier : templates/Articles/view.php -->

<h1><?= h($article->title) ?></h1>
<p><?= h($article->body) ?></p>
```

(suite sur la page suivante)

(suite de la page précédente)

```
<p><small>Créé le : <?= $article->created->format(DATE_RFC850) ?></small></p>
<p><?= $this->Html->link('Modifier', ['action' => 'edit', $article->slug]) ?></p>
```

Vous pouvez vérifier que tout fonctionne en essayant de cliquer sur un lien de `/articles/index` ou en vous rendant manuellement sur une URL de la forme `/articles/view/first-post`.

Ajouter des articles

Maintenant que les views de lecture ont été créées, il est temps de rendre possible la création d'articles. Commencez par créer une action `add()` dans le `ArticlesController`. Notre controller doit maintenant ressembler à ceci :

```
// src/Controller/ArticlesController.php

namespace App\Controller;

use App\Controller\AppController;

class ArticlesController extends AppController
{

    public function initialize(): void
    {
        parent::initialize();

        $this->loadComponent('Paginator');
        $this->loadComponent('Flash'); // Inclusion du FlashComponent
    }

    public function index()
    {
        $articles = $this->Paginator->paginate($this->Articles->find());
        $this->set(compact('articles'));
    }

    public function view($slug)
    {
        $article = $this->Articles->findBySlug($slug)->firstOrFail();
        $this->set(compact('article'));
    }

    public function add()
    {
        $article = $this->Articles->newEmptyEntity();
        if ($this->request->is('post')) {
            $article = $this->Articles->patchEntity($article, $this->request->getData());

            // L'écriture de 'user_id' en dur est temporaire et
            // sera supprimée quand nous aurons mis en place l'authentification.
            $article->user_id = 1;

            if ($this->Articles->save($article)) {
                $this->Flash->success(__('Votre article a été sauvegardé.'));
```

(suite sur la page suivante)

(suite de la page précédente)

```
                return $this->redirect(['action' => 'index']);
            }
            $this->Flash->error(__('Impossible d\'ajouter votre article.'));
        }
        $this->set('article', $article);
    }
}
```

Note

Vous devez inclure le *Flash* dans tous les controllers où vous avez besoin de l'utiliser. Il est souvent conseillé de le charger directement dans le `AppController`.

Voici ce que l'action `add()` fait :

- Si la méthode HTTP de la requête est un POST, cela tentera de sauvegarder les données en utilisant le model Articles.
- Si pour une quelconque raison la sauvegarde ne se fait pas, cela rendra juste la view. Cela nous donne ainsi une chance de montrer les erreurs de validation ou d'autres messages à l'utilisateur.

Toutes les requêtes de CakePHP incluent un objet request qui est accessible via `$this->request`. L'objet request contient des informations à propos de la requête qui vient d'être reçue. Nous utilisons la méthode *`Cake\Http\ServerRequest::is()`* pour vérifier que la requête possède bien le verbe HTTP POST.

Les données passées en POST sont disponibles dans `$this->request->getData()`. Vous pouvez utiliser les fonctions *`pr()`* ou *`debug()`* pour afficher les données si vous voulez voir à quoi elles ressemblent. Pour sauvegarder les données, nous devons tout d'abord « marshaller » les données du POST en une Entity Article. L'Entity sera ensuite persistée en utilisant la classe ArticlesTable que nous avons créée plus tôt.

Après la sauvegarde de notre article, nous utilisons la méthode `success()` du FlashComponent pour définir le message en Session. La méthode `success` est fournie via les méthodes magiques de PHP[42]. Les messages Flash seront affichés sur la page suivante après redirection. Dans notre layout, nous avons `<?= $this->Flash->render() ?>` qui affichera un message Flash et le supprimera du stockage dans la session. Enfin, après la sauvegarde, nous utilisons *`Cake\Controller\Controller::redirect`* pour renvoyer l'utilisateur à la liste des articles. Le paramètre `['action' => 'index']` correspond à l'URL `/articles`, c'est-à-dire l'action index du `ArticlesController`. Vous pouvez vous référer à la méthode `Cake\Routing\Router::url()` dans la documentation API[43] pour voir les formats dans lesquels vous pouvez spécifier une URL.

Création du Template Add

Voici le code de notre template de la view « add » :

```
<!-- File: templates/Articles/add.php -->

<h1>Ajouter un article</h1>
<?php
    echo $this->Form->create($article);
    // Hard code the user for now.
    echo $this->Form->control('user_id', ['type' => 'hidden', 'value' => 1]);
    echo $this->Form->control('title');
    echo $this->Form->control('body', ['rows' => '3']);
```

(suite sur la page suivante)

42. https://php.net/manual/en/language.oop5.overloading.php#object.call
43. https://api.cakephp.org

(suite de la page précédente)

```
    echo $this->Form->button(__('Sauvegarder l\'article'));
    echo $this->Form->end();
?>
```

Nous utilisons le FormHelper pour générer l'ouverture du formulaire HTML. Voici le HTML que `$this->Form->create()` génère :

```
<form method="post" action="/articles/add">
```

Puisque nous appelons `create()` sans passer d'option URL, le `FormHelper` va partir du principe que le formulaire doit être soumis sur l'action courante.

La méthode `$this->Form->control()` est utilisée pour créer un élément de formulaire du même nom. Le premier paramètre indique à CakePHP à quel champ il correspond et le second paramètre vous permet de définir un très grand nombre d'options - dans notre cas, le nombre de lignes (rows) pour le textarea. Il y a un peu d'instrospection et de conventions utilisées ici. La méthode `control()` affichera des éléments de formulaire différents en fonction du champ du model spécifié et utilisera une inflection automatique pour définir le label associé. Vous pouvez personnaliser le label, les inputs ou tout autre aspect du formulaire en utilisant les options. La méthode `$this->Form->end()` ferme le formulaire.

Retournons à notre template **templates/Articles/index.php** pour ajouter un lien « Ajouter un article ». Avant le `<table>`, ajoutons la ligne suivante :

```
<?= $this->Html->link('Ajouter un article', ['action' => 'add']) ?>
```

Ajout de la génération de slug

Si nous sauvons un article tout de suite, la sauvegarde échouerait car nous ne créons pas l'attribut « slug » et la colonne correspondante est définie comme `NOT NULL`. Un slug est généralement une version « URL compatible » du titre d'un article. Nous pouvons utiliser le *callback beforeSave()* de l'ORM pour créer notre slug :

```
<?php
// dans src/Model/Table/ArticlesTable.php
namespace App\Model\Table;

use Cake\ORM\Table;
// la classe Text
use Cake\Utility\Text;
// la classe EventInterface
use Cake\Event\EventInterface;

// Ajouter la méthode suivante

public function beforeSave($event, $entity, $options)
{
    if ($entity->isNew() && !$entity->slug) {
        $sluggedTitle = Text::slug($entity->title);
        // On ne garde que le nombre de caractère correspondant à la longueur
        // maximum définie dans notre schéma
        $entity->slug = substr($sluggedTitle, 0, 191);
    }
}
```

Ce code est simple et ne prend pas en compte les potentiels doublons de slug. Mais nous nous occuperons de ceci plus tard.

Ajout de l'action Edit

Notre application peut maintenant sauvegarder des articles, mais nous ne pouvons pas modifier les articles existants. Rectifions cela maintenant. Ajoutez l'action suivante dans votre `ArticlesController` :

```
// dans src/Controller/ArticlesController.php

// Ajouter la méthode suivante.

public function edit($slug)
{
    $article = $this->Articles
        ->findBySlug($slug)
        ->firstOrFail();

    if ($this->request->is(['post', 'put'])) {
        $this->Articles->patchEntity($article, $this->request->getData());
        if ($this->Articles->save($article)) {
            $this->Flash->success(__('Votre article a été mis à jour.'));

            return $this->redirect(['action' => 'index']);
        }
        $this->Flash->error(__('Impossible de mettre à jour l\'article.'));
    }

    $this->set('article', $article);
}
```

Cette action va d'abord s'assurer que l'utilisateur essaie d'accéder à un enregistrement existant. Si vous n'avez pas passé de paramètre `$slug` ou que l'article n'existe pas, une `NotFoundException` sera lancée et le ErrorHandler de CakePHP rendra la page d'erreur appropriée.

Ensuite l'action va vérifier si la requête est une requête POST ou PUT. Si c'est le cas, nous utiliserons alors les données du POST/PUT pour mettre à jour l'entity de l'article en utilisant la méthode `patchEntity()`. Enfin, nous appelons la méthode `save()`, nous définissons un message Flash approprié et soit nous redirigeons, soit nous affichons les erreurs de validation en fonction du résultat de l'opération de sauvegarde.

Création du Template Edit

Le template edit devra ressembler à ceci :

```
<!-- Fichier : templates/Articles/edit.php -->

<h1>Modifier un article</h1>
<?php
    echo $this->Form->create($article);
    echo $this->Form->control('user_id', ['type' => 'hidden']);
    echo $this->Form->control('title');
    echo $this->Form->control('body', ['rows' => '3']);
    echo $this->Form->button(__('Sauvegarder l\'article'));
    echo $this->Form->end();
?>
```

Ce template affiche le formulaire de modification (avec les valeurs déjà remplies), ainsi que les messages d'erreurs de validation nécessaires.

Vous pouvez maintenant mettre à jour notre view index avec les liens pour modifier les articles :

```
<!-- Fichier : templates/Articles/index.php (liens de modification ajoutés) -->

<h1>Articles</h1>
<p><?= $this->Html->link("Ajouter un article", ['action' => 'add']) ?></p>
<table>
    <tr>
        <th>Titre</th>
        <th>Créé le</th>
        <th>Action</th>
    </tr>

    <!-- C'est ici que nous bouclons sur notre objet Query $articles pour afficher les␣
↪informations de chaque article -->

<?php foreach ($articles as $article): ?>
    <tr>
        <td>
            <?= $this->Html->link($article->title, ['action' => 'view', $article->slug])␣
↪?>
        </td>
        <td>
            <?= $article->created->format(DATE_RFC850) ?>
        </td>
        <td>
            <?= $this->Html->link('Modifier', ['action' => 'edit', $article->slug]) ?>
        </td>
    </tr>
<?php endforeach; ?>

</table>
```

Mise à jour des règles de validation pour les Articles

Jusqu'à maintenant, nos Articles n'avaient aucune validation de données. Occupons-nous de ça en utilisant un *validator* :

```
// src/Model/Table/ArticlesTable.php

// Ajouter ce "use" juste sous la déclaration du namespace pour importer
// la classe Validator
use Cake\Validation\Validator;

// Ajouter la méthode suivante.
public function validationDefault(Validator $validator): Validator
{
    $validator
        ->notEmptyString('title')
        ->minLength('title', 10)
        ->maxLength('title', 255)
```

(suite sur la page suivante)

(suite de la page précédente)

```
        ->notEmptyString('body')
        ->minLength('body', 10);

    return $validator;
}
```

La méthode `validationDefault()` indique à CakePHP comment valider les données quand la méthode `save()` est appelée. Ici, il est spécifié que les champs title et body ne peuvent pas être vides et qu'ils ont aussi des contraintes sur la longueur.

Le moteur de validation de CakePHP est à la fois puissant et flexible. Il vous fournit un jeu de règles sur des validations communes comme les adresses emails, les adresses IP, etc. mais aussi la flexibilité d'ajouter vos propres règles de validation. Pour plus d'informations, rendez-vous dans la section *Validation* de la documentation.

Maintenant que nos règles de validation sont en place, utilisons l'application et essayons d'ajouter un article avec un title ou un body vide pour voir ce qu'il se passe. Puisque nous avons utiliser la méthode *`Cake\View\Helper\FormHelper::control()`* du FormHelper pour créer les éléments de formulaire, nos messages d'erreurs de validation seront affichés automatiquement.

Ajout de l'Action de Suppression

Donnons maintenant la possibilité à nos utilisateurs de supprimer des articles. Commencez par créer une action `delete()` dans `ArticlesController` :

```
// src/Controller/ArticlesController.php

// Ajouter la méthode suivante.

public function delete($slug)
{
    $this->request->allowMethod(['post', 'delete']);

    $article = $this->Articles->findBySlug($slug)->firstOrFail();
    if ($this->Articles->delete($article)) {
        $this->Flash->success(__('L\'article {0} a été supprimé.', $article->title));

        return $this->redirect(['action' => 'index']);
    }
}
```

Ce code va supprimer l'article ayant le slug `$slug` et utilisera la méthode `$this->Flash->success()` pour afficher un message de confirmation à l'utilisateur après l'avoir redirigé sur `/articles`. Si l'utilisateur essaie d'aller supprimer un article avec une requête GET, la méthode `allowMethod()` lancera une exception. Les exceptions non capturées sont récupérées par le gestionnaire d'exception de CakePHP qui affichera une belle page d'erreur. Il existe plusieurs *Exceptions* intégrées qui peuvent être utilisées pour remonter les différentes erreurs HTTP que votre application aurait besoin de générer.

> **Avertissement**
>
> Permettre de supprimer des données via des requêtes GET est très dangereux, car il est possible que des crawlers suppriment accidentellement du contenu. C'est pourquoi nous utilisons la méthode `allowMethod()` dans notre controller.

Puisque nous exécutons seulement de la logique et redirigeons directement sur une autre action, cette action n'a pas de template. Vous devez ensuite mettre à jour votre template index pour ajouter les liens qui permettront de supprimer les articles :

```
<!-- Fichier : templates/Articles/index.php (ajout des liens de suppression) -->

<h1>Articles</h1>
<p><?= $this->Html->link("Add Article", ['action' => 'add']) ?></p>
<table>
    <tr>
        <th>Titre</th>
        <th>Créé le</th>
        <th>Action</th>
    </tr>

    <!-- C'est ici que nous bouclons sur notre objet Query $articles pour afficher les␣
↪informations de chaque article -->

<?php foreach ($articles as $article): ?>
    <tr>
        <td>
            <?= $this->Html->link($article->title, ['action' => 'view', $article->slug])␣
↪?>
        </td>
        <td>
            <?= $article->created->format(DATE_RFC850) ?>
        </td>
        <td>
            <?= $this->Html->link('Modifier', ['action' => 'edit', $article->slug]) ?>
            <?= $this->Form->postLink(
                'Supprimer',
                ['action' => 'delete', $article->slug],
                ['confirm' => 'Êtes-vous sûr ?'])
            ?>
        </td>
    </tr>
<?php endforeach; ?>

</table>
```

Utiliser `postLink()` va créer un lien qui utilisera du JavaScript pour faire une requête POST et supprimer notre article.

> **Note**
>
> Ce code de view utilise également le `FormHelper` pour afficher à l'utilisateur une boîte de dialogue de confirmation en JavaScript avant la suppression effective de l'article.

Maintenant que nous avons un minimum de gestion sur nos articles, il est temps de créer des actions basiques pour nos tables *Tags et Users*.

Tutoriel CMS - Tags et Users

Maintenant que nous avons implémenté une gestion basique de la création d'articles, il est temps de permettre à plusieurs auteurs de travailler sur notre CMS. Dans les étapes précédentes, nous avons créé nos models, nos views et nos controllers à la main. Cette fois, nous allons utiliser *Console Bake* pour créer la base de notre code. Bake est un outil CLI (Command Line Interface) de génération de code qui se base sur les conventions de CakePHP pour créer des applications CRUD (Create, Read, Update, Delete) basique très rapidement. Nous allons utiliser `bake` pour créer le code relatif à la gestion d'utilisateurs :

```
cd /path/to/our/app

bin/cake bake model users
bin/cake bake controller users
bin/cake bake template users
```

Ces 3 commandes vont générer :

- Les fichiers de Table, Entity, et Fixture.
- Le Controller
- Les templates CRUD.
- Les fichiers de Tests pour chaque classe générée.

Bake va aussi utiliser les conventions CakePHP pour définir les associations et les validations pour vos models.

Ajouter un système de Tags aux Articles

Avec plusieurs utilisateurs capables d'accéder à notre petit CMS, il serait bien d'avoir, pour notre application CMS, un moyen de catégoriser notre contenu. Nous allons donc utiliser des tags pour permettre aux utilisateurs d'ajouter des catégories et des labels à leurs contenus. Une fois de plus, nous allons utiliser `bake` pour générer rapidement un code de base :

```
bin/cake bake all tags
```

Une fois que le code de base est généré, créez quelques tags en vous rendant sur la page **http ://localhost :8765/tags/add**.

Maintenant que nous avons une table Tags, nous pouvons créer une association entre la table Articles et la table Tags. Nous pouvons le faire en ajoutant le code suivant à la méthode `initialize` de ArticlesTable :

```
public function initialize(array $config): void
{
    $this->addBehavior('Timestamp');
    $this->belongsToMany('Tags'); // Ajoutez cette ligne
}
```

Cette association fonctionnera avec cette définition qui tient sur une seule ligne car nous avons suivi les conventions de CakePHP lors de la création de nos tables. Pour plus d'informations, rendez-vous dans la section *Associations - Lier les Tables Ensemble*.

Mettre à jour la gestion des Articles pour permettre d'ajouter des Tags

Maintenant que notre application gère les tags, nous devons donner la possibilité à nos utilisateurs d'ajouter les tags sur les articles. Premièrement, mettez à jour l'action `add` pour qu'elle ressemble à ceci :

```
<?php
// dans src/Controller/ArticlesController.php
namespace App\Controller;
```

(suite sur la page suivante)

(suite de la page précédente)

```
use App\Controller\AppController;

class ArticlesController extends AppController
{
    public function add()
    {
        $article = $this->Articles->newEmptyEntity();
        if ($this->request->is('post')) {
            $article = $this->Articles->patchEntity($article, $this->request->getData());

            // L'écriture de 'user_id' en dur est temporaire et
            // sera supprimée quand nous aurons mis en place l'authentification.
            $article->user_id = 1;

            if ($this->Articles->save($article)) {
                $this->Flash->success(__('Votre article a été sauvegardé.'));

                return $this->redirect(['action' => 'index']);
            }
            $this->Flash->error(__('Impossible de sauvegarder l\'article.'));
        }
        // Récupère une liste des tags.
        $tags = $this->Articles->Tags->find('list');

        // Passe les tags au context de la view
        $this->set('tags', $tags);

        $this->set('article', $article);
    }

    // Les autres actions
}
```

Les lignes de code ajoutées chargent une liste des tags sous forme de tableau associatif de la forme `id => title`. Ce format nous permet de créer un nouvel input de tags dans notre template. Ajoutez la ligne suivante dans le bloc PHP avec les autres appels à `control()` dans **templates/Articles/add.php** :

```
echo $this->Form->control('tags._ids', ['options' => $tags]);
```

Cela rendra un select multiple qui utilisera la variable `$tags` pour générer les options du select. Vous devriez maintenant créer quelques articles en leur mettant des tags car dans la section suivante, nous allons ajouter la possibilité de trouver des articles par leurs tags.

Vous devriez également mettre à jour la méthode `edit` pour permettre l'ajout et la modification de tags sur les articles existant. La méthode `edit` devrait maintenant ressembler à ceci :

```
public function edit($slug)
{
    $article = $this->Articles
        ->findBySlug($slug)
        ->contain('Tags') // charge les Tags associés
        ->firstOrFail();
```

(suite sur la page suivante)

(suite de la page précédente)

```
    if ($this->request->is(['post', 'put'])) {
        $this->Articles->patchEntity($article, $this->request->getData());
        if ($this->Articles->save($article)) {
            $this->Flash->success(__('Votre article a été modifié.'));

            return $this->redirect(['action' => 'index']);
        }
        $this->Flash->error(__('Impossible de mettre à jour votre article.'));
    }

    // Récupère une liste des tags.
    $tags = $this->Articles->Tags->find('list');

    // Passe les tags au context de la view
    $this->set('tags', $tags);

    $this->set('article', $article);
}
```

Pensez à ajouter le nouveau select multiple qui permet de sélectionner les tags comme nous l'avons fait dans le template **add.php** au template **templates/Articles/edit.php**.

Trouver des Articles via les Tags

Une fois que les utilisateurs ont catégorisé leurs contenus, ils voudront probablement retrouver ces contenus en fonction des tags utilisés. Pour développer ces fonctionnalités, nous allons implémenter une nouvelle route, une nouvelle action de controller et une fonction de finder pour chercher les articles par tags.

Idéalement, nous voulons une URL qui ressemblera à **http ://localhost :8765/articles/tagged/funny/cat/gifs**. Cela nous permettra de trouver tous les articles avec le tag "funny", "cat" ou "gifs". Nous avons tout d'abord besoin d'ajouter une nouvelle route. Votre fichier **config/routes.php** (avec les commentaires générés par bake supprimés) devra ressembler à :

```
<?php
use Cake\Routing\Route\DashedRoute;
use Cake\Routing\Router;

Router::defaultRouteClass(DashedRoute::class);

$routes->scope('/', function (RouteBuilder $builder) {
    $builder->connect('/', ['controller' => 'Pages', 'action' => 'display', 'home']);
    $builder->connect('/pages/*', ['controller' => 'Pages', 'action' => 'display']);

    // Ceci est la route à ajouter pour notre nouvelle action.
    // Le `*` à la fin permet de préciser à CakePHP que cette action
    // a des paramètres qui lui seront passés
    $builder->scope('/articles', function (RouteBuilder $builder) {
        $builder->connect('/tagged/*', ['controller' => 'Articles', 'action' => 'tags']);
    });

    $builder->fallbacks();
});
```

Le code ci-dessus définit une nouvelle "route" qui permet de connecter le chemin URL **/articles/tagged/** à

`ArticlesController::tags()`. En définissant des routes, vous pouvez isoler le format de vos URLs de la manière dont elles sont implémentées. Si nous venions à visiter **http ://localhost :8765/articles/tagged**, nous verrions une page d'erreur de CakePHP vous indiquant que l'action du controller n'existe pas. Créons de ce pas cette nouvelle méthode. Dans **src/Controller/ArticlesController.php**, ajoutez ce qui suit :

```
public function tags()
{
    // La clé 'pass' est fournie par CakePHP et contient tous les
    // segments d'URL passés dans la requête
    $tags = $this->request->getParam('pass');

    // Utilisation de ArticlesTable pour trouver les articles taggés
    $articles = $this->Articles->find('tagged', [
        'tags' => $tags
    ])
    ->all();

    // Passage des variables dans le contexte de la view du template
    $this->set([
        'articles' => $articles,
        'tags' => $tags
    ]);
}
```

Pour accéder aux autres parties des données de la requête, consultez la section *ServerRequest*.

Puisque les arguments passés sont aussi fournis comme paramètres de la méthode d'action, nous pourrions également écrire l'action en utilisant les arguments variadic de PHP :

```
public function tags(...$tags)
{
    // Utilisation de ArticlesTable pour trouver les articles taggés
    $articles = $this->Articles->find('tagged', [
        'tags' => $tags
    ])
    ->all();

    // Passage des variable dans le contexte de la view du template
    $this->set([
        'articles' => $articles,
        'tags' => $tags
    ]);
}
```

Création de la Méthode Finder

Dans CakePHP, nous aimons garder nos actions de controller le plus minimaliste possible et mettons la majorité de la logique de notre application dans la couche model. Si vous veniez à visiter l'URL **/articles/tagged**, vous verriez une erreur vous indiquant que la méthode `findTagged()` n'existe pas. Dans **src/Model/Table/ArticlesTable.php**, ajoutez le code suivant :

```
// Ajouter ce 'use' juste sous la déclaration du namespace pour importer
// la classe Query
use Cake\ORM\Query;
```

(suite sur la page suivante)

(suite de la page précédente)

```
// L'argument $query est une instance du Query builder.
// Le tableau $options va contenir l'option 'tags' que nous avons passé
// à find('tagged') dans notre action de controller.
public function findTagged(Query $query, array $options)
{
    $columns = [
        'Articles.id', 'Articles.user_id', 'Articles.title',
        'Articles.body', 'Articles.published', 'Articles.created',
        'Articles.slug',
    ];

    $query = $query
        ->select($columns)
        ->distinct($columns);

    if (empty($options['tags'])) {
        // si aucun tag n'est fourni, trouvons les articles qui n'ont pas de tags
        $query->leftJoinWith('Tags')
            ->where(['Tags.title IS' => null]);
    } else {
        // Trouvons les articles qui ont au moins un des tags fourni
        $query->innerJoinWith('Tags')
            ->where(['Tags.title IN' => $options['tags']]);
    }

    return $query->group(['Articles.id']);
}
```

Nous venons d'implémenter *un custom finder*. Ce concept très pratique de CakePHP vous permet de définir des requêtes réutilisables. Les méthodes finder récupèrent toujours en paramètres un objet *Query Builder* et un tableau d'options. Les finders peuvent manipuler la requête et ajouter n'importe quels condition ou critère. Une fois la logique terminée, le finder doit retourner une instance modifiée de l'objet query. Dans notre finder, nous utilisons les méthodes `distinct()` et `leftJoin()` qui nous permettent de trouver les articles différents qui ont les tags correspondant.

Création de la View

Si vous visitez à nouveau **/articles/tagged**, CakePHP vous affichera une nouvelle erreur qui vous fait savoir qu'il manque le fichier de view. A présent, créons le fichier de vue pour notre action `tags()` action :

```
<!-- Dans templates/Articles/tags.php -->
<h1>
    Articles avec les tags
    <?= $this->Text->toList(h($tags), 'or') ?>
</h1>

<section>
<?php foreach ($articles as $article): ?>
    <article>
        <!-- Utilisation du HtmlHelper pour créer le lien -->
        <h4><?= $this->Html->link(
            $article->title,
```

(suite sur la page suivante)

(suite de la page précédente)

```
            ['controller' => 'Articles', 'action' => 'view', $article->slug]
        ) ?></h4>
        <span><?= h($article->created) ?></span>
    </article>
<?php endforeach; ?>
</section>
```

Dans le code ci-dessus, nous utilisons les Helpers *Html* et *Text* pour nous aider à générer le contenu de notre view. Nous utilisons également la fonction raccourcie *h* pour échapper le contenu HTML. Pensez à utiliser `h()` quand vous affichez des données pour éviter les injections de HTML.

Le fichier **tags.php** que nous venons de créer suit les conventions CakePHP pour les templates de view. La convention est d'utiliser le nom de l'action du controller en minuscule et avec un underscore en séparateur.

Vous avez peut-être remarqué que nous utilisons les variables `$tags` et `$articles` dans notre template de view. Quand nous utilisons la méthode `set()` dans notre controller, nous définissons les variables qui doivent être envoyées à notre view. La classe View fera alors en sorte de passer les variables au scope du template comme variables locales.

Vous devriez maintenant être capable de visiter la page **/articles/tagged/funny** et voir tous les articles avec le tag "funny".

Améliorer la Gestion des Tags

Pour le moment, ajouter des tags est assez fastidieux puisque les rédacteurs auront besoin de créer les tags à utiliser avant de les assigner. Nous pouvons améliorer l'UI de notre gestion de tag en utilisant une liste de valeurs séparées par des virgules. Cela nous permettra d'améliorer l'expérience utilisateur et de découvrir d'autres fonctionnalités de l'ORM.

Ajouter un Champ Pré-calculé

Puisque nous souhaitons une manière simple d'accéder aux tags formattés pour une entity, nous ajoutons un champ virtuel/pré-calculé pour l'entity. Dans **src/Model/Entity/Article.php** ajoutez la méthode suivante :

```
// Ajouter ce 'use' juste sous la déclaration du namespace pour importer
// la classe Collection
use Cake\Collection\Collection;

// Mettez à jour la propriété accessible pour qu'elle contienne `tag_string`
protected array $_accessible = [
    //autres champs...
    'tag_string' => true
];
protected function _getTagString()
{
    if (isset($this->_fields['tag_string'])) {
        return $this->_fields['tag_string'];
    }
    if (empty($this->tags)) {
        return '';
    }
    $tags = new Collection($this->tags);
    $str = $tags->reduce(function ($string, $tag) {
        return $string . $tag->title . ', ';
    }, '');
```

(suite sur la page suivante)

(suite de la page précédente)

```
    return trim($str, ', ');
}
```

Cela nous permettra d'accéder à la propriété virtuelle `$article->tag_string`. Nous utiliserons cette propriété plus tard dans nos contrôles (control).

Mettre à jour nos View

Maintenant que notre entity est mise à jour, nous pouvons ajouter un nouvel élément de contrôle pour nos tags. Dans **templates/Articles/add.php** et **templates/Articles/edit.php**, remplacez l'élément de contrôle existant `tags._ids` avec la déclaration suivante :

```
echo $this->Form->control('tag_string', ['type' => 'text']);
```

Nous devrons également mettre à jour le modèle de vue d'article. Dans **templates/Articles/view.php**, ajoutez la ligne comme indiqué :

```
<!-- Fichier: templates/Articles/view.php -->

<h1><?= h($article->title) ?></h1>
<p><?= h($article->body) ?></p>
// Add the following line
<p><b>Tags:</b> <?= h($article->tag_string) ?></p>
```

Vous devriez aussi mettre à jour la méthode de vue pour permettre de récupérer les tags existants :

```
// fichier src/Controller/ArticlesController.php

public function view($slug = null)
{
   // Mettre à jour la récupération des tags avec contain()
   $article = $this->Articles
       ->findBySlug($slug)
        ->contain('Tags')
        ->firstOrFail();
    $this->set(compact('article'));
}
```

Persister la Chaîne de Tags

Maintenant que nous voyons les tags existant sous forme d'une chaîne, nous avons besoin de sauvegarder les tags sous ce format. Puisque que nous avons rendu `tag_string` accessible, l'ORM copiera les données de la requête dans notre entity. Nous pouvons utiliser le hook `beforeSave()` pour parser la chaîne de tags et trouver/construire les entities correspondantes. Ajoutez le code suivant à **src/Model/Table/ArticlesTable.php** :

```
public function beforeSave($event, $entity, $options)
{
    if ($entity->tag_string) {
        $entity->tags = $this->_buildTags($entity->tag_string);
    }

    // Le code déjà existant
```

(suite sur la page suivante)

(suite de la page précédente)

```
}

protected function _buildTags($tagString)
{
    // Trim des tags
    $newTags = array_map('trim', explode(',', $tagString));
    // Retire les tags vides
    $newTags = array_filter($newTags);
    // Dé-doublonne les tags
    $newTags = array_unique($newTags);

    $out = [];
    $query = $this->Tags->find()
        ->where(['Tags.title IN' => $newTags])
        ->all();

    // Retire les tags existant de la liste des nouveaux tags.
    foreach ($query->extract('title') as $existing) {
        $index = array_search($existing, $newTags);
        if ($index !== false) {
            unset($newTags[$index]);
        }
    }
    // Ajout des tags existant.
    foreach ($query as $tag) {
        $out[] = $tag;
    }
    // Ajout des nouveaux tags.
    foreach ($newTags as $tag) {
        $out[] = $this->Tags->newEntity(['title' => $tag]);
    }

    return $out;
}
```

Si vous créez ou modifiez maintenant des articles, vous devriez pouvoir enregistrer les balises sous forme de liste de balises séparées par des virgules et créer automatiquement les balises et les enregistrements de liaison.

Bien que ce code soit plus compliqué que tout ce que nous avons fait jusqu'ici, il permet de mettre en avant les fonctions avancées de l'ORM : vous pouvez manipuler le résultat de la requête en utilisant les méthodes de la classe Collection (voir la section *Collections*) et pouvez également gérer les scénarios où vous avez besoin de créer des entities à la volée.

Remplir automatiquement les Tags

Avant de terminer, nous aurons besoin d'un mécanisme qui chargera les Tag associés (le cas échéant) chaque fois que nous chargerons un article.

Dans votre **src/Model/Table/ArticlesTable.php**, changez :

```
public function initialize(array $config): void
{
    $this->addBehavior('Timestamp');
    // Modifiez cette ligne
```

(suite sur la page suivante)

(suite de la page précédente)

```
    $this->belongsToMany('Tags', [
        'joinTable' => 'articles_tags',
        'dependent' => true
    ]);
}
```

Cela indiquera au modèle de table Articles qu'une table de jointure est associée avec des tags. L'option "dépendent" indique à la table de supprimer tout enregistrement associé de la table de jointure si un article est supprimé.

Enfin, mettez à jour les appels de la méthode `findBySlug()` dans **src/Controller/ArticlesController.php** :

```
public function edit($slug)
{
    // Mettez à jour cette ligne
    $article = $this->Articles
        ->findBySlug($slug)
        ->contain('Tags')
        ->firstOrFail();
...
}

public function view($slug = null)
{
    // Mettez à jour cette ligne
    $article = $this->Articles
        ->findBySlug($slug)
        ->contain('Tags')
        ->firstOrFail();
    $this->set(compact('article'));
}
```

La méthode `contain ()` indique à l'objet `ArticlesTable` de remplir également l'association Tags lorsque l'article est chargé. Maintenant, quand tag_string est appelé pour une entité Article, il y aura des données présentes pour créer la chaîne !

Dans le chapitre suivant, nous ajouter une couche *d'authentification*.

Tutoriel CMS - Authentification

Maintenant que nous avons des utilisateurs dans notre CMS, nous devons leur donner la possibilité de se connecter en utilisant le plugin cakephp/authentication [44].Nous commencerons par nous assurer que les mots de passe sont stockés en toute sécurité dans notre base de données. Ensuite, nous allons fournir une connexion et une déconnexion en état de marche, et permettre aux nouveaux utilisateurs de s'inscrire.

Installation du Plugin d'Authentication

Utilisez composer pour installer le Plugin d'authentification :

```
composer require cakephp/authentication:^2.0
```

44. https://book.cakephp.org/authentication/2

Ajout du Hash du Mot de Passe

Vous devez avoir créé le `Controller`, `Table`, `Entity` et les templates pour la table `utilisateurs` de votre base de données. Vous pouvez le faire manuellement comme vous l'avez fait auparavant pour ArticlesController, ou vous pouvez utiliser le Shell bake pour générer les classes pour vous en tapant :

```
bin/cake bake all users
```

Si vous créez ou mettez à jour un utilisateur, vous remarquerez que les mots de passe sont stockés en clair, ce qui est évidemment très mauvais en terme de sécurité, règlons cela.

Corriger ce point nous permet de parler un peu plus de la couche model de CakePHP. Dans CakePHP, nous séparons les méthodes qui s'occupent des collections d'objets et d'un seul objet en différentes classes. Les méthodes qui s'occupent de collections d'entity sont dans les classes `Table` tandis que les fonctionnalités liées à un seul enregistrement sont mises dans les classes `Entity`.

Par exemple, hasher un mot de passe se fait enregistrement par enregistrement, c'est pourquoi nous allons implémenter ce comportement dans l'objet Entity. Puisque nous voulons hasher le mot de passe à chaque fois qu'il est défini, nous allons utiliser une méthode mutator/setter. Par convention, CakePHP appellera les méthodes de setter chaque fois qu'une propriété se voit affecter une valeur dans une entity. Ajoutons un setter pour le mot de passe. Dans **src/Model/Entity/User.php**, ajoutez le code suivant :

```php
<?php
namespace App\Model\Entity;

use Authentication\PasswordHasher\DefaultPasswordHasher; // Ajouter cette ligne
use Cake\ORM\Entity;

class User extends Entity
{
    // Tout le code de bake sera ici.

    // Ajoutez cette méthode
    protected function _setPassword(string $password) : ?string
    {
        if (strlen($password) > 0) {
            return (new DefaultPasswordHasher())->hash($password);
        }
        return null;
    }
}
```

Maintenant, rendez-vous sur **http ://localhost :8765/users** pour voir une liste des utilisateurs existants. N'oubliez pas que votre serveur local doit fonctionner. Démarrez un serveur PHP autonome utilisant `bin/cake server`.

Vous pouvez modifier l'utilisateur par défaut qui a été créé pendant le chapitre *Installation* du tutoriel. Si vous changez le mot de passe de l'utilisateur, vous devriez voir une version hashée du mot de passe à la place de la valeur par défaut sur l'action index ou view. CakePHP hashe les mots de passe, par défaut, avec bcrypt [45]. Nous recommandons bcrypt pour toutes les nouvelles applications afin de garentir à votre application un niveau de sécurité élevé. C'est l" algorithme de hachage de mot de passe recommandé pour PHP [46].

Note

45. https://codahale.com/how-to-safely-store-a-password/
46. https://www.php.net/manual/en/function.password-hash.php

Créez maintenant un mot de passe haché pour au moins un des comptes utilisateurs ! Il sera nécessaire dans les prochaines étapes. Après avoir mis à jour le mot de passe, vous verrez une longue chaîne stockée dans la colonne du mot de passe. Notez que bcrypt générera un hachage différent même pour le même mot de passe enregistré deux fois.

Ajouter la Connexion

Il est maintenant temps de configurer le Plugin d'authentification. Le plugin gérera le processus d'authentification en utilisant 3 classes différentes :

- `Application` utilisera le middleware d'authentification et fournira un AuthenticationService contenant toute la configuration que nous voulons définir, comment nous allons vérifier les informations d'identification, et où les trouver.
- `AuthenticationService` sera une classe utilitaire pour vous permettre de configurer le processus d'authentification.
- `AuthenticationMiddleware` sera exécuté dans le cadre de la file d'attente du middleware, c'est à dire avant que vos contrôleurs ne soient traités par le framework, et collectera les informations d'identification et les traitera pour vérifier si l'utilisateur est authentifié.

Si vous vous en souvenez, par le passé nous utilisions *AuthComponent* afin de gérer toutes ces étapes. A présent, la logique est divisée en classes spécifiques et le processus d'authentification se déroule avant votre couche de contrôleur. Il vérifie d'abord si l'utilisateur est authentifié (en fonction de la configuration que vous avez fournie) et injecte l'utilisateur ainsi que le résultat de l'authentification dans la requête afin que vous puissiez les utiliser ultérieurement.

Dans **src/Application.php**, ajoutez les imports suivants :

```
// Dans src/Application.php, ajoutez les imports suivants
use Authentication\AuthenticationService;
use Authentication\AuthenticationServiceInterface;
use Authentication\AuthenticationServiceProviderInterface;
use Authentication\Middleware\AuthenticationMiddleware;
use Cake\Routing\Router;
use Psr\Http\Message\ServerRequestInterface;
```

Ensuite, implémentez l'interface d'authentification pour votre classe `Application` :

```
// dans src/Application.php
class Application extends BaseApplication
    implements AuthenticationServiceProviderInterface
{
```

Puis ajoutez le code suivant :

```
    // src/Application.php
    public function middleware(MiddlewareQueue $middlewareQueue): MiddlewareQueue
    {
        $middlewareQueue
            // ... other middleware added before
            ->add(new RoutingMiddleware($this))
            // add Authentication after RoutingMiddleware
            ->add(new AuthenticationMiddleware($this));

        return $middlewareQueue;
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
  public function getAuthenticationService(ServerRequestInterface $request):␣
↪AuthenticationServiceInterface
  {
      $authenticationService = new AuthenticationService([
          'unauthenticatedRedirect' => Router::url('/users/login'),
          'queryParam' => 'redirect',
      ]);

      // Charge les identifiants et s'assure que nous vérifions les champs e-mail et mot␣
↪de passe
      $authenticationService->loadIdentifier('Authentication.Password', [
          'fields' => [
              'username' => 'email',
              'password' => 'password',
          ]
      ]);

      // Charge les authenticators, nous voulons celui de session en premier
      $authenticationService->loadAuthenticator('Authentication.Session');
      // Configure la vérification des données du formulaire pour choisir l'email et le␣
↪mot de passe
      $authenticationService->loadAuthenticator('Authentication.Form', [
          'fields' => [
              'username' => 'email',
              'password' => 'password',
          ],
          'loginUrl' => Router::url('/users/login'),
      ]);

      return $authenticationService;
  }

Ajoutez le code suivant dans votre classe ``AppController``::

  // src/Controller/AppController.php
  public function initialize(): void
  {
      parent::initialize();
      $this->loadComponent('RequestHandler');
      $this->loadComponent('Flash');

      // Ajoutez cette ligne pour vérifier le résultat de l'authentification et␣
↪verrouiller votre site
      $this->loadComponent('Authentication.Authentication');
```

Désormais, à chaque demande, le `AuthenticationMiddleware` inspectera la session contenue dans la requête afin d'y rechercher un utilisateur authentifié. Si nous chargeons la page `/users/login`, il inspectera également les données de formulaire soumises (le cas échéant) pour extraire les informations d'identification. Par défaut, les informations d'identification seront extraites des champs `username` and `password` dans les données de la demande. Le résultat de l'authentification sera injecté dans un attribut de requête nommé `authentification`. Vous pouvez consulter le résultat à tout moment en utilisant `$this->request->getAttribute('authentication')` à partir des actions de votre contrôleur. Toutes vos pages seront restreintes car le `AuthenticationComponent` vérifie le résultat à chaque demande. Lorsqu'il ne parvient pas à trouver un utilisateur authentifié, il redirige l'utilisateur sur la page `/users/`

`login`. Notez qu'à ce stade, le site ne fonctionnera pas car nous n'avons pas encore de page de connexion. Si vous visitez votre site, vous obtiendrez une boucle de redirection infinie. Alors, réglons ça !

Dans votre `UsersController`, ajoutez le code suivant :

```
public function beforeFilter(\Cake\Event\EventInterface $event)
{
    parent::beforeFilter($event);
    // Configurez l'action de connexion pour ne pas exiger d'authentification,
    // évitant ainsi le problème de la boucle de redirection infinie
    $this->Authentication->addUnauthenticatedActions(['login']);
}

public function login()
{
    $this->request->allowMethod(['get', 'post']);
    $result = $this->Authentication->getResult();
    // indépendamment de POST ou GET, rediriger si l'utilisateur est connecté
    if ($result && $result->isValid()) {
        // rediriger vers /articles après la connexion réussie
        $redirect = $this->request->getQuery('redirect', [
            'controller' => 'Articles',
            'action' => 'index',
        ]);

        return $this->redirect($redirect);
    }
    // afficher une erreur si l'utilisateur a soumis un formulaire
    // et que l'authentification a échoué
    if ($this->request->is('post') && !$result->isValid()) {
        $this->Flash->error(__('Votre identifiant ou votre mot de passe est incorrect.
↪'));
    }
}
```

Ajoutez la logique du template pour votre action de connexion :

```
<!-- dans /templates/Users/login.php -->
<div class="users form">
    <?= $this->Flash->render() ?>
    <h3>Connexion</h3>
    <?= $this->Form->create() ?>
    <fieldset>
        <legend><?= __('Veuillez s\'il vous plaît entrer votre nom d\'utilisateur et␣
↪votre mot de passe') ?></legend>
        <?= $this->Form->control('email', ['required' => true]) ?>
        <?= $this->Form->control('password', ['required' => true]) ?>
    </fieldset>
    <?= $this->Form->submit(__('Login')); ?>
    <?= $this->Form->end() ?>

    <?= $this->Html->link("Ajouter un utilisateur", ['action' => 'add']) ?>
</div>
```

Maintenant, la page de connexion nous permettra de nous connecter correctement à l'application. Testez-le en affichant

n'importe quelle page de votre site. Après avoir été redirigé à la page `/users/login`, saisissez l'email et le mot de passe choisis lors de la création de votre utilisateur. Vous devriez être redirigé avec succès après la connexion.

Nous devons ajouter quelques détails supplémentaires pour configurer notre application. Nous voulons que toutes les pages `view` and `index` soient accessibles sans connexion, nous allons donc ajouter cette configuration spécifique dans AppController

```
// dans src/Controller/AppController.php
public function beforeFilter(\Cake\Event\EventInterface $event)
{
    parent::beforeFilter($event);
    // pour tous les contrôleurs de notre application, rendre les actions
    // index et view publiques, en ignorant la vérification d'authentification
    $this->Authentication->addUnauthenticatedActions(['index', 'view']);
}
```

Note

Si aucun de vos utilisateurs ne possède de mot de passe hashé, commentez le bloc `$this->loadComponent('Authentication.Authentication')` dans votre AppController ainsi que toutes les autres lignes dans lesquelles le composant Authenticationest est utilisé. Puis allez à `/users/add` Après avoir sauvegardé le mot de passe pour l'utilisateur, décommentez les lignes que vous venez tout juste de commenter.

Essayez-le en visitant `/articles/add` avant de vous connecter ! Puisque l'action n'est pas autorisée, vous serez redirigé vers la page de connexion. Après vous être connecté avec succès, CakePHP vous redirigera automatiquement vers `/articles/add`.

Ajout de la Déconnexion

Ajoutez l'action de logout à la classe `UsersController` :

```
// dans src/Controller/UsersController.php
public function logout()
{
    $result = $this->Authentication->getResult();
    // indépendamment de POST ou GET, rediriger si l'utilisateur est connecté
    if ($result && $result->isValid()) {
        $this->Authentication->logout();

        return $this->redirect(['controller' => 'Users', 'action' => 'login']);
    }
}
```

Vous pouvez maintenant visiter `/users/logout` pour vous déconnecter. Vous devriez alors être envoyé à la page de connexion.

Autoriser la Création de Compte

Si vous n'êtes pas connecté et essayez de visiter **/users/add**, vous serez redirigé sur la page de connexion. Puisque nous voulons autoriser nos utilisateurs à créer un compte sur notre application, ajoutez ceci à votre `UsersController` :

```
// Ajoutez la méthode beforeFilter au UsersController
$this->Authentication->addUnauthenticatedActions(['login', 'add']);
```

Le code ci-dessus indique à `AuthenticationComponent` que la méthode `add()` du `UsersController` peut être visitée *sans* être authentifié ou avoir besoin d'autorisation. Vous pouvez prendre le temps de nettoyer **Users/add.php** en retirant les liens qui n'ont plus de sens pour cette page ou passer à la section suivante. Nous ne nous occuperons pas des actions d'édition, d'affichage ou de liste spécifiques aux utilisateurs, mais c'est un exercice que vous pouvez faire par vous-même.

Maintenant que les utilisateurs peuvent se connecter, nous voulons limiter les utilisateurs à modifier uniquement les articles qui ils ont été créés par : doc : *application des politiques d'autorisation <./autorisation>*.

Tutoriel CMS - Autorisation

Maintenant que nos utilisateurs peuvent se connecter à notre CMS, nous voulons appliquer des règles d'autorisation pour nous assurer que chaque utilisateur ne puisse éditer que les messages dont il est l'auteur. Nous allons utiliser le plugin authorization [47] pour nous en assurer.

Installer le plugin Authorization

Utilisez composer pour installer le plugin Authorization :

```
composer require "cakephp/authorization:^2.0"
```

Chargez le plugin en ajoutant le code suivant à la méthode `bootstrap()` dans le fichier **src/Application.php** :

```
$this->addPlugin('Authorization');
```

Activer le plugin Authorization

Le plugin Authorization s'intègre dans votre application grâce à la couche middleware et optionnellement par un composant dans vos contrôleurs pour faciliter les vérifications des droits. Premièrement, attachons-nous au middleware. Dans **src/Application.php** ajoutez le code suivant dans les import de la classe :

```
use Authorization\AuthorizationService;
use Authorization\AuthorizationServiceInterface;
use Authorization\AuthorizationServiceProviderInterface;
use Authorization\Middleware\AuthorizationMiddleware;
use Authorization\Policy\OrmResolver;
use Psr\Http\Message\ResponseInterface;
```

Ajoutez `AuthorizationProviderInterface` aux interfaces déjà importées par votre application :

```
class Application extends BaseApplication
    implements AuthenticationServiceProviderInterface,
    AuthorizationServiceProviderInterface
```

Enfin, ajoutez le code suivant à votre méthode `middleware()` :

```
// Attention: Ajoutez Authorization **après** Authentication
$middlewareQueue->add(new AuthorizationMiddleware($this));
```

`AuthorizationMiddleware` va appeler une méthode hook sur votre application qui va permettre de définir le `AuthorizationService` à utiliser. Ajoutez la méthode suivante au fichier **src/Application.php** :

47. https://book.cakephp.org/authorization/2

```
public function getAuthorizationService(ServerRequestInterface $request):␣
↪AuthorizationServiceInterface
{
    $resolver = new OrmResolver();

    return new AuthorizationService($resolver);
}
```

L'OrmResolver permet au plugin Authorization de trouver les classes de stratégies (policies) pour les entités et les requêtes de l'ORM. Des resolvers supplémentaires peuvent être utilisés pour trouver des stratégies pour d'autres types de ressources.

Maintenant, ajoutons `AuthorizationComponent` à `AppController`. Dans **src/Controller/AppController.php** ajoutez le code suivant à la méthode `initialize()` :

```
$this->loadComponent('Authorization.Authorization');
```

Enfin, nous allons définir les actions add, login et logout comme ne nécessitant pas d'autorisation en ajoutant le code suivante à **src/Controller/UsersController.php** :

```
// Dans les méthodes add, login, et logout
$this->Authorization->skipAuthorization();
```

La méthode `skipAuthorization()` doit être appelée dans chaque controller ayant des actions accessibles à tous les utilisateurs, même ceux qui ne sont pas identifiés.

Créons notre Première Stratégie

Le plugin Authorization représente les autorisation et les permissions par des classes Policy. Ces classes contiennent la logique pour vérifier qu'une **identity** a la permission de **faire une action** sur une **ressource**. Notre **identity** sera un utilisateur authentifié, et nos **ressources** sont des entités de l'ORM ainsi que des requêtes sur la base de données. Utilisons **bake** pour créer la base de notre première stratégie.

```
bin/cake bake policy --type entity Article
```

Cette commande va générer une classe de Police vide pour notre entity `Article`. Vous retrouverez le fichier généré dans **src/Policy/ArticlePolicy.php**. Maintenant, modifiez la stratégie pour qu'elle ressemble à cela :

```
<?php
namespace App\Policy;

use App\Model\Entity\Article;
use Authorization\IdentityInterface;

class ArticlePolicy
{
    public function canAdd(IdentityInterface $user, Article $article)
    {
        //Tous les utilisateurs authentifiés peuvent créer des articles.
        return true;
    }

    public function canEdit(IdentityInterface $user, Article $article)
    {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        // Les utilisateurs authentifiés ne peuvent modifier que leurs articles.
        return $this->isAuthor($user, $article);
    }

    public function canDelete(IdentityInterface $user, Article $article)
    {
        // Les utilisateurs authentfiés ne peuvent supprimer que leurs articles.
        return $this->isAuthor($user, $article);
    }

    protected function isAuthor(IdentityInterface $user, Article $article)
    {
        return $article->user_id === $user->getIdentifier();
    }
}
```

Ici nous n'avons défini que quelques règles basiques, libre à vous d'utiliser des logiques plus complexes.

Utiliser Authorization dans ArticlesController

Maintenant que nos stratégies sont créées nous pouvons vérifier les autorisations dans chaque action de notre controller. Si nous oublions de vérifier les autorisations dans une action du controller, le plugin Authorization lèvera une exception. Dans **src/Controller/ArticlesController.php**, ajoutez le code suivant aux méthodes `add`, `edit` et `delete` :

```
public function add()
{
    $article = $this->Articles->newEmptyEntity();
    $this->Authorization->authorize($article);
    // Le reste de la méthode..
}

public function edit($slug)
{
    $article = $this->Articles
        ->findBySlug($slug)
        ->contain('Tags') // charge les Tags associés
        ->firstOrFail();
    $this->Authorization->authorize($article);
    // Le reste de la méthode..
}

public function delete($slug)
{
    $this->request->allowMethod(['post', 'delete']);

    $article = $this->Articles->findBySlug($slug)->firstOrFail();
    $this->Authorization->authorize($article);
    // Le reste de la méthode..
}
```

La méthode `AuthorizationComponent::authorize()` va utiliser le nom de l'action pour retrouver la méthode de la stratégie à appeler. Si vous préférez définir vous-même la méthode de la stratégie à utiliser vous devrez passer le nom de l'opération à `authorize` :

```
$this->Authorization->authorize($article, 'update');
```

Maintenant, ajoutez le code suivant aux méthodes `tags`, `view`, et `index` de votre `ArticlesController` :

```
// Les actions view, index et tags sont des méthodes accessibles
// à tous et ne nécessitent pas de vérifications.
$this->Authorization->skipAuthorization();
```

Amélioration des actions Add & Edit

Bien que nous ayons bloqué l'accès à l'édition, nous sommes toujours vulnérables au changement de l'attribut `user_id` de l'article par l'utilisateur durant l'édition. Nous allons remédier à cela. Commençons avec l'action `add`.

Lorsque nous créons des articles, nous voulons fixer le `user_id` comme étant l'utilisateur actuellement authentifié. Remplacez l'action `add` par le code suivant :

```
// Dans src/Controller/ArticlesController.php

public function add()
{
    $article = $this->Articles->newEmptyEntity();
    $this->Authorization->authorize($article);

    if ($this->request->is('post')) {
        $article = $this->Articles->patchEntity($article, $this->request->getData());

        // Changement: Chercher le user_id de l'utilisateur authentifié.
        $article->user_id = $this->request->getAttribute('identity')->getIdentifier();

        if ($this->Articles->save($article)) {
            $this->Flash->success(__('Votre article a été enregistré avec succès.'));

            return $this->redirect(['action' => 'index']);
        }
        $this->Flash->error(__('Impossible d\'ajouter votre article.'));
    }
    $tags = $this->Articles->Tags->find('list')->all();
    $this->set(compact('article', 'tags'));
}
```

Ensuite nous allons modifier l'action `edit`. Remplacez la méthode d'édition par ce qui suit :

```
// Dans src/Controller/ArticlesController.php

public function edit($slug)
{
    $article = $this->Articles
        ->findBySlug($slug)
        ->contain('Tags') // charge les Tags associés
        ->firstOrFail();
    $this->Authorization->authorize($article);

    if ($this->request->is(['post', 'put'])) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        $this->Articles->patchEntity($article, $this->request->getData(), [
            // Ajout: Empêcher la modification de user_id.
            'accessibleFields' => ['user_id' => false]
        ]);
        if ($this->Articles->save($article)) {
            $this->Flash->success(__('Votre article a été sauvegardé.'));

            return $this->redirect(['action' => 'index']);
        }
        $this->Flash->error(__('Impossible de mettre à jour votre article.'));
    }
    $tags = $this->Articles->Tags->find('list')->all();
    $this->set(compact('article', 'tags'));
}
```

Ici nous définissons les propriétés qui peuvent être assignées en masse en utilisant `patchEntity()`. Voir la section *Changer les Champs Accessibles* pour plus d'informations. N'oubliez pas d'enlever le contrôle du `user_id` dans **templates/Articles/edit.php**, nous n'en avons plus besoin.

Conclusion

Nous avons construit une application CMS basique qui permet à des utilisateurs de s'authentifier, d'écrire des articles, d'y ajouter des tags, de parcourir les articles rédigés, et avons mis en place des contrôles pour nos articles. Nous avons même ajouté des améliorations à l'interface en exploitant le FormHelper et les capacités de l'ORM

Merci d'avoir pris le temps d'explorer CakePHP. Nous vous proposons de continuer votre apprentissage avec *Accès Base de Données & ORM* ou de lire attentivement /topics.

Tutoriel de Bookmarker

Ce tutoriel va vous montrer la création d'une application simple de bookmarking (bookmarker). Pour commencer, nous allons installer CakePHP, créer notre base de données et utiliser les outils que CakePHP nous fournit pour créer une application rapidement.

Voici ce dont vous allez avoir besoin :

1. Un serveur de base de données. Nous allons utiliser un serveur MySQL dans ce tutoriel. Vous devrez en savoir assez sur SQL pour créer une base de données : CakePHP prendra les rênes à partir de là. Puisque nous utilisons MySQL, assurez-vous aussi d'avoir `pdo_mysql` activé dans PHP.
2. Des connaissances de base en PHP.

Avant de commencer, vous devez vous assurer que votre version de PHP est à jour :

```
php -v
```

Vous devez avoir installé au moins la version 8.1 (CLI) de PHP ou supérieure. La version de PHP de votre serveur web doit aussi être 8.1 ou supérieure, et doit être préférablement la même version que celle de votre interface en ligne de commande (CLI). Si vous souhaitez voir ce que donne l'application au final, regardez cakephp/bookmarker[48]. C'est parti !

48. https://github.com/cakephp/bookmarker-tutorial

Récupérer CakePHP

La façon la plus simple pour installer CakePHP est d'utiliser Composer. Composer est un moyen simple d'installer CakePHP depuis votre terminal ou votre prompteur de ligne de commandes. D'abord, vous aurez besoin de télécharger et d'installer Composer si vous ne l'avez pas déjà fait. Si vous avez cURL installé, exécutez ce qui suit :

```
curl -s https://getcomposer.org/installer | php
```

Ou alors vous pouvez télécharger `composer.phar` depuis le site de Composer[49].

Ensuite tapez simplement la ligne suivante dans votre terminal à partir du répertoire d'installation pour installer le squelette d'application CakePHP dans le répertoire **bookmarker** :

```
php composer.phar create-project --prefer-dist cakephp/app:4.* bookmarker
```

Si vous avez téléchargé et exécuté l'"installeur Windows de Composer[50], tapez la ligne suivante dans votre terminal à partir de votre répertoire d'installation. (par exemple C :\wamp\www\dev\cakephp3) :

```
composer self-update && composer create-project --prefer-dist cakephp/app:4.* bookmarker
```

L'avantage d'utiliser Composer est qu'il va automatiquement faire des tâches de configuration importantes, comme de définir les bonnes permissions de fichier et créer votre fichier **config/app.php** pour vous.

Il y a d'autres façons d'installer CakePHP. Si vous ne pouvez ou ne voulez pas utiliser Composer, consultez la section *Installation*.

Peu importe la façon dont vous avez téléchargé et installé CakePHP, une fois que votre configuration est faite, votre répertoire devrait ressembler à ce qui suit :

```
/bookmarker
    /bin
    /config
    /logs
    /plugins
    /src
    /tests
    /tmp
    /vendor
    /webroot
    .editorconfig
    .gitignore
    .htaccess
    .travis.yml
    composer.json
    index.php
    phpunit.xml.dist
    README.md
```

C'est le bon moment pour en apprendre un peu plus sur la façon dont la structure du répertoire de CakePHP fonctionne. Consultez la section *Structure du dossier de CakePHP*.

49. https://getcomposer.org/download/
50. https://getcomposer.org/Composer-Setup.exe

Vérifions notre Installation

Nous pouvons rapidement vérifier que notre installation fonctionne, en vérifiant la page d'accueil par défaut. Avant de faire ceci, vous devrez démarrer le serveur de développement :

```
bin/cake server
```

> **Note**
>
> Sur Windows, cette commande doit être `bin\cake server` (notez l'antislash).

Ceci va lancer le serveur web intégré de PHP sur le port 8765. Ouvrez **http ://localhost :8765** dans votre navigateur web pour voir la page d'accueil. Tous les points devront être cochés sauf pour CakePHP qui n'est pas encore capable de se connecter à votre base de données. Si ce n'est pas le cas, vous devrez installer des extensions PHP supplémentaires ou définir des permissions de répertoire.

Créer la Base de Données

Ensuite, configurons la base de données pour notre application de bookmarking. Si vous ne l'avez pas déjà fait, créez une base de données vide que nous allons utiliser dans ce tutoriel, avec un nom de votre choix, par exemple `cake_bookmarks`. Vous pouvez exécuter le SQL suivant pour créer les tables nécessaires :

```
CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    email VARCHAR(255) NOT NULL,
    password VARCHAR(255) NOT NULL,
    created DATETIME,
    modified DATETIME
);

CREATE TABLE bookmarks (
    id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT NOT NULL,
    title VARCHAR(50),
    description TEXT,
    url TEXT,
    created DATETIME,
    modified DATETIME,
    FOREIGN KEY user_key (user_id) REFERENCES users(id)
);

CREATE TABLE tags (
    id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(191),
    created DATETIME,
    modified DATETIME,
    UNIQUE KEY (title)
);

CREATE TABLE bookmarks_tags (
    bookmark_id INT NOT NULL,
    tag_id INT NOT NULL,
    PRIMARY KEY (bookmark_id, tag_id),
```

(suite sur la page suivante)

(suite de la page précédente)

```
    FOREIGN KEY tag_key(tag_id) REFERENCES tags(id),
    FOREIGN KEY bookmark_key(bookmark_id) REFERENCES bookmarks(id)
);
```

Vous avez peut-être remarqué que la table `bookmarks_tags` utilisait une clé primaire composite. CakePHP accepte les clés primaires composites presque partout, facilitant la construction des applications à tenant multiples.

La table et les noms de colonnes que nous avons utilisés n'étaient pas arbitraires. En utilisant les *conventions de nommage* de CakePHP, nous pouvons mieux contrôler CakePHP et éviter d'avoir à configurer le framework. CakePHP est assez flexible pour s'accommoder de tout schéma de base de données, mais suivre les conventions va vous faire gagner du temps.

Configuration de Base de Données

Ensuite, indiquons à CakePHP où se trouve notre base de données et comment s'y connecter. Pour la plupart d'entre vous, ce sera la première et la dernière fois que vous devrez configurer quelque chose.

La configuration est assez simple : remplacez juste les valeurs dans le tableau `Datasources.default` dans le fichier **config/app.php** avec ceux qui correspondent à votre configuration. Un exemple simple de tableau de configuration pourrait ressembler à ce qui suit :

```
return [
    // Plus de configuration au-dessus.
    'Datasources' => [
        'default' => [
            'className' => 'Cake\Database\Connection',
            'driver' => 'Cake\Database\Driver\Mysql',
            'persistent' => false,
            'host' => 'localhost',
            'username' => 'cakephp',
            'password' => 'AngelF00dC4k3~',
            'database' => 'cake_bookmarks',
            'encoding' => 'utf8',
            'timezone' => 'UTC',
            'cacheMetadata' => true,
        ],
    ],
    // Plus de configuration en dessous.
];
```

Une fois que vous avez sauvegardé votre fichier **config/app.php**, vous devriez voir la section "CakePHP est capable de se connecter à la base de données" cochée.

> **Note**
>
> Une copie du fichier de configuration par défaut de CakePHP se trouve dans **config/app.default.php**.

Génération de Code Scaffold

Comme notre base de données suit les conventions de CakePHP, nous pouvons utiliser l'application de console bake pour générer rapidement une application basique. Dans votre terminal, lancez les commandes suivantes :

```
// Sur Windows vous devez utiliser bin\cake à la place.
bin/cake bake all users
bin/cake bake all bookmarks
bin/cake bake all tags
```

Ceci va générer les controllers, models, views, leurs cas de tests correspondants et les fixtures pour nos ressources users, bookmarks et tags. Si vous avez stoppé votre serveur, relancez-le et allez sur **http ://localhost :8765/bookmarks**.

Vous devriez voir une application basique mais fonctionnelle fournissant des accès aux données vers les tables de la base de données de votre application. Une fois que vous avez la liste des bookmarks, ajoutez quelques users, bookmarks, et tags.

> **Note**
>
> Si vous avez une page Not Found (404), vérifiez que le module mod_rewrite d'Apache est chargé.

Ajouter un Hashage de Mot de Passe

Quand vous avez créé vos users, (en visitant **http ://localhost :8765/users**) vous avez probablement remarqué que les mots de passe sont stockés en clair. C'est très mauvais d'un point du vue sécurité, donc réglons ceci.

C'est aussi un bon moment pour parler de la couche model dans CakePHP. Dans CakePHP, nous séparons les méthodes qui agissent sur une collection d'objets, et celles qui agissent sur un objet unique, dans des classes différentes. Les méthodes qui agissent sur la collection des entities sont mises dans la classe `Table`, alors que les fonctionnalités correspondant à un enregistrement unique sont mises dans la classe `Entity`.

Par exemple, le hashage des mots de passe se fait pour un enregistrement individuel, donc nous allons intégrer ce comportement sur l'objet entity. Comme nous voulons hasher le mot de passe à chaque fois qu'il est défini nous allons utiliser une méthode mutateur/setter. CakePHP va appeler les méthodes setter basées sur les conventions à chaque fois qu'une propriété est définie dans une de vos entities. Ajoutons un setter pour le mot de passe. Dans **src/Model/Entity/User.php**, ajoutez ce qui suit :

```
namespace App\Model\Entity;

use Cake\Auth\DefaultPasswordHasher;
use Cake\ORM\Entity;

class User extends Entity
{

    // Code from bake.

    protected function _setPassword($value)
    {
        $hasher = new DefaultPasswordHasher();

        return $hasher->hash($value);
    }
}
```

Maintenant mettez à jour un des users que vous avez créé précédemment, si vous changez son mot de passe, vous devriez voir un mot de passe hashé à la place de la valeur originale sur la liste ou les pages de vue. CakePHP hashe les mots de passe avec bcrypt [51] par défaut. Vous pouvez aussi utiliser sha1 ou md5 si vous travaillez avec une base de

51. https://codahale.com/how-to-safely-store-a-password/

données existante.

> **Note**
>
> Si le mot de passe n'est pas haché, assurez-vous que vous avez suivi le même cas pour le mot de passe membre de la classe tout en nommant la fonction mutateur/setter

Récupérer les Bookmarks avec un Tag Spécifique

Maintenant que vous avez stocké les mots de passe de façon sécurisé, nous pouvons construire quelques fonctionnalités intéressantes dans notre application. Une fois que vous avez une collection de bookmarks, il peut être pratique de pouvoir les chercher par tag. Ensuite nous allons intégrer une route, une action de controller, et une méthode finder pour chercher les bookmarks par tag.

Idéalement, nous aurions une URL qui ressemble à **http ://localhost :8765/bookmarks/tagged/funny/cat/gifs** Cela nous aide à trouver tous les bookmarks qui ont les tags "funny", "cat" ou "gifs". Avant de pouvoir intégrer ceci, nous allons ajouter une nouvelle route. Votre fichier **config/routes.php** doit ressembler à ceci :

```
<?php
use Cake\Routing\Route\DashedRoute;
use Cake\Routing\Router;

Router::defaultRouteClass(DashedRoute::class);

// Nouvelle route ajoutée pour notre action "tagged".
// Le caractère `*` en fin de chaîne indique à CakePHP que cette action a
// des paramètres passés
Router::scope(
    '/bookmarks',
    ['controller' => 'Bookmarks'],
    function ($routes) {
        $routes->connect('/tagged/*', ['action' => 'tags']);
    }
);

Router::scope('/', function ($routes) {
    // Connecte la page d'accueil par défaut et les routes /pages/*.
    $routes->connect('/', [
        'controller' => 'Pages',
        'action' => 'display', 'home'
    ]);
    $routes->connect('/pages/*', [
        'controller' => 'Pages',
        'action' => 'display'
    ]);

    // Connecte les routes basées sur les conventions par défaut.
    $routes->fallbacks();
});
```

Ce qui est au-dessus définit une nouvelle "route" qui connecte le chemin **/bookmarks/tagged/***, vers `BookmarksController::tags()`. En définissant les routes, vous pouvez isoler la définition de vos URLs, de la façon dont elles sont intégrées. Si nous visitions **http ://localhost :8765/bookmarks/tagged**, nous verrions une page d'erreur de CakePHP. Intégrons maintenant la méthode manquante. Dans **src/Controller/BookmarksController.php**, ajoutez

ce qui suit :

```
public function tags()
{
    // La clé 'pass' est fournie par CakePHP et contient tous les segments
    // d'URL de la "request" (instance de \Cake\Network\Request)
    $tags = $this->request->getParam('pass');

    // On utilise l'objet "Bookmarks" (une instance de
    // \App\Model\Table\BookmarksTable) pour récupérer les bookmarks avec
    // ces tags
    $bookmarks = $this->Bookmarks->find('tagged', [
        'tags' => $tags
    ]);

    // Passe les variables au template de vue (view).
    $this->set([
        'bookmarks' => $bookmarks,
        'tags' => $tags
    ]);
}
```

Pour accéder aux autres parties des données de la « request », référez-vous à la section *ServerRequest*.

Créer la Méthode Finder

Dans CakePHP, nous aimons garder les actions de notre controller légères, et mettre la plupart de la logique de notre application dans les models. Si vous visitez l'URL **/bookmarks/tagged** maintenant, vous verrez une erreur comme quoi la méthode `findTagged()` n'a pas été encore intégrée, donc faisons-le. Dans **src/Model/Table/BookmarksTable.php** ajoutez ce qui suit :

```
// L'argument $query est une instance de \Cake\ORM\Query.
// Le tableau $options contiendra les tags que nous avons passé à find('tagged')
// dans l'action de notre Controller
public function findTagged(Query $query, array $options)
{
    $bookmarks = $this->find()
        ->select(['id', 'url', 'title', 'description']);

    if (empty($options['tags'])) {
        $bookmarks
            ->leftJoinWith('Tags')
            ->where(['Tags.title IS' => null]);
    } else {
        $bookmarks
            ->innerJoinWith('Tags')
            ->where(['Tags.title IN ' => $options['tags']]);
    }

    return $bookmarks->group(['Bookmarks.id']);
}
```

Nous intégrons juste *des finders personnalisés*. C'est un concept très puissant dans CakePHP qui vous permet de faire un package réutilisable de vos requêtes. Les finders attendent toujours un objet *Query Builder* et un tableau d'options en paramètre. Les finders peuvent manipuler les requêtes et ajouter n'importe quels conditions ou critères.

Une fois qu'ils ont terminé, les finders doivent retourner l'objet Query modifié. Dans notre finder nous avons amené les méthodes `innerJoinWith()`, `where()` et `group()` qui nous permet de trouver les bookmarks distinct qui ont un tag correspondante. Lorsque aucune tag n'est fournie, nous utilisons un `leftJoinWith()` et modifions la condition "where", en trouvant des bookmarks sans tags.

Créer la Vue

Maintenant si vous vous rendez à l'url **/bookmarks/tagged**, CakePHP va afficher une erreur vous disant que vous n'avez pas de fichier de vue. Construisons donc le fichier de vue pour notre action `tags()`. Dans **templates/Bookmarks/tags.php** mettez le contenu suivant :

```
<h1>
    Bookmarks tagged with
    <?= $this->Text->toList(h($tags)) ?>
</h1>

<section>
<?php foreach ($bookmarks as $bookmark): ?>
    <article>
        <!-- Utilise le HtmlHelper pour créer un lien -->
        <h4><?= $this->Html->link($bookmark->title, $bookmark->url) ?></h4>
        <small><?= h($bookmark->url) ?></small>

        <!-- Utilise le TextHelper pour formater le texte -->
        <?= $this->Text->autoParagraph(h($bookmark->description)) ?>
    </article>
<?php endforeach; ?>
</section>
```

Dans le code ci-dessus, nous utilisons le *Helper HTML* et le *Helper Text* pour aider à la génération du contenu de notre vue. Nous utilisons également la fonction *h* pour encoder la sortie en HTML. Vous devez vous rappeler de toujours utiliser `h()` lorsque vous affichez des données provenant des utilisateurs pour éviter les problèmes d'injection HTML.

Le fichier **tags.php** que nous venons de créer suit la convention de nommage de CakePHP pour un ficher de template de vue. La convention d'avoir le nom de template en minuscule et en underscore du nom de l'action du controller.

Vous avez peut-être remarqué que nous pouvions utiliser les variables `$tags` et `$bookmarks` dans notre vue. Quand nous utilisons la méthode `set()` dans notre controller, nous définissons les variables spécifiques à envoyer à la vue. La vue va rendre disponible toutes les variables passées dans les templates en variables locales.

Vous devriez maintenant pouvoir visiter l'URL **/bookmarks/tagged/funny** et voir tous les bookmarks taggés avec "funny".

Ainsi nous avons créé une application basique pour gérer des bookmarks, des tags et des users. Cependant, tout le monde peut voir tous les tags de tout le monde. Dans le prochain chapitre, nous allons intégrer une authentification et restreindre la visibilité des bookmarks à ceux qui appartiennent à l'utilisateur courant.

Maintenant continuons avec *Tutoriel de Bookmarker Part 2* pour construire votre application ou plongez dans la documentation pour en apprendre plus sur ce que CakePHP peut faire pour vous.

Tutoriel de Bookmarker Part 2

Après avoir fini *la première partie de ce tutoriel* vous devriez avoir une application basique de bookmarking. Dans ce chapitre, nous ajouterons l'authentification et nous allons restreindre les bookmarks pour que chaque utilisateur puisse voir/modifier seulement ceux qui lui appartiennent.

Ajouter la Connexion

Dans CakePHP, l'authentification est gérée par les *Components (Composants)*. Les components peuvent être imaginés comme des façons de créer des parties réutilisables de code du controller pour une fonctionnalité spécifique ou un concept. Les components peuvent aussi se lancer dans le cycle de vie de l'event du controller et interagir avec votre application de cette façon. Pour commencer, nous ajouterons *AuthComponent* à notre application. Nous voulons que chaque méthode nécessite l'authentification, donc nous allons ajouter AuthComponent dans notre AppController :

```
// Dans src/Controller/AppController.php
namespace App\Controller;

use Cake\Controller\Controller;

class AppController extends Controller
{
    public function initialize(): void
    {
        $this->loadComponent('Flash');
        $this->loadComponent('Auth', [
            'authenticate' => [
                'Form' => [
                    'fields' => [
                        'username' => 'email',
                        'password' => 'password'
                    ]
                ]
            ],
            'loginAction' => [
                'controller' => 'Users',
                'action' => 'login'
            ],
            // Si l'utilisateur arrive sur une page non-autorisée, on le
            // redirige sur la page précédente.
            'unauthorizedRedirect' => $this->referer()
        ]);

        // Autorise l'action display pour que notre controller de pages
        // continue de fonctionner.
        $this->Auth->allow(['display']);
    }
}
```

Nous avons seulement indiqué à CakePHP que nous souhaitions charger les components `Flash` et `Auth`. En plus, nous avons personnalisé la configuration de AuthComponent, puisque notre table users utilise `email` comme username. Maintenant, si vous tapez n'importe quelle URL, vous serez renvoyé vers **/users/login**, qui vous montrera une page d'erreur puisque nous n'avons pas encore écrit ce code. Créons donc l'action login :

```
// Dans src/Controller/UsersController.php

public function login()
{
    if ($this->request->is('post')) {
        $user = $this->Auth->identify();
        if ($user) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
            $this->Auth->setUser($user);

            return $this->redirect($this->Auth->redirectUrl());
        }
        $this->Flash->error('Votre username ou mot de passe est incorrect.');
    }
}
```

Et dans **templates/Users/login.php**, ajoutez ce qui suit :

```
<h1>Connexion</h1>
<?= $this->Form->create() ?>
<?= $this->Form->control('email') ?>
<?= $this->Form->control('password') ?>
<?= $this->Form->button('Login') ?>
<?= $this->Form->end() ?>
```

Maintenant que nous avons un formulaire simple de connexion, nous devrions pouvoir nous connecter avec un de nos utilisateurs qui a un mot de passe hashé.

> **Note**
>
> Si aucun de vos utilisateurs n'a de mot de passe hashé, commentez la ligne `loadComponent('Auth')`. Puis allez modifier l'utilisateur, créez- lui un nouveau mot de passe.

Ajouter la Déconnexion

Maintenant que les personnes peuvent se connecter, vous voudrez aussi probablement fournir un moyen de se déconnecter. Encore une fois, dans `UsersController`, ajoutez le code suivant :

```
public function initialize(): void
{
    parent::initialize();
    $this->Auth->allow(['logout']);
}

public function logout()
{
    $this->Flash->success('Vous êtes maintenant déconnecté.');

    return $this->redirect($this->Auth->logout());
}
```

Ce code autorise l'action `logout` en tant qu'action publique, et implémente la méthode logout. Vous pouvez maintenant visiter la page `/users/logout` pour vous déconnecter. Vous devriez alors être renvoyé vers la page de connexion.

Permettre de s'Enregistrer

Si vous n'êtes pas connecté et que vous essayez de visiter **/users/add** vous serez renvoyés vers la page de connexion. Nous devrions régler cela puisque nous voulons que les utilisateurs s'inscrivent à notre application. Dans `UsersController`, ajoutez ce qui suit :

```
public function initialize(): void
{
    parent::initialize();
    // Ajoute l'action 'add' à la liste des actions autorisées.
    $this->Auth->allow(['logout', 'add']);
}
```

Ce qui est au-dessus indique à `AuthComponent` que l'action `add()` *ne* nécessite *pas* d'authentification ou d'autorisation. Vous pouvez prendre le temps de nettoyer **Users/add.php** et de retirer les liens, ou continuez vers la prochaine section. Nous ne ferons pas de fichier d'édition (edit) ou de vue d'un utilisateur (view), ni de liste d'utilisateurs (index) dans ce tutoriel donc ils ne fonctionneront pas puisque `AuthComponent` va vous refuser l'accès pour ces actions de controller.

Restreindre l'Accès aux Bookmarks

Maintenant que les utilisateurs peuvent se connecter, nous voulons limiter les bookmarks qu'ils peuvent voir à ceux qu'ils ont créés. Nous allons le faire en utilisant un adaptateur "authorization". Puisque nos besoins sont assez simples, nous pouvons écrire quelques lignes de code simple dans notre `BookmarksController`. Mais avant de le faire, nous voulons dire à AuthComponent comment notre application va autoriser les actions. Dans notre `AppController`, ajoutez ce qui suit :

```
public function isAuthorized($user)
{
    return false;
}
```

Ajoutez aussi ce qui suit dans la configuration de `Auth` dans `AppController` :

```
'authorize' => 'Controller',
```

Votre méthode `initialize()` doit maintenant ressembler à ceci :

```
public function initialize(): void
{
    $this->loadComponent('Flash');
    $this->loadComponent('Auth', [
        'authorize'=> 'Controller',//added this line
        'authenticate' => [
            'Form' => [
                'fields' => [
                    'username' => 'email',
                    'password' => 'password'
                ]
            ]
        ],
        'loginAction' => [
            'controller' => 'Users',
            'action' => 'login'
        ],
        'unauthorizedRedirect' => $this->referer()
    ]);

    // Allow the display action so our pages controller
```

(suite sur la page suivante)

(suite de la page précédente)

```
    // continues to work.
    $this->Auth->allow(['display']);
}
```

Nous allons par défaut refuser l'accès, et permettre un accès incrémental où cela est utile. D'abord, nous allons ajouter la logique d'autorisation pour les bookmarks. Dans notre `BookmarksController`, ajoutez ce qui suit :

```
public function isAuthorized($user)
{
    $action = $this->request->params['action'];

    // Add et index sont toujours permises.
    if (in_array($action, ['index', 'add', 'tags'])) {
        return true;
    }
    // Tout autre action nécessite un id.
    if (!$this->request->getParam('pass.0')) {
        return false;
    }

    // Vérifie que le bookmark appartient à l'utilisateur courant.
    $id = $this->request->getParam('pass.0');
    $bookmark = $this->Bookmarks->get($id);
    if ($bookmark->user_id == $user['id']) {
        return true;
    }

    return parent::isAuthorized($user);
}
```

Maintenant, si vous essayez de voir, de modifier ou de supprimer un bookmark qui ne vous appartient pas, vous devriez être redirigé vers la page d'où vous venez. Si aucun message ne s'affiche, ajoutez la ligne suivante dans votre layout :

```
// Dans templates/layout/default.php
<?= $this->Flash->render() ?>
```

Vous devriez maintenant voir les messages d'erreur d'autorisation.

Régler la Vue de Liste et les Formulaires

Alors que view et delete fonctionnent, edit, add et index ont quelques problèmes :

1. Lors de l'ajout d'un bookmark, vous pouvez choisir l'utilisateur.
2. Lors de l'édition d'un bookmark vous pouvez choisir l'utilisateur.
3. La page de liste montre les bookmarks des autres utilisateurs.

Attaquons nous d'abord à add. Pour commencer, retirez `control('user_id')` de **templates/Bookmarks/add.php**. Une fois retiré, nous allons aussi mettre à jour l'action `add()` dans **src/Controller/BookmarksController.php** pour ressembler à ceci :

```
public function add()
{
    $bookmark = $this->Bookmarks->newEntity();
    if ($this->request->is('post')) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        $bookmark = $this->Bookmarks->patchEntity($bookmark, $this->request->getData());
        $bookmark->user_id = $this->Auth->user('id');
        if ($this->Bookmarks->save($bookmark)) {
            $this->Flash->success('Le bookmark a été sauvegardé.');

            return $this->redirect(['action' => 'index']);
        }
        $this->Flash->error('Le bookmark ne peut être sauvegardé. Merci de réessayer.');
    }
    $tags = $this->Bookmarks->Tags->find('list');
    $this->set(compact('bookmark', 'tags'));
    $this->viewBuilder()->setOption('serialize', ['bookmark']);
}
```

En définissant la propriété entity avec les données de session, nous retirons la possibilité que l'utilisateur puisse modifier l'auteur d'un bookmark. Nous ferons la même chose pour le formulaire et l'action edit. Votre action `edit()` dans **src/Controller/BookmarksController.php** devrait ressembler à ceci :

```
public function edit($id = null)
{
    $bookmark = $this->Bookmarks->get($id, [
        'contain' => ['Tags']
    ]);
    if ($this->request->is(['patch', 'post', 'put'])) {
        $bookmark = $this->Bookmarks->patchEntity($bookmark, $this->request->getData());
        $bookmark->user_id = $this->Auth->user('id');
        if ($this->Bookmarks->save($bookmark)) {
            $this->Flash->success('Le bookmark a été sauvegardé.');

            return $this->redirect(['action' => 'index']);
        } else {
            $this->Flash->error('Le bookmark ne peut être sauvegardé. Merci de réessayer.
↪');
        }
    }
    $tags = $this->Bookmarks->Tags->find('list');
    $this->set(compact('bookmark', 'tags'));
    $this->viewBuilder()->setOption('serialize', ['bookmark']);
}
```

Vue de Liste

Maintenant nous devons afficher les bookmarks pour l'utilisateur actuellement connecté. Nous pouvons le faire en mettant à jour l'appel à `paginate()`. Faites en sorte que votre action `index()` dans **src/Controller/BookmarksController.php** ressemble à ceci :

```
public function index()
{
    $this->paginate = [
        'conditions' => [
            'Bookmarks.user_id' => $this->Auth->user('id'),
        ]
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ];
    $this->set('bookmarks', $this->paginate($this->Bookmarks));
    $this->viewBuilder()->setOption('serialize', ['bookmarks']);
}
```

Nous devrions aussi mettre à jour l'action `tags()` et la méthode finder liée, mais nous vous laisserons ceci en exercice que vous pouvez faire vous-même.

Améliorer l'Experience de Tag

Actuellement, ajouter des nouveaux tags est un processus difficile, puisque `TagsController` interdit tous les accès. Plutôt que de permettre l'accès, nous pouvons améliorer l'UI de sélection de tag en utilisant un champ de texte séparé par des virgules. Cela donnera une meilleure expérience à nos utilisateurs, et utilisera quelques unes des super fonctionnalités de l'ORM.

Ajouter un Champ Computed

Comme nous voulons un accès simple vers les tags formatés pour une entity, nous pouvons ajouter un champ virtuel/calculé à l'entity. Dans **src/Model/Entity/Bookmark.php** ajoutez ce qui suit :

```
use Cake\Collection\Collection;

protected function _getTagString()
{
    if (isset($this->_fields['tag_string'])) {
        return $this->_fields['tag_string'];
    }
    if (empty($this->tags)) {
        return '';
    }
    $tags = new Collection($this->tags);
    $str = $tags->reduce(function ($string, $tag) {
        return $string . $tag->title . ', ';
    }, '');

    return trim($str, ', ');
}
```

Cela nous laissera l'accès à la propriété calculée `$bookmark->tag_string`. Nous utiliserons cette propriété dans controls plus tard. Rappelez-vous d'ajouter la propriété `tag_string` dans la liste `_accessible` de votre entity, puisque nous voulons la "sauvegarder" plus tard.

Dans le fichier **src/Model/Entity/Bookmark.php**, ajoutez `tag_string` à la propriété `_accessible` comme ceci :

```
protected array $_accessible = [
    'user_id' => true,
    'title' => true,
    'description' => true,
    'url' => true,
    'user' => true,
    'tags' => true,
    'tag_string' => true,
];
```

Mettre à Jour les Vues

Avec l'entity mise à jour, nous pouvons ajouter un nouveau *control* pour nos tags. Dans **templates/Bookmarks/add.php** et **templates/Bookmarks/edit.php**, remplacez l'input `tags._ids` existant avec ce qui suit :

```
echo $this->Form->control('tag_string', ['type' => 'text']);
```

Persister la Chaîne Tag

Maintenant que nous pouvons voir les tags existants en chaîne, nous voudrions aussi sauvegarder les données. Comme nous marquons les `tag_string` accessibles, l'ORM va copier ces données à partir de la requête dans notre entity. Nous pouvons utiliser une méthode hook `beforeSave()` pour parser la chaîne de tag et trouver/construire les entities liées. Ajoutez ce qui suit dans **src/Model/Table/BookmarksTable.php** :

```
public function beforeSave($event, $entity, $options)
{
    if ($entity->tag_string) {
        $entity->tags = $this->_buildTags($entity->tag_string);
    }
}

protected function _buildTags($tagString)
{
    // Trim tags
    $newTags = array_map('trim', explode(',', $tagString));
    // Retire tous les tags vides
    $newTags = array_filter($newTags);
    // Réduit les tags dupliqués
    $newTags = array_unique($newTags);

    $out = [];
    $query = $this->Tags->find()
        ->where(['Tags.title IN' => $newTags]);

    // Retire les tags existants de la liste des tags nouveaux.
    foreach ($query->extract('title') as $existing) {
        $index = array_search($existing, $newTags);
        if ($index !== false) {
            unset($newTags[$index]);
        }
    }
    // Ajoute les tags existants.
    foreach ($query as $tag) {
        $out[] = $tag;
    }
    // Ajoute les nouveaux tags.
    foreach ($newTags as $tag) {
        $out[] = $this->Tags->newEntity(['title' => $tag]);
    }

    return $out;
}
```

Alors que ce code est un peu plus compliqué que ce que nous avons déjà fait, il permet de montrer la puissance de l'ORM

de CakePHP. Vous pouvez facilement manipuler les résultats de requête en utilisant les méthodes des *Collections*, et gérer les scenarios où vous créez les entities à la volée avec facilité.

Récapitulatif

Nous avons élargi notre application de bookmarking pour gérer les scenarios de contrôle d'authentification et d'autorisation/d'accès basique. Nous avons aussi ajouté quelques améliorations UX en tirant parti du FormHelper et des capacités de l'ORM.

Merci d'avoir pris le temps d'explorer CakePHP. Ensuite, vous pouvez finir le tutoriel du *Tutoriel d'un Blog*, en apprendre plus sur l'*ORM* ou vous pouvez lire attentivement /topics.

Tutoriel d'un Blog

Ce tutoriel vous accompagnera à travers la création d'une simple application de blog. Nous récupérerons et installerons CakePHP, créerons et configurerons une base de données et ajouterons suffisamment de logique applicative pour lister, ajouter, éditer et supprimer des articles.

Voici ce dont vous aurez besoin :

1. Un serveur web fonctionnel. Nous supposerons que vous utilisez Apache, bien que les instructions pour utiliser d'autres serveurs doivent être assez semblables. Nous aurons peut-être besoin de jouer un peu sur la configuration du serveur, mais la plupart des personnes peuvent faire fonctionner CakePHP sans aucune configuration préalable. Assurez-vous d'avoir PHP 8.1 ou supérieur et que les extensions `mbstring` et `intl` sont activées dans PHP.
2. Un serveur de base de données. Dans ce tutoriel, nous utiliserons MySQL. Vous aurez besoin d'un minimum de connaissance en SQL afin de créer une base de données : CakePHP prendra les rênes à partir de là. Puisque nous utilisons MySQL, assurez-vous aussi que vous avez `pdo_mysql` activé dans PHP.
3. Des connaissances de base en PHP.

Maintenant, lançons-nous !

Obtenir CakePHP

Le manière la plus simple pour l'installer est d'utiliser Composer. Composer est une manière simple d'installer CakePHP à partir de votre terminal ou de l'invite de ligne de commande. Pour commencer, vous devrez télécharger et installer Composer si vous ne l'avez pas déjà. Si vous avez cURL, c'est aussi simple que de lancer la commande suivante :

```
curl -s https://getcomposer.org/installer | php
```

Ou vous pouvez télécharger `composer.phar` depuis le site de Composer[52].

Ensuite tapez simplement la ligne suivante dans votre terminal depuis votre répertoire d'installation pour installer le squelette d'application de CakePHP dans le répertoire où vous souhaitez l'utiliser. Pour l'exemple nous utiliserons « blog », mais vous pouvez utiliser le nom que vous souhaitez :

```
php composer.phar create-project --prefer-dist cakephp/app:4.* blog
```

Dans le cas où vous avez déjà composer installé globalement, vous devrez plutôt taper :

```
composer self-update && composer create-project --prefer-dist cakephp/app:4.* blog
```

L'avantage d'utiliser Composer est qu'il va automatiquement réaliser certaines tâches de configurations importantes, comme configurer les bonnes permissions de fichiers et créer votre fichier config/app.php à votre place.

52. https://getcomposer.org/download/

Il y a d'autres moyens d'installer CakePHP. Si vous ne pouvez pas ou ne voulez pas utiliser `Composer`, regardez la section *Installation*.

Quelle que soit la façon dont vous avez téléchargé et installé CakePHP, une fois la configuration terminée, votre répertoire d'installation devrait ressembler à quelque chose comme cela :

```
/cake_install
    /bin
    /config
    /logs
    /plugins
    /src
    /tests
    /tmp
    /vendor
    /webroot
    .editorconfig
    .gitignore
    .htaccess
    .travis.yml
    composer.json
    index.php
    phpunit.xml.dist
    README.md
```

A présent, il est peut-être temps de voir un peu comment fonctionne la structure de fichiers de CakePHP : lisez le chapitre *Structure du dossier de CakePHP*.

Les Permissions des Répertoires tmp et logs

Les répertoires `tmp` and `logs` doivent être accessibles en écriture pour le serveur web. Si vous avez utilisé Composer pour l'installation, ceci a du être fait pour vous et confirmé par un message « Permissions set on <répertoire> ». Si vous avez un message d'erreur à la place, ou si vous voulez le faire manuellement, la meilleure façon est de trouver sous quel utilisateur votre serveur web tourne en faisant (`<?= `whoami`; ?>`) et en attribuant la propriété du répertoire **src/tmp** à cet utilisateur. La commande finale que vous pouvez lancer (dans *nix) pourrait ressembler à ceci :

```
chown -R www-data tmp
chown -R www-data logs
```

Si pour une raison ou une autre, CakePHP ne peut écrire dans ce répertoire, vous en serez informé par un avertissement quand vous n'êtes pas en mode production.

Bien que non recommandé, si vous ne pouvez pas attribuer la propriété de ces répertoires à votre serveur web, vous pouvez simplement définir les permissions sur le dossier en lançant une commande comme celle-ci :

```
chmod -R 777 tmp
chmod -R 777 logs
```

Créer la Base de Données du Blog

Maintenant, mettons en place la base de données MySQL pour notre blog. Si vous ne l'avez pas déjà fait, créez une base de données vide avec le nom de votre choix pour l'utiliser dans ce tutoriel, par exemple `cake_blog`. Pour le moment, nous allons juste créer une seule table pour stocker nos articles.

```
# D'abord, créons la table des articles
CREATE TABLE articles (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(50),
    body TEXT,
    created DATETIME DEFAULT NULL,
    modified DATETIME DEFAULT NULL
);
```

Si vous utilisez PostgreSQL, connectez-vous à la base de données cake_blog et exécutez plutôt cette commande SQL :

```
-- D'abord, créons la table des articles
CREATE TABLE articles (
    id SERIAL PRIMARY KEY,
    title VARCHAR(50),
    body TEXT,
    created TIMESTAMP DEFAULT NULL,
    modified TIMESTAMP DEFAULT NULL
);
```

Nous allons aussi y placer quelques articles qui pourront être utilisés pour les tests. Exécutez les instructions SQL suivantes dans votre base de données (cela marche aussi bien pour MySQL que pour PostgreSQL) :

```
# Puis insérons quelques articles pour les tests
INSERT INTO articles (title,body,created)
    VALUES ('Le titre', 'Ceci est un contenu d\'article.', NOW());
INSERT INTO articles (title,body,created)
    VALUES ('Encore un titre', 'Et un autre contenu d\'article.', NOW());
INSERT INTO articles (title,body,created)
    VALUES ('Le retour du titre', 'C\'est vraiment excitant! Non.', NOW());
```

Le choix des noms de tables et de colonnes n'est pas arbitraire. Si vous respectez les conventions de nommage de CakePHP pour les bases de données et les classes (toutes deux expliquées au chapitre *Conventions de CakePHP*), vous tirerez profit d'un grand nombre de fonctionnalités automatiques et vous éviterez des étapes de configurations. CakePHP est suffisamment souple pour implémenter les pires schémas de bases de données, mais respecter les conventions vous fera gagner du temps.

Consultez le chapitre *Conventions de CakePHP* pour plus d'informations, mais il suffit de comprendre que nommer notre table "articles" permet de la relier automatiquement à notre model Articles, et qu'avoir des champs "modified" et "created" fait qu'ils seront gérés *automagiquement* par CakePHP.

Configurer la base de données

Ensuite, indiquons à CakePHP où se trouve notre base de données et comment s'y connecter. Pour la plupart d'entre vous, c'est la première et dernière fois que vous configurerez quelque chose.

Le fichier de configuration devrait être assez simple : remplacez simplement les valeurs du tableau `Datatsources.default` dans le fichier **config/app.php** avec ceux de votre config. Un exemple de tableau de configuration complet pourrait ressembler à ce qui suit :

```
return [
    // Plus de configuration au-dessus.
    'Datasources' => [
        'default' => [
```

(suite sur la page suivante)

(suite de la page précédente)

```
            'className' => 'Cake\Database\Connection',
            'driver' => 'Cake\Database\Driver\Mysql',
            'persistent' => false,
            'host' => 'localhost',
            'username' => 'cake_blog',
            'password' => 'AngelF00dC4k3~',
            'database' => 'cake_blog',
            'encoding' => 'utf8',
            'timezone' => 'UTC'
        ],
    ],
    // Plus de configuration ci-dessous.
];
```

Une fois votre fichier **config/app.php** sauvegardé, vous devriez être en mesure d'ouvrir votre navigateur internet et de voir la page d'accueil de CakePHP. Elle devrait également vous indiquer que votre fichier de connexion a été trouvé, et que CakePHP peut s'y connecter avec succès.

> **Note**
>
> Une copie du fichier de configuration par défaut de CakePHP se trouve dans **config/app.default.php**.

Configuration facultative

Il y a quelques autres éléments qui peuvent être configurés. La plupart des développeurs configurent les éléments de cette petite liste, mais ils ne sont pas obligatoires pour ce tutoriel. Le premier consiste à définir une chaîne de caractères personnalisée (ou « grain de sel ») afin de sécuriser les hashs.

Le « grain de sel » est utilisé pour générer des hashes. Si vous avez utilisé Composer, cela aussi a été pris en charge pendant l'installation. Sinon, changez sa valeur par défaut en modifiant **config/app.php**. La nouvelle valeur n'a pas beaucoup d'importance du moment qu'elle est difficile à deviner :

```
'Security' => [
    'salt' => 'quelque chose de long et qui contienne plein de valeurs différentes.',
],
```

Une note sur mod_rewrite

Occasionnellement, les nouveaux utilisateurs peuvent avoir des problèmes de mod_rewrite. Par exemple si la page d'accueil de CakePHP a l'air bizarre (pas d'images ou de styles CSS), cela signifie probablement que mod_rewrite ne fonctionne pas sur votre système. Merci de consulter la section *Réécriture d'URL* pour résoudre le problème.

Maintenant continuez vers *Tutoriel d'un Blog - Partie 2* pour commencer à construire votre première application CakePHP.

Tutoriel d'un Blog - Partie 2

Créer un Model Article

Les Models sont le pain quotidien des applications CakePHP. En créant un model CakePHP qui interagira avec notre base de données, nous aurons mis en place les fondations nécessaires pour faire plus tard nos opérations de lecture, d'insertion, d'édition et de suppression.

Les fichiers des classes de model de CakePHP sont séparés entre des objets `Table` et `Entity`. Les objets `Table` fournissent un accès à la collection des entities stockées dans une table spécifique et vont dans **src/Model/Table**. Le fichier que nous allons créer sera sauvegardé dans **src/Model/Table/ArticlesTable.php**. Le fichier complété devrait ressembler à ceci :

```
// src/Model/Table/ArticlesTable.php

namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{
    public function initialize(array $config): void
    {
        $this->addBehavior('Timestamp');
    }
}
```

La convention de nommage est vraiment très importante dans CakePHP. En nommant notre objet Table `ArticlesTable`, CakePHP va automatiquement supposer que cet objet Table sera utilisé dans le `ArticlesController`, et sera lié à une table de la base de données appelée `articles`.

> **Note**
>
> CakePHP créera dynamiquement un objet model pour vous, s'il ne trouve pas le fichier correspondant dans **src/Model/Table**. Cela veut aussi dire que si vous n'avez pas nommé correctement votre fichier (par ex. articles-table.php ou ArticleTable.php). CakePHP ne reconnaîtra pas votre configuration et utilisera les objets par défaut.

Pour plus d'informations sur les models, comme les callbacks et la validation, consultez le chapitre *Accès Base de Données & ORM* du manuel.

> **Note**
>
> Si vous avez terminé la *Partie 1 du Tutoriel du blog* et créé la table `articles` dans notre base de données Blog, vous pouvez utiliser la console bake de CakePHP et sa fonctionnalité de génération de code pour créer le model `ArticlesTable` :
>
> ```
> bin/cake bake model Articles
> ```

Pour plus d'informations sur bake et les fonctionnalités de génération de code, vous pouvez allez voir /bake/usage.

Créer le controller Articles

Nous allons maintenant créer un controller pour nos articles. Le controller est l'endroit où vont se faire toutes les interactions avec les articles. En un mot, c'est l'endroit où vous jouerez avec les models et où vous ferez les tâches liées aux articles. Nous placerons ce nouveau controller dans un fichier appelé **ArticlesController.php** à l'intérieur du dossier **src/Controller**. Voici à quoi devrait ressembler le controller de base :

```
// src/Controller/ArticlesController.php

namespace App\Controller;
```

(suite sur la page suivante)

(suite de la page précédente)

```
class ArticlesController extends AppController
{
}
```

Maintenant, ajoutons une action à notre controller. Les actions représentent souvent une simple fonction ou une interface dans une application. Par exemple, quand les utilisateurs requêtent www.exemple.com/articles/index (qui est également la même chose que www.exemple.com/articles/), ils pourraient s'attendre à voir une liste d'articles. Le code pour cette action devrait ressembler à quelque chose comme ça :

```
// src/Controller/ArticlesController.php

namespace App\Controller;

class ArticlesController extends AppController
{

    public function index()
    {
        $articles = $this->Articles->find()->all();
        $this->set(compact('articles'));
    }
}
```

En définissant la fonction `index()` dans notre `ArticlesController`, les utilisateurs peuvent maintenant accéder à cette logique en demandant www.exemple.com/articles/index. De la même façon, si nous devions définir une fonction appelée `foobar()`, les utilisateurs pourrait y accéder en demandant www.exemple.com/articles/foobar.

Avertissement

Vous pourriez être tenté de nommer vos controllers et vos actions d'une certaine manière pour obtenir une certaine URL. Résistez à cette tentation. Suivez les *Conventions de CakePHP* de CakePHP (le nom des controllers au pluriel, etc.) et nommez vos actions de façon lisible et compréhensible. Vous pouvez lier les URLs à votre code en utilisant ce qu'on appelle le *Routing*, on le verra plus tard.

Dans cet action, la seule instruction utilise `set()` pour transmettre les données du controller à la vue (que nous créerons à la prochaine étape). La méthode `find()` de l'objet `ArticlesTable` renvoie une instance de `Cake\\ORM\\Query` et appelle sa méthode `all()`, qui renvoie une instance de `Cake\\Collection\\CollectionInterface`, qui est affecté dans une variable de la vue appelée "articles".

Note

Si vous avez terminé la *Partie 1 du Tutoriel du blog* et créé la table `articles` dans notre base de données Blog, vous pouvez utiliser la console bake de CakePHP et sa fonctionnalité de génération de code pour créer la classe ArticlesController :

```
bin/cake bake controller Articles
```

Pour plus d'informations sur bake et les fonctionnalités de génération de code, vous pouvez allez voir /bake/usage.

Pour en apprendre plus sur les controllers de CakePHP, consultez le chapitre *Controllers (Contrôleurs)*.

Créer les Vues des Articles

Maintenant que nous avons nos données en provenance du model, ainsi que la logique applicative définie par notre controller, nous allons créer une vue pour l'action `index` que nous avons créée ci-dessus.

Les vues de CakePHP sont juste des fragments de présentation, « assaisonnés », qui s'intègrent au sein du layout de l'application. Pour la plupart des applications, elles sont un mélange de HTML et PHP, mais les vues peuvent aussi être constituées de XML, CSV ou même de données binaires.

Un Layout est un code de présentation qui entoure une vue. Vous pouvez en définir plusieurs et passer de l'un à l'autre, mais pour le moment, utilisons juste celui par défaut.

Vous souvenez-vous, dans la dernière section, comment nous avions assigné la variable "articles" à la vue en utilisant la méthode `set()` ? Cela transmettrait l'objet query à la vue, pour qu'elle puisse ensuite le parcourir avec `foreach`.

Les fichiers de template de CakePHP sont stockés dans **templates**, à l'intérieur d'un dossier dont le nom correspond à celui du controller (nous aurons à créer un dossier appelé "Articles" dans ce cas). Pour mettre en forme les données de ces articles dans un joli tableau, le code de notre vue devrait ressembler à quelque chose comme cela :

```
<!-- Fichier: templates/Articles/index.php -->

<h1>Tous les articles du Blog</h1>
<table>
    <tr>
        <th>Id</th>
        <th>Titre</th>
        <th>Créé le</th>
    </tr>

    <!-- Ici se trouve l'itération sur l'objet query de nos $articles, et l'affichage␣
↪des infos des articles -->

    <?php foreach ($articles as $article): ?>
    <tr>
        <td><?= $article->id ?></td>
        <td>
            <?= $this->Html->link($article->title, ['action' => 'view', $article->id]) ?>
        </td>
        <td>
            <?= $article->created->format(DATE_RFC850) ?>
        </td>
    </tr>
    <?php endforeach; ?>
</table>
```

Espérons que cela vous semble simple.

Vous avez sans doute remarqué l'utilisation d'un objet appelé `$this->Html`. C'est une instance de la classe CakePHP *`Cake\View\Helper\HtmlHelper`*. CakePHP est livré avec un ensemble d'assistants (*helpers*) pour les vues, qui réalisent en un clin d'œil des choses comme le « linking » (mettre les liens dans un texte), l'affichage des formulaires, du JavaScript et de l'AJAX. Vous pouvez en apprendre plus sur la manière de les utiliser dans le chapitre *Helpers (Assistants)*, mais ce qu'il est important de noter ici, c'est que la méthode `link()` générera un lien HTML à partir d'un titre (le premier paramètre) et d'une URL (le second paramètre).

Lorsque vous indiquez des URLs dans CakePHP, il est recommandé d'utiliser les tableaux. La raison est expliquée en détail dans le chapitre des Routes. L'utilisation de tableaux dans les URLs vous permet de tirer profit des capacités de CakePHP à ré-inverser les routes. Vous pouvez aussi utiliser des URLs relatives à la base de l'application sous la forme `/controller/action/param1/param2` ou utiliser les *routes nommées*.

À ce stade, vous devriez être en mesure de pointer votre navigateur sur la page http://www.exemple.com/articles/index. Vous devriez voir votre vue, correctement formatée avec le titre et le tableau listant les articles.

Si vous avez essayé de cliquer sur l'un des liens que nous avons créés dans cette vue (qui lient le titre d'un article à l'URL `/articles/view/un_id_quelconque`), vous avez sûrement été informé par CakePHP que l'action n'a pas encore été définie. S'il ne vous en a pas informé, soit quelque chose s'est mal passé, soit en fait vous aviez déjà défini l'action, auquel cas vous êtes vraiment sournois ! Sinon, nous allons la créer sans plus tarder dans le Controller Articles :

```
// src/Controller/ArticlesController.php

namespace App\Controller;

class ArticlesController extends AppController
{

    public function index()
    {
        $this->set('articles', $this->Articles->find()->all());
    }

    public function view($id = null)
    {
        $article = $this->Articles->get($id);
        $this->set(compact('article'));
    }
}
```

L'appel de `set()` devrait vous être familier. Notez que nous utilisons `get()` plutôt que `find()` parce que nous voulons récupérer les informations d'un seul article.

Notez que notre action « view » prend un paramètre : l'ID de l'article que nous souhaitons consulter. Ce paramètre est transmis à l'action grâce à l'URL. Si un utilisateur demande `/articles/view/3`, alors la valeur "3" est transmise à la variable `$id`.

Nous faisons aussi une petite vérification d'erreurs pour nous assurer qu'un utilisateur accède bien à l'enregistrement. En utilisant la fonction `get()` dans la table Articles, nous nous assurons que l'utilisateur a accès à un enregistrement qui existe effectivement. Dans le cas où l'article requêté n'est pas présent dans la base de données, ou si l'id est incorrect, la fonction `get()` va lancer une `NotFoundException`.

Maintenant, créons la vue pour notre nouvelle action "view" et plaçons-la dans **templates/Articles/view.php**.

```
<!-- Fichier: templates/Articles/view.php -->

<h1><?= h($article->title) ?></h1>
<p><?= h($article->body) ?></p>
<p><small>Créé: <?= $article->created->format(DATE_RFC850) ?></small></p>
```

Vérifiez que cela fonctionne en testant les liens de la page `/articles/index` ou en affichant manuellement un article via `/articles/view/{id}`, en remplaçant {id} par un "id" d'article.

Ajouter des Articles

Lire depuis la base de données et nous afficher les articles est un bon début, mais lançons-nous dans l'ajout de nouveaux articles.

Premièrement, commençons par créer une action `add()` dans le `ArticlesController` :

```php
// src/Controller/ArticlesController.php

namespace App\Controller;

use App\Controller\AppController;

class ArticlesController extends AppController
{
    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('Flash'); // Charge le FlashComponent
    }

    public function index()
    {
        $this->set('articles', $this->Articles->find()->all());
    }

    public function view($id)
    {
        $article = $this->Articles->get($id);
        $this->set(compact('article'));
    }

    public function add()
    {
        $article = $this->Articles->newEmptyEntity();
        if ($this->request->is('post')) {
            $article = $this->Articles->patchEntity($article, $this->request->getData());
            if ($this->Articles->save($article)) {
                $this->Flash->success(__('Votre article a été sauvegardé.'));

                return $this->redirect(['action' => 'index']);
            }
            $this->Flash->error(__('Impossible d\'ajouter votre article.'));
        }
        $this->set('article', $article);
    }
}
```

Note

Vous avez besoin de charger le component *Flash* dans chaque controller où vous voudrez l'utiliser. Si nécessaire, chargez-le dans le controller principal (`AppController`).

Voici ce que fait l'action `add()` : si la requête HTTP est de type POST, elle essaye de sauvegarder les données en utilisant le model « Articles ». Si pour une raison quelconque la sauvegarde a échoué, elle affiche simplement la vue. Cela nous donne la possibilité de montrer à l'utilisateur les erreurs de validation ou d'autres avertissements.

Chaque requête de CakePHP contient un objet `ServerRequest` qui est accessible en utilisant `$this->request`. Cet objet contient des informations utiles sur la requête qui vient d'être reçue, et permet de contrôler le flux de votre

application. Dans ce cas, nous utilisons la méthode *Cake\Http\ServerRequest::is()* pour vérifier que la requête est de type POST.

Lorsqu'un utilisateur utilise un formulaire pour poster des données dans votre application, ces informations sont disponibles dans `$this->request->getData()`. Vous pouvez utiliser les fonctions *pr()* ou *debug()* pour les afficher si vous voulez voir à quoi cela ressemble.

Nous utilisons les méthodes `success()` et `error()` de FlashComponent pour définir un message dans une variable de session. Ces méthodes sont fournies via la méthode magique _call() [53] de PHP. Les messages Flash seront affichés dans la page juste après la redirection. Dans le layout, nous avons `<?= $this->Flash->render() ?>` qui permet d'afficher le message et d'effacer la variable correspondante. La méthode *Cake\Controller\Controller::redirect* du controller permet de rediriger vers une autre URL. Le paramètre `['action' => 'index']` sera traduit vers l'URL /articles, c'est à dire l'action « index » du controller Articles (ArticlesController). Vous pouvez vous référer à l" API [54] de la fonction `Cake\Routing\Router::url()` pour voir les différents formats d'URL acceptés dans les différentes fonctions de CakePHP.

L'appel de la méthode `save()` vérifiera les règles de validation et interrompra l'enregistrement si une erreur survient. Nous verrons la façon dont les erreurs sont traitées dans les sections suivantes.

Valider les Données

Cake place la barre très haute pour briser la monotonie de la validation des champs de formulaires. Tout le monde déteste le développement de formulaires interminables et leurs routines de validations. Cake rend tout cela plus facile et plus rapide.

Pour tirer profit des fonctionnalités de validation, vous devez utiliser le helper *Form* (FormHelper) dans vos vues. La classe *Cake\View\Helper\FormHelper* est disponible par défaut dans toutes les vues avec la variable `$this->Form`.

Voici le code de notre vue **add** :

```
<!-- Fichier: templates/Articles/add.php -->

<h1>Ajouter un article</h1>
<?php
    echo $this->Form->create($article);
    echo $this->Form->control('title');
    echo $this->Form->control('body', ['rows' => '3']);
    echo $this->Form->button(__("Sauvegarder l\'article"));
    echo $this->Form->end();
?>;
```

Nous utilisons le FormHelper pour générer la balise d'ouverture d'un formulaire HTML. Voici le code HTML généré par `$this->Form->create()` :

```
<form method="post" action="/articles/add">
```

Si `create()` est appelée sans aucun paramètre, CakePHP suppose que vous construisez un formulaire qui envoie les données en POST à l'action `add()` (ou `edit()` quand `id` est dans les données du formulaire) du controller actuel.

La méthode `$this->Form->control()` est utilisée pour créer des éléments de formulaire du même nom. Le premier paramètre dit à CakePHP à quel champ ils correspondent et le second paramètre vous permet de spécifier un large éventail d'options - dans notre cas, le nombre de lignes du textarea. Il y a un peu d'introspection et « d'automagie » ici : `control()` affichera différents éléments de formulaire selon le champ spécifié du model.

L'appel de la méthode `$this->Form->end()` clôture le formulaire. Elle crée les champs cachés si la protection de falsification de formulaire et/ou CSRF est activée.

53. https://php.net/manual/fr/language.oop5.overloading.php#object.call
54. https://api.cakephp.org

À présent, revenons en arrière et modifions notre vue **templates/Articles/index.php** pour ajouter un lien « Ajouter un article ». Ajoutez la ligne suivante avant `<table>` :

```
<?= $this->Html->link('Ajouter un article', ['action' => 'add']) ?>
```

Vous vous demandez peut-être : comment je fais pour indiquer à CakePHP mes exigences de validation ? Les règles de validation sont définies dans le model. Retournons donc à notre model Articles et procédons à quelques ajustements :

```
// src/Model/Table/ArticlesTable.php

namespace App\Model\Table;

use Cake\ORM\Table;
use Cake\Validation\Validator;

class ArticlesTable extends Table
{
    public function initialize(array $config): void
    {
        $this->addBehavior('Timestamp');
    }

    public function validationDefault(Validator $validator): Validator
    {
        $validator
            ->notEmptyString('title')
            ->requirePresence('title', 'create')
            ->notEmptyString('body')
            ->requirePresence('body', 'create');

        return $validator;
    }
}
```

Le méthode `validationDefault()` indique à CakePHP comment valider vos données lorsque la méthode `save()` est appelée. Ici, nous avons spécifié que les deux champs « body » et « title » ne doivent pas être vides et que ces champs sont requis à la fois pour les opérations de création et de mise à jour. Le moteur de validation de CakePHP est puissant, il dispose d'un certain nombre de règles intégrées (code de carte bancaire, adresse emails, etc.) et d'une souplesse pour ajouter vos propres règles de validation. Pour plus d'informations sur cette configuration, consultez le chapitre *Validation*.

Maintenant que vos règles de validation sont en place, utilisez l'application pour essayer d'ajouter un article avec un titre et un contenu vides afin de voir si cela fonctionne. Puisque que nous avons utilisé la méthode *`Cake\View\Helper\FormHelper::control()`* du helper « Form » pour créer nos éléments de formulaire, nos messages d'erreurs de validation seront affichés automatiquement.

Éditer des Articles

L'édition d'articles : nous y voilà ! Vous êtes un pro de CakePHP maintenant, vous devriez donc avoir adopté le principe. Créez d'abord l'action, puis la vue. Voici à quoi l'action `edit()` du controller Articles (`ArticlesController`) devrait ressembler :

```
// src/Controller/ArticlesController.php
```

(suite sur la page suivante)

(suite de la page précédente)

```
public function edit($id = null)
{
    $article = $this->Articles->get($id);
    if ($this->request->is(['post', 'put'])) {
        $this->Articles->patchEntity($article, $this->request->getData());
        if ($this->Articles->save($article)) {
            $this->Flash->success(__('Votre article a été mis à jour.'));

            return $this->redirect(['action' => 'index']);
        }
        $this->Flash->error(__('Impossible de mettre à jour votre article.'));
    }

    $this->set('article', $article);
}
```

Cette action s'assure d'abord que l'utilisateur a essayé d'accéder à un enregistrement existant. S'il n'y a pas de paramètre `$id` passé, ou si le article n'existe pas, nous lançons une `NotFoundException` pour que le gestionnaire d'Erreurs ErrorHandler de CakePHP s'en occupe.

Ensuite l'action vérifie si la requête est une requête POST ou PUT. Si elle l'est, alors nous utilisons les données POST pour mettre à jour notre entity article en utilisant la méthode `patchEntity()`. Finalement nous utilisons l'objet table pour sauvegarder l'entity en retour, ou sinon rejeter les données et montrer les erreurs de validation de l'utilisateur.

La vue **edit** devrait ressembler à quelque chose comme cela :

```
<!-- Fichier: templates/Articles/edit.php -->

<h1>Modifier un article</h1>
<?php
    echo $this->Form->create($article);
    echo $this->Form->control('title');
    echo $this->Form->control('body', ['rows' => '3']);
    echo $this->Form->button(__('Sauvegarder l\'article'));
    echo $this->Form->end();
?>
```

Cette vue affiche le formulaire d'édition (avec les données pré-remplies) avec les messages d'erreur de validation nécessaires.

CakePHP déterminera si un `save()` doit générer une insertion d'un article ou la mise à jour d'un article existant en fonction de l'état de l'entity.

Vous pouvez maintenant mettre à jour votre vue **index** avec des liens pour éditer chaque article :

```
<!-- Fichier: templates/Articles/index.php  (liens de modification ajoutés) -->

<h1>Blog articles</h1>
<p><?= $this->Html->link("Ajouter un Article", ['action' => 'add']) ?></p>
<table>
    <tr>
        <th>Id</th>
        <th>Title</th>
        <th>Created</th>
```

(suite sur la page suivante)

(suite de la page précédente)

```
        <th>Action</th>
    </tr>

<!-- C'est ici que nous itérons à travers notre objet query $articles, -->
<!-- en affichant les informations de l'article -->

<?php foreach ($articles as $article): ?>
    <tr>
        <td><?= $article->id ?></td>
        <td>
            <?= $this->Html->link($article->title, ['action' => 'view', $article->id]) ?>
        </td>
        <td>
            <?= $article->created->format(DATE_RFC850) ?>
        </td>
        <td>
            <?= $this->Html->link('Modifier', ['action' => 'edit', $article->id]) ?>
        </td>
    </tr>
<?php endforeach; ?>

</table>
```

Supprimer des Articles

À présent, mettons en place un moyen pour les utilisateurs de supprimer les articles. Commencez par une action `delete()` dans le controller Articles (`ArticlesController`) :

```
// src/Controller/ArticlesController.php

public function delete($id)
{
    $this->request->allowMethod(['post', 'delete']);

    $article = $this->Articles->get($id);
    if ($this->Articles->delete($article)) {
        $this->Flash->success(__("L'article avec l'id: {0} a été supprimé.", h($id)));

        return $this->redirect(['action' => 'index']);
    }
}
```

Cette logique supprime l'article spécifié par `$id`, et utilise `$this->Flash->success()` pour afficher à l'utilisateur un message de confirmation après l'avoir redirigé sur `/articles`. Si l'utilisateur tente une suppression en utilisant une requête GET, la méthode `allowMethod()` lève une exception. Les exceptions manquées sont capturées par le gestionnaire d'exceptions de CakePHP et un joli message d'erreur est affiché. Il y a plusieurs *Exceptions* intégrées qui peuvent être utilisées pour indiquer les différentes erreurs HTTP que votre application pourrait rencontrer.

Étant donné que nous exécutons juste un peu de logique et de redirection, cette action n'a pas de vue. Vous voudrez peut-être néanmoins mettre à jour votre vue **index** avec des liens pour permettre aux utilisateurs de supprimer des articles :

```
<!-- Fichier: templates/Articles/index.php -->

<h1>Blog articles</h1>
<p><?= $this->Html->link('Ajouter un Article', ['action' => 'add']) ?></p>
<table>
    <tr>
        <th>Id</th>
        <th>Title</th>
        <th>Created</th>
        <th>Actions</th>
    </tr>

<!-- C'est ici que nous itérons à travers notre objet query $articles, -->
<!-- en affichant les informations de l'article -->

    <?php foreach ($articles as $article): ?>
    <tr>
        <td><?= $article->id ?></td>
        <td>
            <?= $this->Html->link($article->title, ['action' => 'view', $article->id]) ?>
        </td>
        <td>
            <?= $article->created->format(DATE_RFC850) ?>
        </td>
        <td>
            <?= $this->Form->postLink(
                'Supprimer',
                ['action' => 'delete', $article->id],
                ['confirm' => 'Êtes-vous sûr ?'])
            ?>
            <?= $this->Html->link('Modifier', ['action' => 'edit', $article->id]) ?>
        </td>
    </tr>
    <?php endforeach; ?>

</table>
```

Utiliser `postLink()` permet de créer un lien qui utilise du JavaScript pour supprimer notre article en faisant une requête POST.

Avertissement

Autoriser la suppression par une requête GET est dangereux à cause des robots d'indexation qui peuvent supprimer accidentellement toutes vos données.

Note

Ce code de vue utilise aussi le helper `FormHelper` pour demander à l'utilisateur une confirmation en JavaScript avant de supprimer un article.

Routes

Pour certains, le routage par défaut de CakePHP fonctionne suffisamment bien. Les développeurs qui sont sensibles à la lisibilité pour l'utilisateur et à la compatibilité avec les moteurs de recherches apprécieront la manière dont CakePHP lie des URLs à des actions spécifiques. Nous allons donc faire une petite modification des routes dans ce tutoriel.

Pour plus d'informations sur les techniques de routage avancées, consultez le chapitre *Connecter les Routes*.

Par défaut, CakePHP effectue une redirection d'une personne visitant la racine de votre site (par ex : http://www.exemple.com) vers le controller Pages (`PagesController`) et affiche le rendu de la vue appelée **home**. Au lieu de cela, nous voudrions la remplacer avec notre controller Articles (`ArticlesController`) en créant une règle de routage.

Le routage de CakePHP se trouve dans **config/routes.php**. Vous devrez commenter ou supprimer la ligne qui définit la route par défaut. Elle ressemble à cela :

```
$builder->connect('/', ['controller' => 'Pages', 'action' => 'display', 'home']);
```

Cette ligne connecte l'URL "/" à la page d'accueil par défaut de CakePHP. Nous voulons que cette URL soit connectée à notre propre controller, remplacez donc la ligne par celle-ci :

```
$builder->connect('/', ['controller' => 'Articles', 'action' => 'index']);
```

Cela devrait connecter les utilisateurs demandant "/" à l'action `index()` de notre controller Articles (`ArticlesController`).

> **Note**
>
> CakePHP peut aussi faire du routage inversé (*reverse routing*). Si, avec la route définie plus haut, vous passez `['controller' => 'Articles', 'action' => 'index']` à une fonction qui attend un tableau, l'URL générée sera "/". Il est d'ailleurs bien avisé de toujours utiliser un tableau pour les URLs car cela signifie que vos routes définissent où vont les URLs, et ainsi cela vous assure qu'elles pointent toujours le même endroit.

Conclusion

Gardez à l'esprit que ce tutoriel était très basique. CakePHP a *beaucoup* plus de fonctionnalités à offrir et il est aussi souple dans d'autres domaines que nous n'avons pas souhaité couvrir ici pour simplifier les choses. Utilisez le reste de ce manuel comme un guide pour développer des applications plus riches en fonctionnalités.

Maintenant que vous avez créé une application CakePHP basique, vous pouvez soit continuer vers *Tutoriel d'un Blog - Partie 3*, ou commencer votre propre projet. Vous pouvez aussi lire attentivement les /topics ou l"API [55] pour en apprendre plus sur CakePHP.

Si vous avez besoin d'aide, il y a plusieurs façons d'obtenir de l'aide - merci de regarder la page *Où obtenir de l'aide* Bienvenue sur CakePHP !

Prochaines lectures suggérées

Voici les différents chapitres que les gens veulent souvent lire après :

1. *Layouts* : Personnaliser les layouts de votre application.
2. *Elements* : Inclure et réutiliser des portions de vues.
3. /bake/usage Générer un code CRUD basique.
4. *Tutoriel d'un Blog - Authentification* : Tutoriel sur l'enregistrement et la connexion d'utilisateurs.

55. https://api.cakephp.org

Tutoriel d'un Blog - Partie 3

Créer une Catégorie en Arbre (Tree)

Continuons notre application de blog et imaginons que nous souhaitions catégoriser nos articles. Nous souhaitons que les catégories soit triées, et pour cela, nous allons utiliser le *behavior Tree* pour nous aider à organiser les catégories.

Mais d'abord, nous devons modifier nos tables.

Plugin Migrations

Nous allons utiliser le plugin migrations [56] pour créer une table dans notre base de données. Si vous avez déjà une table articles dans votre base de données, supprimez-la.

Maintenant ouvrez le fichier **composer.json** de votre application. Normalement vous devriez voir que le plugin migrations est déjà dans `require`. Si ce n'est pas le cas, ajoutez-le en utilisant :

```
composer require cakephp/migrations:~1.0
```

Le plugin migrations va maintenant être dans le dossier **plugins** de votre application. Ajoutez aussi `Plugin::load('Migrations');` à la méthode `bootstrap` de votre application.

Une fois que le plugin est chargé, lancez la commande suivante pour créer un fichier de migration :

```
bin/cake bake migration CreateArticles title:string body:text category_id:integer␣
↪created modified
```

Un fichier de migration sera généré dans le dossier **config/Migrations** avec ce qui suit :

```
<?php

use Migrations\AbstractMigration;

class CreateArticles extends AbstractMigration
{
    public function change()
    {
        $table = $this->table('articles');
        $table->addColumn('title', 'string', [
            'default' => null,
            'limit' => 255,
            'null' => false,
        ]);
        $table->addColumn('body', 'text', [
            'default' => null,
            'null' => false,
        ]);
        $table->addColumn('category_id', 'integer', [
            'default' => null,
            'limit' => 11,
            'null' => false,
        ]);
        $table->addColumn('created', 'datetime', [
            'default' => null,
```

(suite sur la page suivante)

56. https://github.com/cakephp/migrations

(suite de la page précédente)

```
            'null' => false,
        ]);
        $table->addColumn('modified', 'datetime', [
            'default' => null,
            'null' => false,
        ]);
        $table->create();
    }
}
```

Exécutez une autre commande pour créer une table `categories`. Si vous voulez spécifier une longueur de champ, vous pouvez le faire entre crochets dans le type du champ, par exemple :

```
bin/cake bake migration CreateCategories parent_id:integer lft:integer[10]␣
↪rght:integer[10] name:string[100] description:string created modified
```

Ceci va générer le fichier suivant dans **config/Migrations** :

```
<?php

use Migrations\AbstractMigration;

class CreateCategories extends AbstractMigration
{
    public function change()
    {
        $table = $this->table('categories');
        $table->addColumn('parent_id', 'integer', [
            'default' => null,
            'limit' => 11,
            'null' => false,
        ]);
        $table->addColumn('lft', 'integer', [
            'default' => null,
            'limit' => 10,
            'null' => false,
        ]);
        $table->addColumn('rght', 'integer', [
            'default' => null,
            'limit' => 10,
            'null' => false,
        ]);
        $table->addColumn('name', 'string', [
            'default' => null,
            'limit' => 100,
            'null' => false,
        ]);
        $table->addColumn('description', 'string', [
            'default' => null,
            'limit' => 255,
            'null' => false,
        ]);
```

(suite sur la page suivante)

(suite de la page précédente)

```
        $table->addColumn('created', 'datetime', [
            'default' => null,
            'null' => false,
        ]);
        $table->addColumn('modified', 'datetime', [
            'default' => null,
            'null' => false,
        ]);
        $table->create();
    }
}
```

Maintenant que les fichiers de migration sont créés, vous pouvez les modifier avant de créer vos tables. Nous devons changer `'null' => false` pour le champ `parent_id` par `'null' => true` car une catégorie de premier niveau a un `parent_id` null.

Exécutez la commande suivante pour créer vos tables :

```
bin/cake migrations migrate
```

Modifier les Tables

Avec nos tables définies, nous pouvons maintenant nous focaliser sur la catégorisation de nos articles.

Nous supposons que vous avez déjà les fichiers (Tables, Controllers et Templates des Articles) de la partie 2. Donc nous allons juste ajouter les références aux catégories.

Nous devons associer ensemble les tables Articles et Categories. Ouvrez le fichier **src/Model/Table/ArticlesTable.php** et ajoutez ce qui suit :

```
// src/Model/Table/ArticlesTable.php

namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{
    public function initialize(array $config): void
    {
        $this->addBehavior('Timestamp');
        // Ajoute juste la relation belongsTo avec CategoriesTable
        $this->belongsTo('Categories', [
            'foreignKey' => 'category_id',
        ]);
    }
}
```

Générer les Squelettes de Code des Catégories

Créez tous les fichiers en lançant les commandes **bake** suivantes :

```
bin/cake bake model Categories
bin/cake bake controller Categories
bin/cake bake template Categories
```

Au choix, vous pouvez aussi tous les créer en une seule ligne :

```
bin/cake bake all Categories
```

L'outil bake a créé tous les fichiers en un clin d'œil. Vous pouvez les lire rapidement si vous voulez vous re-familiariser avec le fonctionnement de CakePHP.

> **Note**
>
> Si vous utilisez Windows, pensez à utiliser \ à la place de /.

Vous devrez modifier ce qui suit dans **templates/Categories/add.php** et **templates/Categories/edit.php** :

```
echo $this->Form->control('parent_id', [
    'options' => $parentCategories,
    'empty' => 'Pas de catégorie parente'
]);
```

Attacher TreeBehavior à CategoriesTable

Le *TreeBehavior* vous aide à gérer des structures hiérarchiques en arbre dans une table de base de données. Il utilise la logique MPTT [57] pour gérer les données. Les structures en arbre MPTT sont optimisées pour lire des données, ce qui les rend souvent pratiques pour lire des applications lourdes comme les blogs.

Si vous ouvrez le fichier **src/Model/Table/CategoriesTable.php**, vous verrez que le TreeBehavior a été attaché à votre CategoriesTable dans la méthode `initialize()`. Bake ajoute automatiquement ce behavior à toutes les Tables qui contiennent les colonnes `lft` et `rght` :

```
$this->addBehavior('Tree');
```

Avec le TreeBehavior attaché, vous serez capable d'accéder à certaines fonctionnalités comme la réorganisation de l'ordre des categories. Nous verrons cela dans un moment.

Mais pour l'instant, vous devez retirer les lignes suivantes dans vos templates **add** et **edit** des catégories :

```
echo $this->Form->control('lft');
echo $this->Form->control('rght');
```

De plus, vous devez désactiver ou retirer les requirePresence du validateur pour les colonnes `lft` et `rght` dans votre model CategoriesTable :

```
public function validationDefault(Validator $validator): Validator
{
    $validator
        ->add('id', 'valid', ['rule' => 'numeric'])
        ->allowEmptyString('id', 'create');

    $validator
```

(suite sur la page suivante)

57. https://www.sitepoint.com/hierarchical-data-database-2/

(suite de la page précédente)

```
        ->add('lft', 'valid', ['rule' => 'numeric'])
    //    ->requirePresence('lft', 'create')
        ->notEmpty('lft');

    $validator
        ->add('rght', 'valid', ['rule' => 'numeric'])
    //    ->requirePresence('rght', 'create')
        ->notEmpty('rght');
}
```

Ces champs sont gérés automatiquement par le TreeBehavior quand une catégorie est sauvegardée.

En utilisant votre navigateur, ajoutez quelques nouvelles catégories en utilisant l'action `/yoursite/categories/add`.

Réorganiser l'Ordre des Catégories avec le TreeBehavior

Dans votre fichier de template **index** des catégories, vous pouvez lister les catégories et les réordonner.

Modifions la méthode index dans votre **CategoriesController.php** et ajoutons les méthodes `moveUp()` et `moveDown()` pour pouvoir réorganiser l'ordre des catégories dans l'arbre :

```
class CategoriesController extends AppController
{
    public function index()
    {
        $categories = $this->Categories->find()
            ->order(['lft' => 'ASC'])
            ->all();
        $this->set(compact('categories'));
        $this->viewBuilder()->setOption('serialize', ['categories']);
    }

    public function moveUp($id = null)
    {
        $this->request->allowMethod(['post', 'put']);
        $category = $this->Categories->get($id);
        if ($this->Categories->moveUp($category)) {
            $this->Flash->success('La catégorie a été remontée.');
        } else {
            $this->Flash->error("La catégorie n'a pas pu être remontée. Veuillez␣
↪réessayer.");
        }

        return $this->redirect($this->referer(['action' => 'index']));
    }

    public function moveDown($id = null)
    {
        $this->request->allowMethod(['post', 'put']);
        $category = $this->Categories->get($id);
        if ($this->Categories->moveDown($category)) {
            $this->Flash->success('La catégorie a été descendue.');
        } else {
```

(suite sur la page suivante)

(suite de la page précédente)

```
            $this->Flash->error("La catégorie n'a pas pu être descendue. Veuillez␣
↪réessayer.");
        }

        return $this->redirect($this->referer(['action' => 'index']));
    }
}
```

Remplacez le contenu existant dans **templates/Categories/index.php** par ceci :

```
<div class="actions large-2 medium-3 columns">
    <h3><?= __('Actions') ?></h3>
    <ul class="side-nav">
        <li><?= $this->Html->link(__('Nouvelle Categorie'), ['action' => 'add']) ?></li>
    </ul>
</div>
<div class="categories index large-10 medium-9 columns">
    <table cellpadding="0" cellspacing="0">
    <thead>
        <tr>
            <th>Id</th>
            <th>Id parent</th>
            <th>Gauche</th>
            <th>Droite</th>
            <th>Nom</th>
            <th>Description</th>
            <th>Créée le</th>
            <th class="actions">><?= __('Actions') ?></th>
        </tr>
    </thead>
    <tbody>
    <?php foreach ($categories as $category): ?>
        <tr>
            <td><?= $category->id ?></td>
            <td><?= $category->parent_id ?></td>
            <td><?= $category->lft ?></td>
            <td><?= $category->rght ?></td>
            <td><?= h($category->name) ?></td>
            <td><?= h($category->description) ?></td>
            <td><?= h($category->created) ?></td>
            <td class="actions">
                <?= $this->Html->link(__('Voir'), ['action' => 'view', $category->id]) ?>
                <?= $this->Html->link(__('Editer'), ['action' => 'edit', $category->id])␣
↪?>
                <?= $this->Form->postLink(__('Supprimer'), ['action' => 'delete',
↪$category->id], ['confirm' => __('Etes vous sur de vouloir supprimer # {0}?',
↪$category->id)]) ?>
                <?= $this->Form->postLink(__('Descendre'), ['action' => 'moveDown',
↪$category->id], ['confirm' => __('Etes vous sur de vouloir descendre # {0}?',
↪$category->id)]) ?>
                <?= $this->Form->postLink(__('Monter'), ['action' => 'moveUp', $category-
↪>id], ['confirm' => __('Etes vous sur de vouloir monter # {0}?', $category->id)]) ?>
```

(suite sur la page suivante)

(suite de la page précédente)

```
            </td>
        </tr>
    <?php endforeach; ?>
    </tbody>
    </table>
</div>
```

Modifier ArticlesController

Dans notre `ArticlesController`, nous allons récupérer la liste de toutes les catégories. Ceci va nous permettre de choisir une catégorie pour un Article lorsqu'on va le créer ou le modifier :

```
// src/Controller/ArticlesController.php

namespace App\Controller;

use Cake\Http\Exception\NotFoundException;

class ArticlesController extends AppController
{

    // ...

    public function add()
    {
        $article = $this->Articles->newEmptyEntity();
        if ($this->request->is('post')) {
            $article = $this->Articles->patchEntity($article, $this->request->getData());
            if ($this->Articles->save($article)) {
                $this->Flash->success(__('Votre article a été enregistré.'));

                return $this->redirect(['action' => 'index']);
            }
            $this->Flash->error(__("Impossible d\'ajouter votre article."));
        }
        $this->set('article', $article);

        // Ajout de la liste des catégories pour pouvoir choisir
        // une catégorie pour un article
        $categories = $this->Articles->Categories->find('treeList');
        $this->set(compact('categories'));
    }
}
```

Modifier les Templates des Articles

Le fichier **add** des articles devrait ressembler à ceci :

```
<!-- Fichier: templates/Articles/add.php -->

<h1>Ajouter un Article</h1>
<?php
```

(suite sur la page suivante)

(suite de la page précédente)

```
echo $this->Form->create($article);
// Ajout des input liés aux catégories (via la méthode "control")
echo $this->Form->control('category_id');
echo $this->Form->control('title');
echo $this->Form->control('body', ['rows' => '3']);
echo $this->Form->button(__("Enregistrer l'article"));
echo $this->Form->end();
```

Quand vous allez à l'adresse `/yoursite/categories/add`, vous devriez voir une liste de choix des catégories.

Tutoriel d'un Blog - Authentification

Poursuivant notre exemple *Tutoriel d'un Blog*, imaginons que nous souhaitions interdire aux utilisateurs non connectés de créer des articles.

Créer la Table et le Controller pour Users

Premièrement, créons une nouvelle table dans notre base de données blog pour enregistrer les données de nos utilisateurs :

```
CREATE TABLE users (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    email VARCHAR(255),
    password VARCHAR(255),
    role VARCHAR(20),
    created DATETIME DEFAULT NULL,
    modified DATETIME DEFAULT NULL
);
```

Si vous utilisez PostgreSQL, connectez-vous à la base de données cake_blog et exécutez plutôt la commande SQL suivante :

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    email VARCHAR(255),
    password VARCHAR(255),
    role VARCHAR(20),
    created TIMESTAMP DEFAULT NULL,
    modified TIMESTAMP DEFAULT NULL
);
```

Nous avons respecté les conventions de CakePHP pour le nommage des tables, mais nous profitons d'une autre convention : en utilisant les colonnes `email` et `password` dans une table `users`, CakePHP sera capable de configurer automatiquement la plupart des choses pour nous quand nous réaliserons la connexion de l'utilisateur.

La prochaine étape est de créer notre classe `UsersTable`, qui a la responsabilité de trouver, sauvegarder et valider toute donnée d'utilisateur :

```
// src/Model/Table/UsersTable.php
namespace App\Model\Table;

use Cake\ORM\Table;
use Cake\Validation\Validator;
```

(suite sur la page suivante)

(suite de la page précédente)

```
class UsersTable extends Table
{

    public function validationDefault(Validator $validator): Validator
    {
        return $validator
            ->notEmpty('email', "Un email est nécessaire")
            ->email('email')
            ->notEmpty('password', 'Un mot de passe est nécessaire')
            ->notEmpty('role', 'Un rôle est nécessaire')
            ->add('role', 'inList', [
                'rule' => ['inList', ['admin', 'author']],
                'message' => 'Merci d\'entrer un rôle valide'
            ]);
    }

}
```

Créons aussi notre UsersController. Le contenu suivant correspond à la classe obtenue grâce à l'utilitaire de génération de code fourni par CakePHP :

```
// src/Controller/UsersController.php

namespace App\Controller;

use App\Controller\AppController;
use Cake\Event\EventInterface;

class UsersController extends AppController
{

    public function index()
    {
        $this->set('users', $this->Users->find()->all());
    }

    public function view($id)
    {
        $user = $this->Users->get($id);
        $this->set(compact('user'));
    }

    public function add()
    {
        $user = $this->Users->newEmptyEntity();
        if ($this->request->is('post')) {
            $user = $this->Users->patchEntity($user, $this->request->getData());
            if ($this->Users->save($user)) {
                $this->Flash->success(__("L'utilisateur a été sauvegardé."));

                return $this->redirect(['action' => 'add']);
```

(suite sur la page suivante)

(suite de la page précédente)

```
            }
            $this->Flash->error(__("Impossible d'ajouter l\'utilisateur."));
        }
        $this->set('user', $user);
    }

}
```

De la même façon que nous avons créé les vues pour nos articles en utilisant l'outil de génération de code, nous pouvons implémenter les vues des utilisateurs. Dans le cadre de ce tutoriel, nous allons juste montrer le **add.php** :

```
<!-- templates/Users/add.php -->

<div class="users form">
<?= $this->Form->create($user) ?>
    <fieldset>
        <legend><?= __('Ajouter un utilisateur') ?></legend>
        <?= $this->Form->control('email') ?>
        <?= $this->Form->control('password') ?>
        <?= $this->Form->control('role', [
            'options' => ['admin' => 'Admin', 'author' => 'Author']
        ]) ?>
    </fieldset>
<?= $this->Form->button(__('Ajouter')); ?>
<?= $this->Form->end() ?>
</div>
```

Authentification (Connexion et Déconnexion)

Nous sommes maintenant prêts à ajouter notre couche d'authentification. Dans CakePHP, cette couche est gérée par le plugin `authentication`. Commençons par l'installer. Utilisez composer pour l'installation du plugin :

```
composer require "cakephp/authentication:^2.0"
```

Puis ajoutez le code suivant à la méthode `bootstrap()` de votre application :

```
// dans la méthode bootstrap() de src/Application.php
$this->addPlugin('Authentication');
```

Hachage des Mots de Passe

Ensuite, nous allons créer l'entité `User` et ajouter un hachage de mots de passe. Créez le fichier d'entité **src/Model/Entity/User.php** et ajoutez ce qui suit :

```
// src/Model/Entity/User.php
namespace App\Model\Entity;

use Cake\Auth\DefaultPasswordHasher;
use Cake\ORM\Entity;

class User extends Entity
{
```

(suite sur la page suivante)

(suite de la page précédente)

```
    // Rend les champs assignables en masse sauf pour la clé primaire "id".
    protected array $_accessible = [
        '*' => true,
        'id' => false
    ];

    // ...

    protected function _setPassword($password)
    {
        if (strlen($password) > 0) {
            return (new DefaultPasswordHasher)->hash($password);
        }
    }

    // ...
}
```

Maintenant, à chaque fois qu'un mot de passe est assigné à l'entité utilisateur, il est haché en utilisant la classe `DefaultPasswordHasher`.

Configurer l'Authentification

Il est maintenant temps de configurer le Plugin Authentication. Le Plugin va gérer le processus d'identification en utilisant 3 classes différentes :

- `Application` utilisera le Middleware Authentication et fournira un AuthenticationService. Il comportera toute la configuration que nous voulons pour définir comment nous allons vérifier les identifiants fournis, et où nous allons trouver les informations avec lesquelles les comparer.
- `AuthenticationService` sera une classe utilitaire pour vous permettre de configurer le processus d'authentification.
- `AuthenticationMiddleware` sera exécuté comme une étape de la middleware queue. Il s'exécute avant que vos contrôleurs soient appelés par le framework, et va chercher les identifiants ou preuves de connexion pour vérifier si l'utilisateur est connecté.

La logique d'authentification est divisée en classes spécifiques et le processus d'authentification se met en route avant la couche de vos contrôleurs. En tout premier, l'authentification cherche à authentifier l'utilisateur (selon la configuration que vous aurez définie) puis injecte l'utilisateur et les résultats d'authentification dans la requête, pour qu'ils soient consultables par la suite.

Dans **src/Application.php**, ajoutez les imports suivants :

```
// Dans src/Application.php ajoutez les imports suivants
use Authentication\AuthenticationService;
use Authentication\AuthenticationServiceInterface;
use Authentication\AuthenticationServiceProviderInterface;
use Authentication\Middleware\AuthenticationMiddleware;
use Psr\Http\Message\ServerRequestInterface;
```

Puis implémentez l'interface d'authentification dans votre classe Application :

```
// dans src/Application.php
class Application extends BaseApplication
```

(suite sur la page suivante)

(suite de la page précédente)

```
    implements AuthenticationServiceProviderInterface
{
```

Et ajoutez ce qui suit :

```
// src/Application.php
public function middleware(MiddlewareQueue $middlewareQueue): MiddlewareQueue
    $middlewareQueue
        // ... autres middlewares ajoutés auparavant
        ->add(new RoutingMiddleware($this))
        // ajoutez Authentication après RoutingMiddleware
        ->add(new AuthenticationMiddleware($this));

    return $middlewareQueue;

public function getAuthenticationService(ServerRequestInterface $request):␣
↪AuthenticationServiceInterface
{
    $authenticationService = new AuthenticationService([
        'unauthenticatedRedirect' => '/users/login',
        'queryParam' => 'redirect',
    ]);

    // Charger les identificateurs. S'assurer que nous vérifions les champs email et␣
↪password
    $authenticationService->loadIdentifier('Authentication.Password', [
        'fields' => [
            'username' => 'email',
            'password' => 'password',
        ]
    ]);

    // Charger les authentificateurs. En général vous voudrez mettre Session en premier.
    $authenticationService->loadAuthenticator('Authentication.Session');
    // Configurer la connexion par formulaire pour qu'elle aille chercher
    // les champs email et password.
    $authenticationService->loadAuthenticator('Authentication.Form', [
        'fields' => [
            'username' => 'email',
            'password' => 'password',
        ],
        'loginUrl' => '/users/login',
    ]);

    return $authenticationService;
}
```

Dans votre classe `AppController`, ajoutez ce code :

```
public function initialize(): void
{
    parent::initialize();
```

(suite sur la page suivante)

(suite de la page précédente)

```
    $this->loadComponent('RequestHandler');
    $this->loadComponent('Flash');

    // AJoutez cette ligne pour vérifier le résultat de l'authentification
    // et donc verrouiller l'accès à votre site.
    $this->loadComponent('Authentication.Authentication');
```

Maintenant, à chaque requête, l'`AuthenticationMiddleware` va examiner la session de la requête pour y rechercher un utilisateur authentifié. Si nous sommes en train de charger la page `/users/login`, il va aussi inspecter les données envoyées par formulaire (s'il y en a) pour en extraire les identifiants utilisateur. Par défaut, les identifiants seront extraits des champs `email` et `password` dans les données de la requête. Le résultat de l'authentification sera injecté dans un attribut de la requête nommé `authentication`. Vous pouvez consulter le résultat à n'importe quel moment en utilisant `$this->request->getAttribute('authentication')` depuis les actions de vos contrôleurs. Toutes vos pages auront un accès restreint puisque l'`AuthenticationComponent` vérifie le résultat à chaque requête. Lorsqu'il échouera à trouver un utilisateur authentifié, il redirigera l'utilisateur vers la page `/users/login`. Veuillez noter qu'à ce stade, le site ne fonctionnera pas puisque nous n'avons pas encore de page de connexion. Si vous visitez le site, vous obtiendrez une « boucle infinie de redirections ». Alors, corrigeons ça !

Dans votre `UsersController`, ajoutez ce code :

```
public function beforeFilter(\Cake\Event\EventInterface $event)
{
    parent::beforeFilter($event);
    // Configurer l'action login pour ne pas exiger d'authentification, et
    // ainsi empêcher un problème de boucle infinie de redirections
    $this->Authentication->addUnauthenticatedActions(['login']);
}

public function login()
{
    $this->request->allowMethod(['get', 'post']);
    $result = $this->Authentication->getResult();
    // Qu'on soit en POST ou en GET, rediriger l'utilisateur s'il est déjà connecté
    if ($result->isValid()) {
        // rediriger vers /articles après une connexion réussie
        $redirect = $this->request->getQuery('redirect', [
            'controller' => 'Articles',
            'action' => 'index',
        ]);

        return $this->redirect($redirect);
    }
    // afficher une erreur si l'utilisateur a validé le formulaire mais que
    // l'authentification a échoué
    if ($this->request->is('post') && !$result->isValid()) {
        $this->Flash->error(__('Invalid email or password'));
    }
}
```

Ajoutez la logique du template pour votre action login :

```
<!-- dans /templates/Users/login.php -->
<div class="users form">
```

(suite sur la page suivante)

(suite de la page précédente)

```
    <?= $this->Flash->render() ?>
    <h3>Login</h3>
    <?= $this->Form->create() ?>
    <fieldset>
        <legend><?= __('Merci d\'entrer vos nom d'utilisateur et mot de passe') ?></
↪legend>
        <?= $this->Form->control('email', ['required' => true]) ?>
        <?= $this->Form->control('password', ['required' => true]) ?>
    </fieldset>
    <?= $this->Form->submit(__('Se Connecter')); ?>
    <?= $this->Form->end() ?>

    <?= $this->Html->link("Ajouter un utilisateur", ['action' => 'add']) ?>
</div>
```

À présent, la page de connexion va nous permettre de nous connecter correctement dans notre application. Testez-le en essayant d'accéder à une page quelconque de votre site. Après avoir été redirigé vers la page `/users/login`, entrez l'e-mail et le mot de passe que vous aviez choisis précédemment quand vous avez créé l'utilisateur. Vous devriez être connecté sans problème et redirigé vers la bonne page.

Nous avons encore besoin de quelques détails pour configurer notre application. Nous voulons que toutes les pages `view` et `index` soient accessibles sans avoir à se connecter, donc nous allons ajouter cette configuration spécifique dans `AppController` :

```
// dans src/Controller/AppController.php
public function beforeFilter(\Cake\Event\EventInterface $event)
{
    parent::beforeFilter($event);
    // pour tous les contrôleurs de notre application, rendre les actions
    // index et viex publiques en sautant l'étape d'authentification.
    $this->Authentication->addUnauthenticatedActions(['index', 'view']);
```

Déconnexion

Ajoutez l'action logout à votre classe `UsersController` :

```
// dans src/Controller/UsersController.php
public function logout()
{
    $result = $this->Authentication->getResult();
    // Qu'on soit en POST ou en GET, rediriger l'utilisateur s'il est déjà connecté
    if ($result->isValid()) {
        $this->Authentication->logout();

        return $this->redirect(['controller' => 'Users', 'action' => 'login']);
    }
}
```

À présent vous pouvez visiter l'URL `/users/logout` pour vous déconnecter. Vous devriez alors être renvoyé vers la page de connexion. Si vous êtes arrivés à ce point, félicitations, vous avez maintenant un blog simple qui :

- Autorise les utilisateurs connectés à créer et éditer des articles.
- Autorise les utilisateurs non connectés à consulter des articles et des tags.

Lectures suivantes suggérées

1. /bake/usage Génération basique CRUD de code
2. Documentation de Authentication Plugin.

CHAPITRE 5

Contribuer

Il y a plusieurs façons de contribuer à CakePHP. Les sections suivantes couvrent les différentes manières de contribuer à CakePHP :

Documentation

Contribuer à la documentation est simple. Les fichiers sont hébergés sur https://github.com/cakephp/docs. N'hésitez pas à forker le dépôt, ajoutez vos changements/améliorations/traductions et retournez les avec un pull request. Vous pouvez même modifier les documents en ligne avec GitHub, sans télécharger les fichiers – le bouton « Improve this Doc » (Améliorer cette Doc) sur toutes les pages vous redirigera vers l'éditeur en ligne de Github pour la page correspondante.

La documentation de CakePHP est intégrée de façon continue [58], et déployée après chaque pull request est fusionnée.

Traductions

Envoyez un Email à l'équipe docs (docs at cakephp dot org) ou venez discuter sur IRC (#cakephp on freenode) de tout effort de traduction auquel vous souhaitez participer.

Nouvelle Traduction d'une Langue

Nous souhaitons créer des traductions aussi complètes que possible. Cependant, il peut arriver des fois où un fichier de traduction n'est pas à jour. Vous devriez toujours considérer la version anglais comme la version qui fait autorité.

Si votre langue n'est pas dans les langues actuellement proposées, merci de nous contacter sur Github et nous envisagerons de créer un squelette de dossier pour cette langue. Les sections suivantes sont les premières par lesquelles vous devriez commencer puisque ce sont des fichiers qui ne changent pas souvent :

- index.rst
- intro.rst
- quickstart.rst

58. https://en.wikipedia.org/wiki/Continuous_integration

— installation.rst
— dossier /intro
— dossier /tutorials-and-examples

Note pour les Administrateurs de la Doc

La structure de tous les dossiers de langue doivent refléter la structure du dossier anglais. Si la structure change pour la version anglaise, nous devrions appliquer ces changements dans les autres langues.

Par exemple, si un nouveau fichier anglais est créé dans **en/file.rst**, nous devrions :

— Ajouter le fichier dans les autres langues : **fr/file.rst**, **zh/file.rst**, ...
— Supprimer le contenu, mais en gardant les `title`, informations `meta` et d'éventuels éléments `toc-tree`. La note suivante sera ajoutée en anglais tant que personne n'a transmis le fichier :

```
File Title
##########

.. note::
    The documentation is not currently supported in XX language for this
    page.

    Please feel free to send us a pull request on
    `Github <https://github.com/cakephp/docs>`_ or use the **Improve This Doc**
    button to directly propose your changes.

    You can refer to the English version in the select top menu to have
    information about this page's topic.

// If toc-tree elements are in the English version
.. toctree::
    :maxdepth: 1

    one-toc-file
    other-toc-file

.. meta::
    :title lang=xx: File Title
    :keywords lang=xx: title, description,...
```

Astuces de traducteurs

— Parcourez et modifiez le contenu à traduire dans le langage voulu - sinon vous ne verrez pas ce qui a déjà été traduit.
— N'hésitez pas à plonger droit dans votre langue qui existe déjà dans le livre.
— Utilisez une Forme Informelle [59].
— Traduisez à la fois le contenu et le titre en même temps.
— Comparez au contenu anglais avant de soumettre une correction (si vous corrigez quelque chose, mais n'intégrez pas un changement "en amont", votre soumission ne sera pas acceptée).
— Si vous avez besoin d'écrire un terme anglais, entourez le avec les balises `<em>`. Ex : « asdf asdf *Controller* asdf » ou « asdf asdf Kontroller (*Controller*) asfd » comme il se doit.
— Ne soumettez pas de traductions partielles.
— Ne modifiez pas une section avec un changement en attente.
— N'utilisez pas d" entités HTML [60] pour les caractères accentués, le livre utilise UTF-8.

59. https://en.wikipedia.org/wiki/Register#Linguistics
60. https://en.wikipedia.org/wiki/List_of_XML_and_HTML_character_entity_references

— Ne changez pas les balises (HTML) de façon significative ou n'ajoutez pas de nouveau contenu.
— Si le contenu original manque d'informations, soumettez une modification pour cette version originale.

Guide de mise en forme de la documentation

La documentation du nouveau CakePHP est écrit avec le formatage du texte ReST[61]. ReST (Re Structured Text) est une syntaxe de texte de balisage similaire à markdown, ou textile. Pour maintenir la cohérence, il est recommandé quand vous ajoutez quelque chose à la documentation CakePHP que vous suiviez les directives suivantes sur la façon de formater et de structurer votre texte.

Longueur des lignes

Les lignes de texte doivent être de 80 colonnes au maximum. Seules exceptions, pour les URLs longues et les extraits de code.

En-têtes et Sections

Les sections d'en-tête sont créées par le soulignage du titre avec les caractères de ponctuation, avec une longueur de texte au moins aussi longue.

— # Est utilisé pour indiquer les titres de page.
— = Est utilisé pour les sections dans une page.
— - Est utilisé pour les sous-sections.
— ~ Est utilisé pour les sous-sous-sections.
— ^ Est utilisé pour les sous-sous-sous-sections.

Les en-têtes ne doivent pas être imbriqués sur plus de 5 niveaux de profondeur. Les en-têtes doivent être précédés et suivis par une ligne vide.

Les Paragraphes

Les paragraphes sont simplement des blocks de texte, avec toutes les lignes au même niveau d'indentation. Les paragraphes ne doivent être séparés par plus d'une ligne vide.

Le balisage interne

— Un astérisque : *text* pour une accentuation (italiques) Nous les utiliserons pour mettre en exergue des infos générales.
  — `*text*`.
— Deux astérisques : **text** pour une forte accentuation (caractères gras) Nous les utiliserons pour les répertoires de travail, les sujets de liste à puce, les noms de table et en excluant le mot « table » suivant.
  — `**/config/Migrations**`, `**articles**`, etc.
— Deux backquotes : `text` pour les exemples de code Nous les utiliserons pour les noms d'options de méthode, les noms de colonne des tables, les noms d'objet en excluant le mot « object » suivant et pour les noms de méthode/fonction – en incluant « () ».
  — ``cascadeCallbacks``, ``true``, ``id``, ``PagesController``, ``config()``, etc.

Si les astérisques ou les backquotes apparaissent dans le texte et peuvent être confondus avec les délimiteurs du balisage interne, ils doivent être échappés avec un backslash.

Le balisage interne a quelques restrictions :

— Il ne **doit pas** être imbriqué.
— Le contenu ne doit pas commencer ou finir avec un espace : `* text*` est mauvais.
— Le contenu doit être séparé du texte environnant par des caractères qui ne sont pas des mots. Utilisez un backslash pour échapper pour régler le problème : `unmot\ *engras*\ long`.

61. https://en.wikipedia.org/wiki/ReStructuredText

Listes

La liste du balisage est très similaire à celle de markdown. Les listes non ordonnées commencent par une ligne avec un unique astérisque et un espace. Les listes numérotées peuvent être créées avec, soit les numéros, soit # pour une numérotation automatique :

```
* C'est une balle
* Ici aussi. Mais cette ligne
  a deux lignes.

1. Première ligne
2. Deuxième ligne

#. Numérotation automatique
#. Va vous faire économiser du temps.
```

Les listes indentées peuvent aussi être créées, en indentant les sections et en les séparant avec une ligne vide :

```
* Première ligne
* Deuxième ligne

    * Allez plus profondément
    * Whoah

* Retour au premier niveau.
```

Les listes avec définitions peuvent être créées en faisant ce qui suit :

```
term
    définition
CakePHP
    Un framework MVC pour PHP
```

Les termes ne peuvent pas être sur plus d'une ligne, mais les définitions peuvent être multi-lignes et toutes les lignes doivent toujours être indentées.

Liens

Il y a plusieurs types de liens, chacun avec ses propres utilisations.

Liens externes

Les liens vers les documents externes peuvent être faits avec ce qui suit :

```
`Lien externe vers php.net <https://php.net>`_
```

Le lien résultant ressemblerait à ceci : Lien externe vers php.net [62]

Lien vers les autres pages

`:doc:`

Les autres pages de la documentation peuvent être liées en utilisant le modèle `:doc:`. Vous pouvez faire un lien à un document spécifique en utilisant, soit un chemin de référence absolu ou relatif. Vous pouvez omettre l'extension `.rst`. Par exemple, si la référence `:doc:`form`` apparait dans le document `core-helpers/html`, alors

62. https://php.net

le lien de référence `core-helpers/form`. Si la référence était `:doc:`/core-helpers` il serait en référence avec `/core-helpers` sans soucis de où il a été utilisé.

Les liens croisés de référencement

`:ref:`

Vous pouvez recouper un titre quelconque dans n'importe quel document en utilisant le modèle `:ref:`. Le label de la cible liée doit être unique à travers l'entière documentation. Quand on crée les labels pour les méthodes de classe, il vaut mieux utiliser `class-method` comme format pour votre label de lien.

L'utilisation la plus commune des labels est au-dessus d'un titre. Exemple :

```
.. _nom-label:

Section en-tête
---------------

Plus de contenu ici.
```

Ailleurs, vous pouvez référencer la section suivante en utilisant `:ref:`label-name`. Le texte du lien serait le titre qui précède le lien. Vous pouvez aussi fournir un texte de lien sur mesure en utilisant `:ref:`Texte de lien <nom-label>`.

Eviter l'Affichage d'Avertissements de Sphinx

Sphinx va afficher des avertissements si un fichier n'est pas référencé dans un toc-tree. C'est un bon moyen de s'assurer que tous les fichiers ont un lien pointé vers eux, mais parfois vous n'avez pas besoin d'insérer un lien pour un fichier, par exemple pour nos fichiers *epub-contents* et *pdf-contents*. Dans ces cas, vous pouvez ajouter `:orphan:` en haut du fichier pour supprimer les avertissements disant que le fichier n'est pas dans le toc-tree.

Description des classes et de leur contenu

La documentation de CakePHP utilise phpdomain [63] pour fournir des directives sur mesure pour décrire les objets PHP et les constructs. Utiliser les directives et les modèles est requis pour donner une bonne indexation et des fonctionnalités de référencement croisé.

Description des classes et constructs

Chaque directive remplit l'index, et l'index des espaces de nom.

`.. php:global:: name`

Cette directive déclare une nouvelle variable globale PHP.

`.. php:function:: name(signature)`

Définit une nouvelle fonction globale en-dehors de la classe.

`.. php:const:: name`

Cette directive déclare une nouvelle constante PHP, vous pouvez aussi l'utiliser imbriquée à l'intérieur d'une directive de classe pour créer les constantes de classe.

`.. php:exception:: name`

Cette directive déclare un nouvelle Exception dans l'espace de noms courant. La signature peut inclure des arguments du constructeur.

63. https://pypi.org/project/sphinxcontrib-phpdomain/

.. php:class:: name

Décrit une classe. Méthodes, attributs, et constantes appartenant à la classe doivent être à l'intérieur du corps de la directive :

```
.. php:class:: MyClass

    Description de la Classe

   .. php:method:: method($argument)

   Description de la méthode
```

Attributs, méthodes et constantes ne doivent pas être imbriqués. Ils peuvent aussi suivre la déclaration de classe :

```
.. php:class:: MyClass

    Texte sur la classe

.. php:method:: methodName()

    Texte sur la méthode
```

Voir aussi

php:method, *php:attr*, *php:const*

.. php:method:: name(signature)

Décrit une méthode de classe, ses arguments, les valeurs retournées et les exceptions :

```
.. php:method:: instanceMethod($one, $two)

    :param string $un: Le premier param\ètre.
    :param string $deux: Le deuxième param\ètre.
    :returns: Un tableau de trucs.
    :throws: InvalidArgumentException

   C\'est un m\éthode d\'instanciation.
```

.. php:staticmethod:: ClassName::methodName(signature)

Décrire une méthode statique, ses arguments, les valeurs retournées et les exceptions.

see *php:method* pour les options.

.. php:attr:: name

Décrit une propriété/attribut sur une classe.

Eviter l'Affichage d'Avertissements de Sphinx

Sphinx va afficher des avertissements si une fonction est référencée dans plusieurs fichiers. C'est un bon moyen de s'assurer que vous n'avez pas ajouté une fonction deux fois, mais parfois vous voulez en fait écrire une fonction dans deux ou plusieurs fichiers, par exemple *debug object* est référencé dans */development/debugging* et dans */core-libraries/global-constants-and-functions*. Dans ce cas, vous pouvez ajouter `:noindex:` sous la fonction debug pour supprimer les avertissements. Gardez uniquement une référence **sans** `:no-index:` pour que la fonction soit référencée :

```
.. php:function:: debug(mixed $var, boolean $showHtml = null, $showFrom = true)
    :noindex:
```

Référencement croisé

Les modèles suivants se réfèrent aux objets PHP et les liens sont générés si une directive assortie est trouvée :

:php:func:

Référence une fonction PHP.

:php:global:

Référence une variable globale dont le nom a un préfixe $.

:php:const:

Référence soit une constante globale, soit une constante de classe. Les constantes de classe doivent être précédées par la classe propriétaire :

```
DateTime a une constante :php:const:`DateTime::ATOM`.
```

:php:class:

Référence une classe par nom :

```
:php:class:`ClassName`
```

:php:meth:

Référence une méthode d'une classe. Ce modèle supporte les deux types de méthodes :

```
:php:meth:`DateTime::setDate`
:php:meth:`Classname::staticMethod`
```

:php:attr:

Référence une propriété d'un objet :

```
:php:attr:`ClassName::$propertyName`
```

:php:exc:

Référence une exception.

Code source

Les blocks de code littéral sont créés en finissant un paragraphe avec `::`. Le block littéral doit être indenté, et comme pour tous les paragraphes, être séparé par des lignes uniques :

```
C'est un paragraphe::

    while ($i--) {
        faireDesTrucs()
    }

C'est un texte régulier de nouveau.
```

Le texte littéral n'est pas modifié ou formaté, la sauvegarde du niveau d'indentation est supprimée.

Notes et avertissements

Il y a souvent des fois où vous voulez informer le lecteur d'une astuce importante, une note spéciale ou un danger potentiel. Les avertissements dans sphinx sont justement utilisés pour cela. Il y a cinq types d'avertissements.

- `.. tip::` Les astuces sont utilisées pour documenter ou ré-itérer des informations intéressantes ou importantes. Le contenu de cette directive doit être écrit dans des phrases complètes et inclure toutes les ponctuations appropriées.
- `.. note::` Les notes sont utilisées pour documenter une information particulièrement importante. Le contenu de cette directive doit être écrit dans des phrases complètes et inclure toutes les ponctuations appropriées.
- `.. warning::` Les avertissements sont utilisés pour documenter des blocks potentiellement dangereux, ou des informations relatives à la sécurité. Le contenu de la directive doit être écrite en phrases complètes et inclure toute la ponctuation appropriée.
- `.. versionadded:: X.Y.Z` Les avertissements « ajouté en version X.Y.Z » sont utilisés pour spécifier l'ajout de fonctionnalités dans une version spécifique, `X.Y.Z` étant la version à laquelle l'ajout de la fonctionnalité en question a eu lieu
- `.. deprecated:: X.Y.Z` À la différence des avertissements « ajouté en version », les avertissements « déprécié en version » servent à indiquer la dépréciation d'une fonctionnalité à une version précise, `X.Y.Z` étant la version à laquelle le retrait de la fonctionnalité en question a eu lieu.

Tous les avertissements sont faits de la même façon :

```
.. note::

    Indenté, précédé et suivi par une ligne vide. Exactement comme
    un paragraphe.

Ce texte n'est pas une partie de la note.
```

Exemples

Astuce

C'est une astuce utile que vous allez probablement oubliée.

Note

Vous devriez y faire attention.

Avertissement

Cela pourrait être dangereux.

Ajouté dans la version 4.0.0 : Cette super fonctionnalité a été ajoutée à partir de la version 4.0.0.

Obsolète depuis la version 4.0.1 : Cette vieille fonctionnalité a été dépréciée à partir de la version 4.0.1.

Tickets

Avoir des retours et de l'aide de la communauté sous forme de tickets est une partie extrêmement importante dans le processus de développement de CakePHP. Tous les tickets CakePHP sont hébergés sur Github [64].

Rapporter des bugs

Bien écrits, les rapports de bug sont très utiles. Il y a quelques étapes pour faire le meilleur rapport de bug possible :

- **A Faire** Merci de faire des recherches [65] pour un ticket similaire éventuellement existant, et s'assurer que personne n'a déjà reporté le bug ou qu'il n'a pas déjà été résolu dans le répertoire.
- **A Faire** Merci d'inclure des instructions détaillées sur la manière de **reproduire le bug**. Cela peut être sous la forme de cas de Test ou un bout de code démontrant le problème. Ne pas avoir une possibilité de reproduire le problème, signifie qu'il est moins facile de le régler.
- **A Faire** Merci de donner autant de détails que possible sur votre environnement : (OS, version de PHP, Version de CakePHP).
- **A ne pas Faire** : Merci de ne pas utiliser un ticket système pour poser une question de support technique. Le canal IRC #cakephp sur Freenode [66] a plusieurs développeurs prêts à répondre à vos questions. Les autres possibilités sont le Groupe Google de CakePHP [67] et le déjà populaire Stack Overflow [68].

Rapporter des problèmes de sécurité

Si vous avez trouvé un problème de sécurité dans CakePHP, merci de bien vouloir utiliser la procédure suivante, plutôt que le système de rapport de bug classique. Au lieu d'utiliser le tracker de bug, la mailing-liste ou le canal IRC, merci d'envoyer un email à **security [at] cakephp.org**. Les emails envoyés à cette adresse vont à l'équipe qui construit le cœur de CakePHP via une mailing-liste privée.

Pour chaque rapport, nous essayons d'abord de confirmer la vulnérabilité. Une fois confirmée, l'équipe du cœur de CakePHP va entreprendre les actions suivantes :

- La reconnaissance faite au rapporteur que nous avons reçu son problème, et que nous travaillons à sa réparation. Nous demandons au rapporteur de garder le problème confidentiel jusqu'à ce que nous l'annoncions.
- Obtenir une préparation d'un fix/patch.
- Préparer un message décrivant la vulnérabilité, et sa possible exploitation.
- Sortir de nouvelles versions de toutes les versions affectées.
- Montrer de façon évidente le problème dans la publication de l'annonce.

Code

Les correctifs et les pull requests sont les meilleures façons de contribuer au code de CakePHP. Les pull requests peuvent être créés sur Github, et sont préférés aux correctifs attachés aux tickets.

Configuration Initiale

Avant de travailler sur les correctifs pour CakePHP, c'est une bonne idée de définir la configuration de votre environnement. Vous aurez besoin des logiciels suivants :

- Git
- PHP 8.1 ou supérieur
- PHPUnit 5.7.0 ou supérieur

Mettez en place vos informations d'utilisateur avec votre nom/titre et adresse e-mail de travail :

64. https://github.com/cakephp/cakephp/issues
65. https://github.com/cakephp/cakephp/search?q=it+is+broken&ref=cmdform&type=Issues
66. https://webchat.freenode.net
67. https://groups.google.com/group/cake-php
68. https://stackoverflow.com/questions/tagged/cakephp

```
git config --global user.name 'Bob Barker'
git config --global user.email 'bob.barker@example.com'
```

Note

Si vous êtes nouveau sous Git, nous vous recommandons fortement de lire l'excellent livre gratuit ProGit [69].

Récupérez un clone du code source de CakePHP sous GitHub :

- Si vous n'avez pas de compte github [70] , créez-en un.
- Forkez le dépôt de CakePHP [71] en cliquant sur le bouton **Fork**.

Après que le fork est fait, clonez votre fork sur votre machine locale :

```
git clone git@github.com:YOURNAME/cakephp.git
```

Ajoutez le dépôt CakePHP d'origine comme un dépôt distant. Vous utiliserez ceci plus tard pour aller chercher les changements du dépôt CakePHP. Cela vous permettra de rester à jour avec CakePHP :

```
cd cakephp
git remote add upstream git://github.com/cakephp/cakephp.git
```

Maintenant que vous avez configuré CakePHP, vous devriez être en mesure de définir une *connexion à la base* `$test`, et *d'exécuter tous les tests*.

Travailler sur un Correctif

A chaque fois que vous voulez travailler sur un bug, une fonctionnalité ou une amélioration, créez une branche avec un sujet.

La branche que vous créez devra être basée sur la version pour laquelle votre correctif/amélioration tourne. Par exemple, si vous réglez un bug dans `3.x`, vous pouvez utiliser la branche `master` comme base de votre branche. Si vos changements ont pou objet de régler un bug pour les séries de version 2.x, vous devrez utiliser la branche `2.x`. Cela simplifiera la fusion future de vos changements puisque Github ne vous permet pas de modifier la branche cible :

```
# régler un bug dans 3.x
git fetch upstream
git checkout -b ticket-1234 upstream/master

# régler un bug dans 2.x
git fetch upstream
git checkout -b ticket-1234 upstream/2.x
```

Astuce

Utiliser un nom descriptif pour vos branches, en référence au ticket ou au nom de la fonctionnalité, est une bonne convention. Ex : ticket-1234, great-fonctionnality.

Ce qui précède va créer une branche locale basée sur la branche (CakePHP) 2.x en amont. Travaillez sur votre correctif, et faîtes autant de commits que vous le souhaitez ; mais gardez à l'esprit ce qui suit :

69. https://git-scm.com/book/
70. https://github.com
71. https://github.com/cakephp/cakephp

— Suivez ceci *Normes de codes*.
— Ajoutez un cas de test pour montrer que le bug est réglé, ou que la nouvelle fonctionnalité marche.
— Faîtes des commits logiques, et écrivez des messages de commit bien clairs et concis.

Soumettre un Pull Request

Une fois que vos changements sont faits et que vous êtes prêts pour la fusion dans CakePHP, vous pouvez mettre à jour votre branche :

```
# Rebaser un fix en haut de la branche master
git checkout master
git fetch upstream
git merge upstream/master
git checkout <branch_name>
git rebase master
```

Cela récupérera et fusionnera tous les changements qui se sont passés dans CakePHP depuis que vous avez commencé. Cela rebasera - ou remettra vos changements au dessus du code actuel. Il y aura peut-être un conflit pendant le `rebase`. Si le rebase quitte rapidement, vous pourrez voir les fichiers qui sont en conflit/Non fusionnés avec `git status`. Résolvez chaque conflit et continuer le rebase :

```
git add <filename> # Faîtes ceci pour chaque fichier en conflit.
git rebase --continue
```

Vérifiez que tous les tests continuent de fonctionner. Ensuite faîtes un push de votre branche à votre fork :

```
git push origin <branch-name>
```

Si vous avez rebasé après avoir pusher votre branche, vous devrez utiliser le push avec l'option force :

```
git push --force origin <branch-name>
```

Une fois que votre branche est sur GitHub, vous pouvez soumettre un pull request sur GitHub.

Choisir l'Emplacement dans lequel vos Changements seront Fusionnés

Quand vous faîtes vos pull requests, vous devez vous assurer de sélectionner la bonne branche de base, puisque vous ne pouvez pas l'éditer une fois que le pull request est créée.

— Si votre changement est un **bugfix** et n'introduit pas de nouvelles fonctionnalités et corrige seulement un comportement existant qui est présent dans la version courante. Dans ce cas, choisissez **master** comme votre cible de fusion.
— Si votre changement est une **nouvelle fonctionnalité** ou un ajout au framework, alors vous devez choisir la branche avec le nombre de la version prochaine. Par exemple si la version stable courante est `4.2.0`, la branche acceptant les nouvelles fonctionnalités sera `4.next`.
— Si votre changement casse une fonctionnalité existante, ou casse l'API, alors vous devrez choisir la prochaine version majeure. Par exemple, si la version courante est `4.2.0` alors la prochaine fois qu'un comportement peut être cassé sera dans `5.x` ainsi vous devez cibler cette branche.

Note

Souvenez-vous que tout le code auquel vous contribuez pour CakePHP sera sous Licence MIT, et la Cake Software Foundation [72] sera la propriétaire de toutes les contributions de code. Les contributeurs doivent suivre les Guidelines de la Communauté CakePHP [73].

72. https://cakefoundation.org/old
73. https://cakephp.org/get-involved

Tous les bugs réparés fusionnés sur une branche de maintenance seront aussi fusionnés périodiquement à la version publiée par l'équipe centrale (core team).

Normes de codes

Les développeurs de CakePHP vont utiliser le guide pour l'écriture de code PSR-12 [74] en plus des règles de code suivantes.

Il est recommandé que les autres personnes qui développent des Ingrédients de Cake suivent les mêmes normes.

Vous pouvez utiliser le Code Sniffer de CakePHP [75] pour vérifier que votre code suit les normes requises.

Ajout de Nouvelles Fonctionnalités

Aucune nouvelle fonctionnalité ne devrait être ajoutée, sans avoir fait ses propres tests - qui doivent être validés avant de les committer au dépôt.

Configuration de l'IDE

Merci de vous assurer que votre IDE est configuré avec « trim right » pour les espaces vides. Il ne doit pas y a voir d'espaces à la fin des lignes.

La plupart des IDE modernes supporte aussi un fichier `.editorconfig`. Le squelette d'application CakePHP est fourni avec par défaut. Il contient déjà les meilleurs pratiques par défaut.

Indentation

Quatre espaces seront utilisés pour l'indentation.

Ainsi, l'indentation devrait ressembler à ceci :

```
// niveau de base
    // niveau 1
        // niveau 2
    // niveau 1
// niveau de base
```

Ou :

```
$booleanVariable = true;
$stringVariable = "moose";
if ($booleanVariable) {
    echo "Valeur booléenne si true";
    if ($stringVariable === "élan") {
        echo "Nous avons rencontré un élan";
    }
}
```

Dans les cas où vous utilisez un appel de fonction multi-lignes, utilisez les instructions suivantes :

- Les parenthèses ouvrantes d'un appel de fonction multi-lignes doivent être le dernier contenu de la ligne.
- Seul un argument est permis par ligne dans un appel de fonction multi-lignes.
- Les parenthèses fermantes d'un appel de fonction multi-lignes doivent être elles-même sur une ligne.

Par exemple, plutôt qu'utiliser le format suivant :

74. https://www.php-fig.org/psr/psr-12/
75. https://github.com/cakephp/cakephp-codesniffer

```
$matches = array_intersect_key($this->_listeners,
                array_flip(preg_grep($matchPattern,
                    array_keys($this->_listeners), 0)));
```

Utilisez ceci à la place :

```
$matches = array_intersect_key(
    $this->_listeners,
    array_flip(
        preg_grep($matchPattern, array_keys($this->_listeners), 0)
    )
);
```

Longueur des lignes

Il est recommandé de garder les lignes à une longueur d'environ 100 caractères pour une meilleure lisibilité du code. Les lignes ne doivent pas être plus longues que 120 caractères.

En résumé :

- 100 caractères est la limite soft.
- 120 caractères est la limite hard.

Structures de Contrôle

Les structures de contrôle sont par exemple « `if` », « `for` », « `foreach` », « `while` », « `switch` » etc. Ci-dessous, un exemple avec « `if` » :

```
if ((expr_1) || (expr_2)) {
    // action_1;
} elseif (!(expr_3) && (expr_4)) {
    // action_2;
} else {
    // default_action;
}
```

- Dans les structures de contrôle, il devrait y avoir 1 (un) espace avant la première parenthèse et 1 (un) espace entre les dernières parenthèses et l'accolade ouvrante.
- Toujours utiliser des accolades dans les structures de contrôle, même si elles ne sont pas nécessaires. Elles augmentent la lisibilité du code, et elles vous donnent moins d'erreurs logiques.
- L'ouverture des accolades doit être placée sur la même ligne que la structure de contrôle. La fermeture des accolades doit être placée sur de nouvelles lignes, et ils doivent avoir le même niveau d'indentation que la structure de contrôle. La déclaration incluse dans les accolades doit commencer sur une nouvelle ligne, et le code qu'il contient doit gagner un nouveau niveau d'indentation.
- Les attributs inline ne devraient pas être utilisés à l'intérieur des structures de contrôle.

```
// mauvais = pas d'accolades, déclaration mal placée
if (expr) statement;

// mauvais = pas d'accolades
if (expr)
    statement;

// bon
if (expr) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
    statement;
}

// mauvais = inline assignment
if ($variable = Class::function()) {
    statement;
}

// bon
$variable = Class::function();
if ($variable) {
    statement;
}
```

Opérateurs Ternaires

Les opérateurs ternaires sont permis quand l'opération entière rentre sur une ligne. Les opérateurs ternaires plus longs doivent être séparés en expression `if else`. Les opérateurs ternaires ne doivent pas être imbriqués. Des parenthèses optionnelles peuvent être utilisées autour de la condition vérifiée de l'opération pour rendre le code plus clair :

```
// Bien, simple et lisible
$variable = isset($options['variable']) ? $options['variable'] : true;

// Imbrications des ternaires est mauvaise
$variable = isset($options['variable']) ? isset($options['othervar']) ? true : false :␣
↪false;
```

Fichiers de Template

Dans les fichiers de template (fichiers .php) les développeurs devront utiliser les structures de contrôle en mot (keyword control structures). Les structures de contrôle en mot sont plus faciles à lire dans des fichiers de template complexes. Les structures de contrôle peuvent soit être contenues dans un block PHP plus large, soit dans des balises PHP séparées :

```
<?php
if ($isAdmin):
    echo '<p>Vous êtes l utilisateur admin.</p>';
endif;
?>
<p>Ce qui suit suit est aussi acceptable:</p>
<?php if ($isAdmin): ?>
    <p>Vous êtes l utilisateur admin.</p>
<?php endif; ?>
```

Comparaison

Toujours essayer d'être aussi strict que possible. Si un test non strict est délibéré, il peut être sage de le commenter afin d'éviter de le confondre avec une erreur.

Pour tester si une variable est null, il est recommandé d'utiliser une vérification stricte :

```
if ($value === null) {
      // ...
}
```

La valeur avec laquelle on vérifie devra être placée sur le côté droit :

```
// non recommandé
if (null === $this->foo()) {
    // ...
}

// recommandé
if ($this->foo() === null) {
    // ...
}
```

Appels des Fonctions

Les fonctions doivent être appelées sans espace entre le nom de la fonction et la parenthèse ouvrante. Il doit y avoir un espace entre chaque paramètre d'un appel de fonction :

```
$var = foo($bar, $bar2, $bar3);
```

Comme vous pouvez le voir, il doit y avoir un espace des deux côtés des signes égal (=).

Définition des Méthodes

Exemple d'une définition de méthode :

```
public function someFunction($arg1, $arg2 = '')
{
    if (expr) {
        statement;
    }

    return $var;
}
```

Les paramètres avec une valeur par défaut, doivent être placés en dernier dans la définition de la fonction. Essayez de faire en sorte que vos fonctions retournent quelque chose, au moins `true` ou `false`, ainsi cela peut déterminer si l'appel de la fonction est un succès :

```
public function connection($dns, $persistent = false)
{
    if (is_array($dns)) {
        $dnsInfo = $dns;
    } else {
        $dnsInfo = BD::parseDNS($dns);
    }

    if (!($dnsInfo) || !($dnsInfo['phpType'])) {
        return $this->addError();
    }

    return true;
}
```

Il y a des espaces des deux côtés du signe égal.

Typehinting

Les arguments qui attendent des objets, des tableaux ou des callbacks (fonctions de rappel) peuvent être typés. Nous ne typons que les méthodes publiques car le typage prend du temps :

```
/**
 * Some method description.
 *
 * @param \Cake\ORM\Table $table The table class to use.
 * @param array $array Some array value.
 * @param callable $callback Some callback.
 * @param bool $boolean Some boolean value.
 */
public function foo(Table $table, array $array, callable $callback, $boolean)
{
}
```

Ici `$table` doit être une instance de `\Cake\ORM\Table`, `$array` doit être un `array` et `$callback` doit être de type `callable` (un callback valide).

Notez que si vous souhaitez autoriser que `$array` soit aussi une instance de `\ArrayObject`, vous ne devez pas typer puisque `array` accepte seulement le type primitif :

```
/**
 * Description de la method.
 *
 * @param array|\ArrayObject $array Some array value.
 */
public function foo($array)
{
}
```

Fonctions Anonymes (Closures)

La définition des fonctions anonymes suit le guide sur le style de codage PSR-12 [76], où elles sont déclarées avec un espace après le mot clé *function*, et un espace avant et après le mot clé *use* :

```
$closure = function ($arg1, $arg2) use ($var1, $var2) {
    // code
};
```

Chaînage des Méthodes

Le chaînage des méthodes doit avoir plusieurs méthodes réparties sur des lignes distinctes et indentées avec quatre espaces :

```
$email->from('foo@example.com')
    ->to('bar@example.com')
    ->subject('Un super message')
    ->send();
```

76. https://www.php-fig.org/psr/psr-12/

Commenter le Code

Tous les commentaires doivent être écrits en anglais, et doivent clairement décrire le block de code commenté.

Les commentaires doivent inclure les tags de phpDocumentor[77] suivants :

- @deprecated[78] Using the `@version <vector> <description>` format, where `version` and `description` are mandatory.
- @example[79]
- @ignore[80]
- @internal[81]
- @link[82]
- @see[83]
- @since[84]
- @version[85]

Les tags de PhpDoc sont un peu du même style que les tags de JavaDoc dans Java. Les tags sont seulement traités s'il sont la première chose dans la ligne DocBlock, par exemple :

```
/**
 * Exemple de Tag.
 *
 * @author ce tag est analysé, mais @version est ignoré
 * @version 1.0 ce tag est aussi analysé
 */
```

```
/**
 * Exemple de tag inline phpDoc.
 *
 * Cette fonction travaille dur avec foo() pour gouverner le monde.
 *
 * @return void
 */
function bar()
{
}

/**
 * Foo function
 *
 * @return void
 */
function foo()
{
}
```

Les blocks de commentaires, avec une exception du premier block dans le fichier, doivent toujours être précédés par un retour à la ligne.

77. https://phpdoc.org
78. https://docs.phpdoc.org/latest/guide/references/phpdoc/tags/deprecated.html
79. https://docs.phpdoc.org/latest/guide/references/phpdoc/tags/example.html
80. https://docs.phpdoc.org/latest/guide/references/phpdoc/tags/ignore.html
81. https://docs.phpdoc.org/latest/guide/references/phpdoc/tags/internal.html
82. https://docs.phpdoc.org/latest/guide/references/phpdoc/tags/link.html
83. https://docs.phpdoc.org/latest/guide/references/phpdoc/tags/see.html
84. https://docs.phpdoc.org/latest/guide/references/phpdoc/tags/since.html
85. https://docs.phpdoc.org/latest/guide/references/phpdoc/tags/version.html

Types de Variables

Les types de variables pour l'utilisation dans DocBlocks :

Type
: Description

mixed
: Une variable avec un type indéfini (ou multiple).

int
: Variable de type Integer (Tout nombre).

float
: Type Float (nombres à virgule).

bool
: Type Logique (true ou false).

string
: Type String (toutes les valeurs en « « ou " ").

null
: Type null. Habituellement utilisé avec un autre type.

array
: Type Tableau.

object
: Type Objet.

resource
: Type Ressource (retourné par exemple par mysql_connect()). Rappelez vous que quand vous spécifiez un type en mixed, vous devez indiquer s'il est inconnu, ou les types possibles.

callable
: Fonction de rappel.

Vous pouvez aussi combiner les types en utilisant le caractère pipe :

```
int|bool
```

Pour plus de deux types, il est habituellement mieux d'utiliser seulement `mixed`.

Quand vous retournez l'objet lui-même, par ex pour chaîner, vous devriez utiliser `$this` à la place :

```
/**
 * Foo function.
 *
 * @return $this
 */
public function foo()
{
    return $this;
}
```

Inclure les Fichiers

`include`, `require`, `include_once` et `require_once` n'ont pas de parenthèses :

```
// mauvais = parenthèses
require_once('ClassFileName.php');
require_once ($class);
```

(suite sur la page suivante)

(suite de la page précédente)

```
// bon = pas de parenthèses
require_once 'ClassFileName.php';
require_once $class;
```

Quand vous incluez les fichiers avec des classes ou librairies, utilisez seulement et toujours la fonction require_once [86].

Les Balises PHP

Toujours utiliser les balises longues (`<?php ?>`) plutôt que les balises courtes (`<? ?>`). L'echo court doit être utilisé dans les fichiers de template (**.php**) lorsque cela est nécessaire.

Echo court

L'echo court doit être utilisé dans les fichiers de vue à la place de `<?php echo`. Il doit être immédiatement suivi par un espace unique, la variable ou la valeur de la fonction pour faire un `echo`, un espace unique, et la balise de fermeture de php :

```
// wrong = semicolon, aucun espace
<td><?=$name;?></td>

// good = espaces, aucun semicolon
<td><?= $name ?></td>
```

Depuis PHP 5.4, le tag echo court (`<?=`) ne doit plus être considéré. un "tag court" est toujours disponible quelque soit la directive ini de `short_open_tag`.

Convention de Nommage

Fonctions

Ecrivez toutes les fonctions en camelBack :

```
function nomDeFonctionLongue()
{
}
```

Classes

Les noms de classe doivent être écrits en CamelCase, par exemple :

```
class ClasseExemple
{
}
```

Variables

Les noms de variable doivent être aussi descriptifs que possible, mais aussi courts que possible. Tous les noms de variables doivent démarrer avec une lettre minuscule, et doivent être écrites en camelBack s'il y a plusieurs mots. Les variables contenant des objets doivent d'une certaine manière être associées à la classe d'où elles proviennent. Exemple :

86. https://php.net/require_once

```
$user = 'John';
$users = ['John', 'Hans', 'Arne'];

$dispatcher = new Dispatcher();
```

Visibilité des Membres

Utilisez les mots clés `public`, `protected` et `private` de PHP pour les méthodes et les variables.

Exemple d'Adresses

Pour tous les exemples d'URL et d'adresse email, utilisez « example.com », « example.org » et « example.net », par exemple :

- Email : someone@example.com
- WWW : http://www.example.com
- FTP : ftp://ftp.example.com

Le nom de domaine « example.com » est réservé à cela (voir **RFC 2606** [87]) et est recommandé pour l'utilisation dans la documentation ou comme exemples.

Fichiers

Les noms de fichier qui ne contiennent pas de classes, doivent être écrits en minuscules et soulignés, par exemple :

```
nom_de_fichier_long.php
```

Casting

Pour le casting, nous utilisons :

Type
: Description

(bool)
: Cast pour boolean.

(int)
: Cast pour integer.

(float)
: Cast pour float.

(string)
: Cast pour string.

(array)
: Cast pour array.

(object)
: Cast pour object.

Constantes

Les constantes doivent être définies en majuscules :

```
define('CONSTANTE', 1);
```

Si un nom de constante a plusieurs mots, ils doivent être séparés par un caractère underscore, par exemple :

87. https://datatracker.ietf.org/doc/html/rfc2606.html

```
define('NOM_LONG_DE_CONSTANTE', 2);
```

Attention quand vous utilisez empty()/isset()

While `empty()` often seems correct to use, it can mask errors and cause unintended effects when `'0'` and `0` are given. When variables or are already defined, the usage of `empty()` is not recommended. When working with variables, it is better to rely on type-coercion to boolean instead of `empty()` :

```
function manipulate($var)
{
    // Not recommended, $var is already defined in the scope
    if (empty($var)) {
        // ...
    }

    // Use boolean type coercion
    if (!$var) {
        // ...
    }
    if ($var) {
        // ...
    }
}
```

When dealing with defined properties you should favour `null` checks over `empty()/isset()` checks :

```
class Thing
{
    private $property; // Defined

    public function readProperty()
    {
        // Not recommended as the property is defined in the class
        if (!isset($this->property)) {
            // ...
        }
        // Recommended
        if ($this->property === null) {

        }
    }
}
```

When working with arrays, it is better to merge in defaults over using `empty()` checks. By merging in defaults, you can ensure that required keys are defined :

```
function doWork(array $array)
{
    // Merge defaults to remove need for empty checks.
    $array += [
        'key' => null,
    ];
```

(suite sur la page suivante)

(suite de la page précédente)

```
    // Not recommended, the key is already set
    if (isset($array['key'])) {
        // ...
    }

    // Recommended
    if ($array['key'] !== null) {
        // ...
    }
}
```

Guide de Compatibilité Rétroactive

Nous assurer que la mise à jour de vos applications se fasse facilement et en douceur est important à nos yeux. C'est pour cela que nous ne cassons la compatibilité que pour les versions majeures. Vous connaissez peut-être le versioning sémantique [88] qui est la règle générale que nous utilisons pour tous les projets CakePHP. En résumé, le versioning sémantique signifie que seules les versions majeures (comme 2.0, 3.0, 4.0) peuvent casser la compatibilité rétroactive. Les versions mineures (comme 2.1, 3.1, 3.2) peuvent introduire de nouvelles fonctionnalités, mais ne cassent pas la compatibilité. Les versions de fix de Bug (comme 2.1.2, 3.0.1) n'ajoutent pas de nouvelles fonctionnalités, mais règlent seulement des bugs ou améliorent la performance.

> **Note**
>
> CakePHP a commencé à utiliser le versioning sémantique à partir de la version 2.0.0. Ces règles ne s'appliquent pas pour la version 1.x.

Pour clarifier les changements que vous pouvez attendre dans chaque version en entier, nous avons plus d'informations détaillées pour les développeurs utilisant CakePHP et pour les développeurs travaillant sur CakePHP qui définissent les attentes de ce qui peut être fait dans des versions mineures. Les versions majeures peuvent avoir autant de changements que nécessaires.

Guides de Migration

Pour chaque version majeure et mineure, l'équipe de CakePHP va fournir un guide de migration. Ces guides expliquent les nouvelles fonctionnalités et tout changement entraînant des modifications de chaque version. Ils se trouvent dans la section *Annexes* du cookbook.

Utiliser CakePHP

Si vous construisez votre application avec CakePHP, les conventions suivantes expliquent la stabilité que vous pouvez attendre.

Interfaces

En-dehors des versions majeures, les interfaces fournies par CakePHP **ne** vont **pas** connaître de modification des méthodes existantes. De nouvelles méthodes peuvent être ajoutées, mais aucune méthode existante ne sera changée.

88. https://semver.org/

Classes

Les classes fournies par CakePHP peuvent être construites et ont leurs méthodes public et les propriétés utilisées par le code de l'application et en-dehors des versions majeures, la compatibilité rétroactive est assurée.

> **Note**
>
> Certaines classes dans CakePHP sont marquées avec la balise de doc `@internal` de l'API. Ces classes **ne** sont **pas** stables et n'assurent pas forcément de compatibilité rétroactive.

Dans les versions mineures, les nouvelles méthodes peuvent être ajoutées aux classes, et les méthodes existantes peuvent avoir de nouveaux arguments ajoutés. Tout argument nouveau aura des valeurs par défaut, mais si vous surchargez des méthodes avec une signature différente, vous verrez peut-être des erreurs fatales. Les méthodes qui ont de nouveaux arguments ajoutés seront documentées dans le guide de migration pour cette version.

La table suivante souligne plusieurs cas d'utilisations et la compatibilité que vous pouvez attendre de CakePHP :

Si vous…	Backwards compatibility ?
Typehint against the class	Oui
Crée une nouvelle instance	Oui
Etendre la classe	Oui
Access a public property	Oui
Appel d'une méthode publique	Oui
Etendre une classe et…	
Surcharger une propriété publique	Oui
Accéder à une propriété protégée	Non [1]
Surcharger une propriété protégée	Non [1]
Surcharger une méthode protégée	Non [1]
Apple d'une méthode protégée	Non [1]
Ajouter une propriété publique	Non
Ajouter une méthode publique	Non
Ajouter un argument pour une méthode qui surcharge	Non [1]
Ajouter une valeur d'argument par défaut pour une méthode existante	Oui

Travailler avec CakePHP

Si vous aidez à rendre CakePHP encore meilleur, merci de garder à l'esprit les conventions suivantes lors des ajouts/changements de fonctionnalités :

Dans une version mineure, vous pouvez :

1. Votre code *peut* être cassé par des versions mineures. Vérifiez le guide de migration pour plus de détails.

Dans une versions mineure, pouvez-vous...	
Classes	
Retirer une classe	Non
Retirer une interface	Non
Retirer un trait	Non
Faire des final	Non
Faire des abstract	Non
Changer de nom	Oui [2]
Propriétés	
Ajouter une propriété publique	Oui
Retirer une propriété publique	Non
Ajouter une propriété protégée	Oui
Retirer une propriété protégée	Oui [3]
Méthodes	
Ajouter une méthode publique	Oui
Retirer une méthode publique	Non
Ajouter une méthode protégée	Oui
Déplacer une classe parente	Oui
Retirer une méthode protégée	Oui[3]
Réduire la visibilité	Non
Changer le nom de méthode	Oui[2]
Ajouter un nouvel argument avec la valeur par défaut	Oui
Ajouter un nouvel argument requis pour une méthode existante.	Non
Retirer une valeur par défaut à partir d'un argument existant	Non

Depréciations

Dans chaque version mineure, les fonctionnalités peuvent être dépréciées. Si les fonctionnalités sont dépréciées, la documentation de l'API et des avertissements à l'exécution seront ajoutés. Les erreurs à l'exécution vous aideront à localiser le code qui doit être mis à jour avant qu'il ne casse. Si vous souhaitez désactiver les avertissements à l'exécution, vous pouvez le faire en utilisant la valeur de configuration `Error.errorLevel` :

```
// dans config/app.php
// ...
'Error' => [
    'errorLevel' => E_ALL ^ E_USER_DEPRECATED,
]
// ...
```

Va désactiver les avertissements de dépréciation à l'exécution.

2. Vous pouvez changer des noms de classe/méthode tant que le vieux nom reste disponible. C'est généralement à éviter à moins que le renommage apporte un vrai bénéfice.
3. Nous essayons d'éviter ceci à tout prix. Tout retrait doit être documenté dans le guide de migration.

CHAPITRE 6

Installation

CakePHP a quelques exigences système :

- Un serveur HTTP. Par exemple : Apache. Avoir mod_rewrite est préférable, mais en aucun cas nécessaire. Vous pouvez également utiliser nginx ou Microsoft IIS si vous préférez.
- PHP minimum 8.1 (**8.4** pris en charge).
- L'extension PHP mbstring
- L'extension PHP intl
- L'extension PHP simplexml
- L'extension PHP PDO

> **Note**
>
> Dans XAMPP, l'extension intl est incluse mais vous devez décommenter `extension=php_intl.dll` (ou `extension=intl`) dans **php.ini** et redémarrer le serveur dans le Panneau de Contrôle de XAMPP.
>
> Dans WAMP, l'extension intl est « activée » par défaut mais ne fonctionne pas. Pour la rendre fonctionnelle, dirigez-vous dans le dossier php (par défaut) **C :\wamp\bin\php\php{version}**, copiez tous les fichiers qui ressemblent à **icu*.dll** et collez-les dans le répertoire bin d'apache **C :\wamp\bin\apache\apache{version}\bin**. Ensuite redémarrez tous les services et tout devrait être bon.

Techniquement, un moteur de base de données n'est pas nécessaire, mais nous imaginons que la plupart des applications vont en utiliser un. CakePHP supporte une diversité de moteurs de stockage de données :

- MySQL (5.7 ou supérieur)
- MariaDB (10.1 ou supérieur)
- PostgreSQL (9.6 ou supérieur)
- Microsoft SQL Server (2012 ou supérieur)
- SQLite 3

La base de données Oracle est prise en charge via le plugin communautaire *Pilote pour Oracle Database <https ://github.com/CakeDC/cakephp-oracle-driver>* _.

> **Note**
>
> Tous les drivers intégrés requièrent PDO. Vous devez vous assurer que vous avez les bonnes extensions PDO installées.

Installer CakePHP

Avant de commencer, vous devez vous assurer que votre version de PHP est à jour :

```
php -v
```

Vous devez avoir PHP 8.1 (CLI) ou supérieur. La version PHP de votre serveur Web doit également être de 8.1 ou supérieur, le serveur web doit utiliser la même version de PHP que votre interface en ligne de commande (CLI).

Installer Composer

CakePHP utilise Composer [89], un outil de gestion de dépendances comme méthode officielle supportée pour l'installation.

— Installer Composer sur Linux et macOS

1. Exécutez le script d'installation comme décrit dans la documentation officielle de Composer [90] et suivez les instructions pour installer Composer.
2. Exécutez la commande suivante pour déplacer composer.phar vers un répertoire qui est dans votre path :

```
mv composer.phar /usr/local/bin/composer
```

— Installer Composer sur Windows

Pour les systèmes Windows, vous pouvez télécharger l'installeur Windows de Composer ici [91]. D'autres instructions pour l'installeur Windows de Composer se trouvent dans le README ici [92].

Créer un Projet CakePHP

Vous pouvez créer une nouvelle application CakePHP en utilisant la commande `create-project` de Composer :

```
composer create-project --prefer-dist cakephp/app:~5.0 my_app_name
```

Une fois que Composer a fini le téléchargement du squelette de l'application et du cœur de la librairie de CakePHP, vous devriez avoir une application CakePHP fonctionnelle, installée via Composer. Assurez-vous de garder les fichiers composer.json et composer.lock avec le reste de votre code source.

Vous pouvez maintenant naviguer vers le chemin où vous avez installé votre application CakePHP et voir la page d'accueil par défaut. Pour changer le contenu de cette page, modifiez : **templates/Pages/home.php**.

Bien que Composer soit la méthode d'installation recommandée, il existe des versions pré-installées disponibles sur Github [93]. Ces téléchargements contiennent le squelette d'une application avec toutes les dépendances installées. Le `composer.phar` est aussi inclus donc vous avez tout ce qui est nécessaire pour pouvoir l'utiliser.

89. https://getcomposer.org
90. https://getcomposer.org/download/
91. https://github.com/composer/windows-setup/releases/
92. https://github.com/composer/windows-setup
93. https://github.com/cakephp/cakephp/tags

Rester à jour avec les derniers changements de CakePHP

Par défaut le **composer.json** de l'application ressemble à cela :

```
"require": {
    "cakephp/cakephp": "5.0.*"
 }
```

A chaque fois que vous exécutez `php composer.phar update`, vous recevrez des correctifs pour cette version mineure. Vous pouvez cependant modifier la version de CakePHP en `^5.0` pour recevoir également les dernières versions mineures stables de la branche `5.x`.

Permissions

CakePHP utilise le répertoire **tmp** pour un certain nombre d'opérations. Les descriptions de model, les vues mises en cache, et les informations de session en sont juste quelques exemples. Le répertoire **logs** est utilisé pour écrire les fichiers de log par le moteur par défaut `FileLog`.

A ce titre, assurez-vous que les répertoires **logs**, **tmp** et tous ses sous-répertoires dans votre installation CakePHP sont accessibles en écriture pour l'utilisateur du serveur web. Le processus d'installation avec Composer va rendre **tmp** et ses sous-dossiers accessibles en écriture pour que l'application fonctionne rapidement, mais vous pouvez mettre à jour les permissions pour une meilleur sécurité et les garder en écriture seulement pour l'utilisateur du serveur web.

Un problème habituel est que les répertoires **logs** et **tmp** et les sous-répertoires doivent être accessibles en écriture à la fois pour le serveur web et pour l'utilisateur des lignes de commande. Sur un système UNIX, si votre utilisateur du serveur web est différent de l'utilisateur des lignes de commande, vous pouvez lancer les commandes suivantes, une seule fois, dans votre projet pour vous assurer que les permissions sont bien configurées :

```
HTTPDUSER=`ps aux | grep -E '[a]pache|[h]ttpd|[_]www|[w]ww-data|[n]ginx' | grep -v root␣
↪| head -1 | cut -d\  -f1`
setfacl -R -m u:${HTTPDUSER}:rwx tmp
setfacl -R -d -m u:${HTTPDUSER}:rwx tmp
setfacl -R -m u:${HTTPDUSER}:rwx logs
setfacl -R -d -m u:${HTTPDUSER}:rwx logs
```

Si vous souhaitez utiliser les outils de la console CakePHP, vous devez vous assurer que le fichier `bin/cake` est exécutable. Sur *nix ou macOS, vous pouvez simplement exécuter la commande suivante :

```
chmod +x bin/cake
```

Sur Windows, le fichier **.bat** devrait déjà être exécutable. Si vous utilisez Vagrant ou un autre environnement virtualisé, tous les dossiers partagés devront être partagés avec des permissions d'exécution (veuillez vous référer à la documentation de votre environnement virtualisé pour savoir comment procéder).

Si, pour une quelconque raison, vous ne pouvez pas changer les permissions du fichier `bin/cake`, vous pouvez lancer la console CakePHP avec la commande suivante :

```
php bin/cake.php
```

Serveur de Développement

Une installation de développement est la méthode la plus rapide pour lancer CakePHP. Dans cet exemple, nous utiliserons la console de CakePHP pour exécuter le serveur web PHP intégré qui va rendre votre application disponible sur **http ://host :port**. A partir du répertoire de l'application, lancez :

```
bin/cake server
```

Par défaut, sans aucun argument fourni, cela rendra accessible votre application sur **http ://localhost :8765/**.

Si vous avez quelque chose qui rentre en conflit avec **localhost** ou le port 8765, vous pouvez dire à la console CakePHP de démarrer le serveur web sur un hôte et/ou un port spécifique utilisant les arguments suivants :

```
bin/cake server -H 192.168.13.37 -p 5673
```

Cela affichera votre application sur **http ://192.168.13.37 :5673/**.

C'est tout ! Votre application CakePHP est lancée sans avoir à configurer un serveur web.

> **Note**
>
> Essayez `bin/cake server -H 0.0.0.0` si le serveur est inaccessible depuis d'autres hôtes.

> **Avertissement**
>
> Ce serveur *n'a pas* vocation à être utilisé, ni ne devrait être utilisé dans un environnement de production. Il est juste à utiliser pour un serveur de développement basique.

Si vous préférez utiliser un vrai serveur web, vous pouvez déplacer votre installation CakePHP (ainsi que les fichiers cachés) dans le document root de votre serveur web. Vous pouvez pointer votre navigateur vers le répertoire dans lequel vous avez déplacé les fichiers et voir votre application en action.

Production

Une installation de production est une façon plus flexible de lancer CakePHP. Utiliser cette méthode permet à tout un domaine d'agir comme une seule application CakePHP. Cet exemple vous aidera à installer CakePHP n'importe où dans votre système de fichiers et à le rendre disponible à l'adresse : http://www.exemple.com. Notez que cette installation demande d'avoir les droits pour modifier le `DocumentRoot` sur le serveur web Apache.

Après avoir installé votre application en utilisant une des méthodes ci-dessus dans un répertoire de votre choix, nous considérerons que vous avez choisi le répertoire /cake_install, votre installation de production devrait ressembler à quelque chose comme ceci dans votre système de fichiers :

```
/cake_install/
     bin/
     config/
     logs/
     plugins/
     resources/
     src/
     templates/
     tests/
     tmp/
     vendor/
     webroot/ (ce répertoire est défini comme DocumentRoot)
     .gitignore
     .htaccess
```

(suite sur la page suivante)

(suite de la page précédente)

```
    .travis.yml
    composer.json
    index.php
    phpunit.xml.dist
    README.md
```

Les développeurs utilisant Apache devront définir la directive `DocumentRoot` pour le domaine à :

```
DocumentRoot /cake_install/webroot
```

Si votre serveur web est correctement configuré, vous devriez maintenant pouvoir accéder à votre application CakePHP à l'adresse http://www.exemple.com.

A vous de jouer !

Ok, regardons CakePHP en action. Selon la configuration que vous utilisez, vous pouvez pointer votre navigateur vers http://exemple.com/ ou http://localhost:8765/. A ce niveau, vous serez sur la page d'accueil par défaut de CakePHP, et un message qui vous donnera le statut de la connexion de votre base de données courante.

Félicitations ! Vous êtes prêt à *créer votre première application CakePHP*.

Réécriture d'URL

Apache

Bien que CakePHP soit conçu par défaut pour fonctionner avec mod_rewrite, et c'est généralement le cas, nous avons remarqué que quelques utilisateurs ont du mal à faire en sorte que tout se passe bien sur leurs systèmes.

Voici quelques choses que vous pourriez essayer pour que cela fonctionne correctement. Premièrement, regardez votre fichier httpd.conf (assurez-vous que vous avez édité le httpd.conf du système plutôt que celui d'un utilisateur ou d'un site spécifique).

Ces fichiers peuvent varier selon les différentes distributions et les versions d'Apache. Vous pouvez consulter https://cwiki.apache.org/confluence/display/httpd/DistrosDefaultLayout pour plus d'informations.

1. Assurez-vous que l'utilisation des fichiers .htaccess est permise et que AllowOverride est défini à All pour le bon DocumentRoot. Vous devriez voir quelque chose comme :

```
# Chaque répertoire auquel Apache a accès peut être configuré en
# fonction des services et fonctionnalités autorisés et/ou
# désactivés dans ce répertoire (et ses sous-répertoires).
#
# Tout d'abord, nous configurons le "défaut" pour qu'il s'agisse
# d'un ensemble très restrictif de fonctionnalités.
#
<Directory />
    Options FollowSymLinks
    AllowOverride All
#    Order deny,allow
#    Deny from all
</Directory>
```

2. Assurez-vous que vous avez chargé correctement mod_rewrite. Vous devriez voir quelque chose comme :

```
LoadModule rewrite_module libexec/apache2/mod_rewrite.so
```

Dans de nombreux systèmes, ces lignes seront commentées par défaut, vous devrez donc simplement supprimer le symbole # en début de ligne.

Après avoir effectué les changements, redémarrez Apache pour être sûr que les paramètres soient effectifs.

Vérifiez que vos fichiers .htaccess sont effectivement dans le bon répertoire.

Vérifiez que vos fichiers .htaccess sont bien dans les bons répertoires. Certains systèmes d'exploitation traitent les fichiers qui commencent par "." comme cachés et ne les copient donc pas.

3. Assurez-vous que votre copie de CakePHP provient de la section téléchargements du site ou de notre dépôt Git, et qu'elle a été décompressée correctement, en vérifiant les fichiers .htaccess.

   Le répertoire app de CakePHP (sera copié dans le répertoire supérieur de votre application par bake) :

```
<IfModule mod_rewrite.c>
   RewriteEngine on
   RewriteRule    ^$    webroot/    [L]
   RewriteRule    (.*) webroot/$1    [L]
</IfModule>
```

   Le répertoire webroot de CakePHP (sera copié dans la racine web de votre application par bake) :

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^ index.php [L]
</IfModule>
```

   Si votre site CakePHP a toujours des problèmes avec mod_rewrite, vous pouvez essayer de modifier les paramètres des Hôtes Virtuels. Sur Ubuntu, éditez le fichier **/etc/apache2/sites-available/default** (l'endroit dépend de la distribution). Dans ce fichier, assurez-vous que `AllowOverride None` a été changé en `AllowOverride All`, donc vous avez :

```
<Directory />
    Options FollowSymLinks
    AllowOverride All
</Directory>
<Directory /var/www>
    Options FollowSymLinks
    AllowOverride All
    Order Allow,Deny
    Allow from all
</Directory>
```

   Sur macOS, une autre solution est d'utiliser l'outil virtualhostx [94] pour créer un Hôte Virtuel pour pointer vers votre dossier.

   Pour de nombreux services d'hébergement (GoDaddy, 1and1), votre serveur web est distribué à partir d'un répertoire utilisateur qui utilise déjà mod_rewrite. Si vous installez CakePHP dans un répertoire utilisateur (http://exemple.com/~username/cakephp/), ou toute autre structure URL qui utilise déjà mod_rewrite, vous aurez devrez ajouter des instructions RewriteBase aux fichiers .htaccess que CakePHP utilise (.htaccess, webroot/.htaccess).

   Ceci peut être ajouté dans la même section que la directive RewriteEngine, par exemple, votre fichier .htaccess

94. https://clickontyler.com/virtualhostx/

dans webroot ressemblerait à :

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteBase /path/to/app
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^ index.php [L]
</IfModule>
```

Les détails de ces changements dépendront de votre configuration, et peuvent inclure des choses supplémentaires qui ne sont pas liées à CakePHP. Veuillez vous référer sur la documentation en ligne d'Apache pour plus d'informations.

4. (Facultatif) Pour améliorer la configuration de production, vous devez empêcher les ressources invalides d'être analysées par CakePHP. Modifiez votre .htaccess dans webroot pour quelque chose comme :

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteBase /path/to/app
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteCond %{REQUEST_URI} !^/(webroot/)?(img|css|js)/(.*)$
    RewriteRule ^ index.php [L]
</IfModule>
```

Ce qui précède empêchera l'envoi de ressources incorrectes à index.php et affichera à la place la page 404 de votre serveur web.

De plus, vous pouvez créer une page HTML 404 correspondante, ou utiliser la page 404 de CakePHP intégrée en ajoutant une directive `ErrorDocument` :

```
ErrorDocument 404 /404-not-found
```

nginx

nginx n'utilise pas les fichiers .htaccess comme Apache, il est donc nécessaire de créer ces URL réécrites dans la configuration disponible sur le site. Ceci se trouve généralement dans `/etc/nginx/sites-available/your_virtual_host_conf_file`. En fonction de votre configuration, vous devrez modifier ceci, mais au minimum, vous aurez besoin de PHP fonctionnant comme une instance FastCGI. La configuration suivante redirige la requête vers `webroot/index.php` :

```
location / {
    try_files $uri $uri/ /index.php?$args;
}
```

Un exemple de la directive server est le suivant :

```
server {
    listen   80;
    listen   [::]:80;
    server_name www.example.com;
    return 301 http://example.com$request_uri;
}

server {
    listen   80;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    listen   [::]:80;
    server_name example.com;

    root   /var/www/example.com/public/webroot;
    index  index.php;

    access_log /var/www/example.com/log/access.log;
    error_log /var/www/example.com/log/error.log;

    location / {
        try_files $uri $uri/ /index.php?$args;
    }

    location ~ \.php$ {
        try_files $uri =404;
        include fastcgi_params;
        fastcgi_pass 127.0.0.1:9000;
        fastcgi_index index.php;
        fastcgi_intercept_errors on;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
    }
}
```

Note

Les configurations récentes de PHP-FPM sont configurées pour écouter le socket unix php-fpm au lieu du port TCP 9000 sur l'adresse 127.0.0.0.1. Si vous avez des erreurs 502 bad gateway avec la configuration ci-dessus, essayez de mettre à jour `fastcgi_pass` pour utiliser le socket unix (ex : fastcgi_pass unix :/var/run/php/php7.1-fpm.sock ;) au lieu du port TCP.

NGINX Unit

NGINX Unit[95] est configurable dynamiquement en runtime ; la configuration suivante repose sur `webroot/index.php`, servant également d'autres Scripts ``.php` s'ils sont présents via `cakephp_direct` :

```
{
    "listeners": {
        "*:80": {
            "pass": "routes/cakephp"
        }
    },

    "routes": {
        "cakephp": [
            {
                "match": {
                    "uri": [
                        "*.php",
                        "*.php/*"
```

(suite sur la page suivante)

95. https://unit.nginx.org

(suite de la page précédente)

```
                    ]
                },

                "action": {
                    "pass": "applications/cakephp_direct"
                }
            },
            {
                "action": {
                    "share": "/path/to/cakephp/webroot/",
                    "fallback": {
                        "pass": "applications/cakephp_index"
                    }
                }
            }
        ]
    },

    "applications": {
        "cakephp_direct": {
            "type": "php",
            "root": "/path/to/cakephp/webroot/",
            "user": "www-data"
        },

        "cakephp_index": {
            "type": "php",
            "root": "/path/to/cakephp/webroot/",
            "user": "www-data",
            "script": "index.php"
        }
    }
}
```

Pour activer cette configuration (en supposant qu'elle soit enregistrée sous `cakephp.json`) :

```
# curl -X PUT --data-binary @cakephp.json --unix-socket \
       /path/to/control.unit.sock http://localhost/config
```

IIS7 (serveurs Windows)

IIS7 ne supporte pas nativement les fichiers .htaccess. Bien qu'il existe des add-ons qui peuvent ajouter ce support, vous pouvez également importer des règles htaccess dans IIS pour utiliser les réécritures natives de CakePHP. Pour ce faire, suivez les étapes suivantes :

1. Utilisez l'installeur de la plateforme Web de Microsoft[96] pour installer l'URL Rewrite Module 2.0[97] ou téléchargez-le directement (32-bit[98] / 64-bit[99]).
2. Créez un nouveau fichier appelé web.config dans votre dossier racine de CakePHP.
3. Utilisez Notepad ou tout autre éditeur XML-safe, copiez le code suivant dans votre nouveau fichier web.config :

96. https://www.microsoft.com/web/downloads/platform.aspx
97. https://www.iis.net/downloads/microsoft/url-rewrite
98. https://download.microsoft.com/download/D/8/1/D81E5DD6-1ABB-46B0-9B4B-21894E18B77F/rewrite_x86_en-US.msi
99. https://download.microsoft.com/download/1/2/8/128E2E22-C1B9-44A4-BE2A-5859ED1D4592/rewrite_amd64_en-US.msi

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <system.webServer>
        <rewrite>
            <rules>
                <rule name="Exclude direct access to webroot/*"
                  stopProcessing="true">
                    <match url="^webroot/(.*)$" ignoreCase="false" />
                    <action type="None" />
                </rule>
                <rule name="Rewrite routed access to assets(img, css, files, js, favicon)
↪"
                  stopProcessing="true">
                    <match url="^(font|img|css|files|js|favicon.ico)(.*)$" />
                    <action type="Rewrite" url="webroot/{R:1}{R:2}"
                      appendQueryString="false" />
                </rule>
                <rule name="Rewrite requested file/folder to index.php"
                  stopProcessing="true">
                    <match url="^(.*)$" ignoreCase="false" />
                    <action type="Rewrite" url="index.php"
                      appendQueryString="true" />
                </rule>
            </rules>
        </rewrite>
    </system.webServer>
</configuration>
```

Une fois que le fichier web.config est créé avec les bonnes règles de réécriture IIS, les liens CakePHP, les CSS, le JavaScript, et le reroutage devraient fonctionner correctement.

Lighttpd

Lighttpd n'utilise pas de fichiers **.htaccess** comme Apache, il est donc nécessaire d'ajouter une configuration `url.rewrite-once` dans **conf/lighttpd.conf**. Assurez-vous que les éléments suivants sont présents dans votre configuration lighthttpd :

```
server.modules += (
    "mod_alias",
    "mod_cgi",
    "mod_rewrite"
)

# Directory Alias
alias.url       = ( "/TestCake" => "C:/Users/Nicola/Documents/TestCake" )

# CGI Php
cgi.assign      = ( ".php" => "c:/php/php-cgi.exe" )

# Rewrite Cake Php (on /TestCake path)
url.rewrite-once = (
    "^/TestCake/(css|files|img|js|stats)/(.*)$" => "/TestCake/webroot/$1/$2",
    "^/TestCake/(.*)$" => "/TestCake/webroot/index.php/$1"
```

(suite sur la page suivante)

(suite de la page précédente)

```
)
```

Les lignes ci-dessus incluent la configuration PHP CGI et un exemple de configuration d'une application pour le chemin `/TestCake`.

Je ne peux pas utiliser la réécriture d'URL

Si vous ne voulez pas ou ne pouvez pas obtenir mod_rewrite (ou un autre module compatible) sur votre serveur, vous devrez utiliser les belles URLs intégrées à CakePHP. Dans **config/app.php**, décommentez la ligne qui ressemble à :

```
'App' => [
    // ...
    // 'baseUrl' => env('SCRIPT_NAME'),
]
```

Supprimez ces fichiers .htaccess :

```
/.htaccess
webroot/.htaccess
```

Vos URLs ressembleront à www.example.com/index.php/controllername/actionname/param plutôt qu'à www.example.com/controllername/actionname/param.

CHAPITRE 7

Configuration

Alors que les conventions enlèvent le besoin de configurer tout CakePHP, vous aurez tout de même besoin de configurer quelques petites choses comme les accès à la base de données.

De plus, certaines options de configuration facultatives vous permettent de changer les valeurs par défaut & les implémentations avec des options qui conviennent à votre application.

Configurer votre Application

La configuration est généralement stockée soit dans les fichiers PHP ou INI, et chargée pendant le bootstrap de l'application. CakePHP est fourni avec un fichier de configuration par défaut, mais si cela et nécessaire, vous pouvez ajouter des fichiers supplémentaires de configuration et les charger dans le bootstrap de votre application. *Cake\Core\Configure* est utilisée pour la configuration globale, et les classes comme `Cache` fournissent les méthodes `setConfig()` pour faciliter la configuration et la rendre plus transparente.

Le squelette de l'application contient un fichier **config/app.php** qui doit contenir la configuration qui ne varie pas selon les différents environnements dans lesquels votre application est déployée. Le fichier **config/app_local.php** doit contenir les données de configuration qui varient selon les environnements et doivent être gérées par la gestion de la configuration ou vos outils de déploiement. Ces deux fichiers font référence à des variables d'environnement via la fonction `env()` qui permet de définir les valeurs de configuration de l'environnement du serveur.

Charger les Fichiers de Configuration Supplémentaires

Si votre application a beaucoup d'options de configuration, il peut être utile de séparer la configuration dans plusieurs fichiers. Après avoir créé chacun des fichiers dans votre répertoire **config/**, vous pouvez les charger dans **bootstrap.php** :

```
use Cake\Core\Configure;
use Cake\Core\Configure\Engine\PhpConfig;

Configure::config('default', new PhpConfig());
```

(suite sur la page suivante)

(suite de la page précédente)

```
Configure::load('app', 'default', false);
Configure::load('other_config', 'default');
```

Variables d'Environnement

Beaucoup de fournisseurs de cloud, comme Heroku, vous permettent de définir des variables pour les données de configuration. Vous pouvez configurer CakePHP via des variables d'environnement à la manière 12factor app [100]. Les variables d'environnement permettent à votre application d'avoir besoin de moins d'états, facilitant la gestion de votre application lors de déploiements sur plusieurs environnements.

Comme vous pouvez le voir dans votre fichier **app.php**, la fonction `env()` est utilisée pour lire des données de configuration depuis l'environnement et construire la configuration de l'application. CakePHP utilise les chaînes *DSN* pour les configurations des bases de données, des logs, des transports d'emails et du cache, vous permettant de faire varier les configurations d'un environnement à l'autre.

Lors d'un développement local, CakePHP utilise dotenv [101] pour permettre l'utilisation des variables d'environnement. Utilisez composer pour ajouter cette bibliothèque, puis décommentez un bloc de code dans `bootstrap.php` pour l'exploiter.

Vous verrez un fichier `config/.env.default` dans votre application. En copiant ce fichier dans `config/.env` et en modifiant les valeurs, vous pourrez configurer votre application.

Il est conseillé de ne pas commiter le fichier `config/.env` dans votre dépôt et d'utiliser le fichier `config/.env.default` comme template avec des valeurs par défaut (ou des placeholders) pour que les membres de votre équipe sachent quelles variables sont utilisées et ce que chaque variable est censée contenir.

Une fois vos variables d'environnement définies, vous pouvez utiliser la fonction `env()` pour lire les données depuis l'environnement :

```
$debug = env('APP_DEBUG', false);
```

La seconde valeur passée à la fonction `env()` est la valeur par défaut. Cette valeur sera utilisée si aucune variable d'environnement n'existe pas pour la clé fournie.

Configuration Générale

Ci-dessous se trouve une description des variables et la façon dont elles modifient votre application CakePHP.

debug
: Change la sortie de debug de CakePHP. `false` = Mode Production. Pas de messages, d'erreurs ou d'avertissements montrés. `true` = Errors et avertissements montrés.

App.namespace
: Le namespace sous lequel se trouvent les classes de l'app.

> **Note**
>
> Quand vous changez le namespace dans votre configuration, vous devez aussi mettre à jour le fichier **composer.json** pour utiliser aussi ce namespace. De plus, créez un nouvel autoloader en lançant `php composer.phar dumpautoload`.

100. https://12factor.net/
101. https://github.com/josegonzalez/php-dotenv

App.baseUrl
Décommentez cette définition si vous **n'** envisagez **pas** d'utiliser le mod_rewrite d'Apache avec CakePHP. N'oubliez pas aussi de retirer vos fichiers .htaccess.

App.base
Le répertoire de base où l'app se trouve. Si à `false`, il sera détecté automatiquement. Si ce n'est pas `false`, assurez-vous que votre chaîne commence avec un `/` et ne se termine PAS par un `/`. Par exemple, `/` *basedir* est une valeur correcte pour App.base. Sinon, le composant AuthComponent ne fonctionnera pas correctement.

App.encoding
Définit l'encodage que votre application utilise. Cet encodage est utilisé pour générer le charset dans le layout, et les entities encodées. Il doit correspondre aux valeurs d'encodage spécifiées pour votre base de données.

App.webroot
Le répertoire webroot.

App.wwwRoot
Le chemin vers webroot.

App.fullBaseUrl
Le nom de domaine complet (y compris le protocole) vers la racine de votre application. Ceci est utilisé pour la génération d'URLS absolues. Par défaut, cette valeur est générée en utilisant la variable d'environnement $_SERVER. Cependant, vous devriez la définir manuellement pour optimiser la performance ou si vous êtes inquiets sur le fait que des gens puissent manipuler le header `Host`. Dans un contexte de CLI (à partir des shells), *fullBaseUrl* ne peut pas être lu dans $_SERVER, puisqu'il n'y a aucun serveur web impliqué. Vous devez le spécifier vous-même si vous avez besoin de générer des URLs à partir d'un shell (par exemple pour envoyer des emails).

App.imageBaseUrl
Le chemin Web vers le répertoire public des images dans webroot. Si vous utilisez un *CDN*, vous devez définir cette valeur vers la localisation du CDN.

App.cssBaseUrl
Le chemin Web vers le répertoire public des css dans webroot. Si vous utilisez un *CDN*, vous devez définir cette valeur vers la localisation du CDN.

App.jsBaseUrl
Le chemin Web vers le répertoire public des js dans webroot. Si vous utilisez un *CDN*, vous devriez définir cette valeur vers la localisation du CDN.

App.paths
Les chemins de Configure pour les ressources non basées sur les classes. Accepte les sous-clés `plugins`, `templates`, `locales`, qui permettent la définition de chemins respectivement pour les plugins, les templates de view et les fichiers de locales.

App.uploadedFilesAsObjects
Définit si les fichiers téléchargés sont représentés sous forme d'objets (`true`), ou de tableaux (`false`). Cette option est considérée comme activée par défaut. Référez-vous à *File Uploads section* dans le chapitre Objets Request & Response pour de plus amples informations.

Security.salt
Une chaîne au hasard utilisée dans les hashages. Cette valeur est aussi utilisée comme sel HMAC quand on fait des chiffrements symétriques.

Asset.timestamp
Ajoute un timestamp qui est le dernier temps modifié du fichier particulier à la fin des URLs des fichiers d'asset (CSS, JavaScript, Image) lors de l'utilisation des helpers adéquats. Valeurs valides :
— (bool) `false` - Ne fait rien (par défaut)
— (bool) `true` - Ajoute le timestamp quand debug est à `true`
— (string) "force" - Ajoute toujours le timestamp.

Asset.cacheTime
Fixe la valeur de la mise en cache des ressources (assets). Elle détermine la valeur du header http `Cache-Control max-age`, ansi que du header http `Expire` pour les ressources. Cela peut-être tout ce que votre version de la fonction strtotime [102] de php accepte. La valeur par défaut est `+1 day`.

102. https://php.net/manual/en/function.strtotime.php

Utilisation d'un CDN

Pour utiliser un CDN pour charger vos actifs statiques, modifiez `App.imageBaseUrl`, `App.cssBaseUrl`, `App.jsBaseUrl` pour pointer vers l'URI du CDN, par exemple : `https://mycdn.example.com/` (notez le `/` à la fin).

A toutes les images, scripts et styles chargés via HtmlHelper sera ajouté le path absolu du CDN, correspondant au même chemin relatif utilisé dans l'application. Notez s'il vous plaît il existe un cas d'utilisation spécifique lors de l'utilisation de ressources basée sur les plugins : les plugins n'utiliseront pas le prefixe de plugin quand l'URI absolue définie dans `... BaseUrl` est utilisée, par exemple par défaut :

— `$this->Helper->assetUrl('TestPlugin.logo.png')` resolves to `test_plugin/logo.png`

Si vous fixez `App.imageBaseUrl` à `https://mycdn.example.com/` :

— `$this->Helper->assetUrl('TestPlugin.logo.png')` resolves to `https://mycdn.example.com/logo.png`.

Configuration de la Base de Données

Regardez la *Configuration de la Base de Données* pour plus d'informations sur la configuration de vos connections à la base de données.

Configuration de la Mise en Cache

Consultez *Configuration du cache* pour plus d'informations sur la configuration de la mise en cache dans CakePHP.

Configuration de Gestion des Erreurs et des Exceptions

Consultez les sections sur *Configuration des erreurs et des exceptions* pour des informations sur la configuration des gestionnaires d'erreur et d'exception.

Configuration des Logs

Consultez *Configuration des flux d'un log (journal)* pour des informations sur la configuration des logs dans CakePHP.

Configuration de Email

Consultez *Configuration des Email* pour avoir des informations sur la configuration prédéfinie d'email dans CakePHP.

Configuration de Session

Consultez *Configuration de Session* pour avoir des informations sur la configuration de la gestion des sessions dans CakePHP.

Configuration du Routing

Consultez *Configuration des Routes* pour plus d'informations sur la configuration du routing et de la création de routes pour votre application.

Chemins de Classe Supplémentaires

Les chemins de classe supplémentaires sont définis dans les autoloaders que votre application utilise. Quand vous utilisez `Composer` pour générer votre autoloader, vous pouvez faire ce qui suit, pour fournir des chemins à utiliser pour les controllers dans votre application :

```
"autoload": {
    "psr-4": {
        "App\\Controller\\": "/path/to/directory/with/controller/folders/",
        "App\\": "src/"
    }
}
```

Ce qui est au-dessus va configurer les chemins pour les namespaces `App` et `App\Controller`. La première clé va être cherchée, et si ce chemin ne contient pas la classe/le fichier, la deuxième clé va être cherchée. Vous pouvez aussi faire correspondre un namespace unique vers plusieurs répertoires avec ce qui suit :

```
"autoload": {
    "psr-4": {
        "App\\": ["src/", "/path/to/directory/"]
    }
}
```

Les chemins de Plugin, View Template et de Locale

Puisque les plugins, view templates et locales ne sont pas des classes, ils ne peuvent pas avoir un autoloader configuré. CakePHP fournit trois variables de configuration pour configurer des chemins supplémentaires pour vos ressources. Dans votre **config/app.php**, vous pouvez définir les variables :

```
return [
    // Plus de configuration
    'App' => [
        'paths' => [
            'plugins' => [
                ROOT . DS . 'plugins' . DS,
                '/path/to/other/plugins/'
            ],
            'templates' => [
                ROOT . DS . 'templates' . DS,
                ROOT . DS . 'templates2' . DS
            ],
            'locales' => [
                ROOT . DS . 'resources' . DS . 'locales' . DS
            ]
        ]
    ]
];
```

Les chemins doivent finir par un séparateur de répertoire, sinon ils ne fonctionneront pas correctement.

Configuration de Inflection

Regardez *Configuration d'Inflexion* pour plus d'informations.

Classe Configure

class Cake\Core**Configure**

La classe Configure de CakePHP peut être utilisée pour stocker et récupérer des valeurs spécifiques d'exécution ou d'application. Attention, cette classe vous permet de stocker tout dedans, puis de l'utiliser dans toute autre partie de votre code : une tentation évidente de casser le modèle MVC avec lequel CakePHP a été conçu. Le but principal de la classe Configure est de garder les variables centralisées qui peuvent être partagées entre beaucoup d'objets. Souvenez-vous d'essayer de suivre la règle « convention plutôt que configuration » et vous ne casserez pas la structure MVC que cakePHP fournit.

Ecrire des Données de Configuration

static Cake\Core\Configure::**write**(*$key*, *$value*)

Utilisez `write()` pour stocker les données de configuration de l'application :

```
Configure::write('Company.name','Pizza, Inc.');
Configure::write('Company.slogan','Pizza for your body and soul');
```

> **Note**
>
> La *notation avec points* utilisée dans le paramètre `$key` peut être utilisée pour organiser vos paramètres de configuration en groupes logiques.

L'exemple ci-dessus pourrait aussi être écrit en un appel unique :

```
Configure::write('Company', [
    'name' => 'Pizza, Inc.',
    'slogan' => 'Pizza for your body and soul'
]);
```

Vous pouvez utiliser `Configure::write('debug', $bool)` pour intervertir les modes de debug et de production à la volée.

> **Note**
>
> Toutes les modifications de configuration effectuées à l'aide de `Configure::write()` se font en mémoire et seront perdues à la requête (request) suivante.

Lire les Données de Configuration

static Cake\Core\Configure::**read**(*$key = null*, *$default = null*)

Utilisé pour lire les données de configuration de l'application. Si une clé est fournie, la donnée est retournée. En utilisant nos exemples pour write() ci-dessus, nous pouvons lire cette donnée :

```
// Renvoie: 'Pizza, Inc.'
Configure::read('Company.name');

// Renvoie: 'Pizza for your body and soul'
Configure::read('Company.slogan');

Configure::read('Company');
// Retourne:
['name' => 'Pizza, Inc.', 'slogan' => 'Pizza for your body and soul'];

// Renvoie 'fallback' car Company.nope nexiste pas.
Configure::read('Company.nope', 'fallback');
```

Si $key est laissée à null, toutes les valeurs dans Configure seront retournées.

static Cake\Core\Configure::**readOrFail**(*$key*)

Permet de lire les données de configuration tout comme *Cake\Core\Configure::read* mais s'attend à trouver une paire clé/valeur. Dans le cas où la paire demandée n'existe pas, une `RuntimeException` sera lancée :

```
Configure::readOrFail('Company.name');    // Renvoie: 'Pizza, Inc.'
Configure::readOrFail('Company.geolocation');  // Lancera un exception

Configure::readOrFail('Company');

//  Renvoie:
['name' => 'Pizza, Inc.', 'slogan' => 'Pizza for your body and soul'];
```

Vérifier si les Données de Configuration sont Définies

static Cake\Core\Configure::**check**(*$key*)

Utilisée pour vérifier si une clé/chemin existe et a une valeur non nulle :

```
$exists = Configure::check('Company.name');
```

Supprimer une Donnée de Configuration

static Cake\Core\Configure::**delete**(*$key*)

Utilisée pour supprimer l'information de la configuration de l'application :

```
Configure::delete('Company.name');
```

Lire & Supprimer les Données de Configuration

static Cake\Core\Configure::**consume**(*$key*)

Lit et supprime une clé de Configure. C'est utile quand vous voulez combiner la lecture et la suppression de valeurs en une seule opération.

static Cake\Core\Configure::**consumeOrFail**(*$key*)

Lit et supprime une donnée de configuration tout comme *Cake\Core\Configure::consume* mais s'attend à trouver une paire clé/valeur. Si la paire demandée n'existe pas une `RuntimeException` sera levée :

```
Configure::consumeOrFail('Company.name');    // Renvoie: 'Pizza, Inc.'
Configure::consumeOrFail('Company.geolocation');  // Lèvera une exception

Configure::consumeOrFail('Company');

// Renvoie:
['name' => 'Pizza, Inc.', 'slogan' => 'Pizza for your body and soul'];
```

Lire et Ecrire les Fichiers de Configuration

static Cake\Core\Configure::**setConfig**(*$name*, *$engine*)

CakePHP est fourni avec deux lecteurs de fichiers de configuration intégrés. `Cake\Core\Configure\Engine\PhpConfig` est capable de lire les fichiers de config de PHP, dans le même format dans lequel Configure a lu historiquement. `Cake\Core\Configure\Engine\IniConfig` est capable de lire les fichiers de config ini du cœur. Regardez la documentation PHP [103] pour plus d'informations sur les spécificités des fichiers ini. Pour utiliser un lecteur de config du cœur, vous aurez besoin de l'attacher à Configure en utilisant `Configure::config()` :

```
use Cake\Core\Configure\Engine\PhpConfig;

// Lire les fichiers de config à partir de config
Configure::config('default', new PhpConfig());

// Lire les fichiers de config à partir du chemin
Configure::config('default', new PhpConfig('/path/to/your/config/files/'));
```

Vous pouvez avoir plusieurs lecteurs attachés à Configure, chacun lisant différents types de fichiers de configuration, ou lisant à partir de différents types de sources. Vous pouvez interagir avec les lecteurs attachés en utilisant certaines autres méthodes de Configure. Pour vérifier les alias qui sont attachés au lecteur, vous pouvez utiliser `Configure::configured()` :

```
// Récupère le tableau d'alias pour les lecteurs attachés.
Configure::configured();

// Vérifie si un lecteur spécifique est attaché
Configure::configured('default');
```

static Cake\Core\Configure::**drop**(*$name*)

Vous pouvez aussi retirer les lecteurs attachés. `Configure::drop('default')` retirerait l'alias du lecteur par défaut. Toute tentative future pour charger les fichiers de configuration avec ce lecteur serait en échec :

```
Configure::drop('default');
```

Chargement des Fichiers de Configuration

static Cake\Core\Configure::**load**(*$key*, *$config = 'default'*, *$merge = true*)

Une fois que vous attachez un lecteur de config à Configure, vous pouvez charger les fichiers de configuration :

103. https://php.net/parse_ini_file

```
// Charge my_file.php en utilisant l'objet de lecture 'default'.
Configure::load('my_file', 'default');
```

Les fichiers de configuration chargés fusionnent leurs données avec la configuration exécutée existante dans Configure. Cela vous permet d'écraser et d'ajouter de nouvelles valeurs dans la configuration existante exécutée. En configurant `$merge` à `true`, les valeurs se combineront à celles de la configuration existante.

Créer et Modifier les Fichiers de Configuration

static `Cake\Core\Configure::`**dump**(*$key*, *$config = 'default'*, *$keys = []*)

Déverse toute ou quelques données de Configure dans un fichier ou un système de stockage supporté par le lecteur. Le format de sérialisation est décidé en configurant le lecteur de config attaché dans $config. Par exemple, si l'adaptateur "default" est `Cake\Core\Configure\Engine\PhpConfig`, le fichier généré sera un fichier de configuration PHP qu'on pourra charger avec `Cake\Core\Configure\Engine\PhpConfig`.

Etant donné que le lecteur "default" est une instance de PhpConfig. Sauvegarder toutes les données de Configure dans le fichier *my_config.php* :

```
Configure::dump('my_config', 'default');
```

Sauvegarde seulement les erreurs gérant la configuration :

```
Configure::dump('error', 'default', ['Error', 'Exception']);
```

`Configure::dump()` peut être utilisée pour soit modifier, soit surcharger les fichiers de configuration qui sont lisibles avec *`Configure::load()`*

Stocker la Configuration de Runtime

static `Cake\Core\Configure::`**store**(*$name*, *$cacheConfig = 'default'*, *$data = null*)

Vous pouvez aussi stocker les valeurs de configuration exécutées pour l'utilisation dans une requête future. Comme configure ne se souvient seulement que des valeurs pour la requête courante, vous aurez besoin de stocker toute information de configuration modifiée si vous souhaitez l'utiliser dans des requêtes suivantes :

```
// Stocke la configuration courante dans la clé 'user_1234' dans le cache 'default'.
Configure::store('user_1234', 'default');
```

Les données de configuration stockées persistent dans la configuration appelée Cache. Consultez la documentation sur *La mise en cache* pour plus d'informations sur la mise en cache.

Restaurer la configuration de runtime

static `Cake\Core\Configure::`**restore**(*$name*, *$cacheConfig = 'default'*)

Une fois que vous avez stocké la configuration exécutée, vous aurez probablement besoin de la restaurer afin que vous puissiez y accéder à nouveau. `Configure::restore()` fait exactement cela :

```
// restaure la configuration exécutée à partir du cache.
Configure::restore('user_1234', 'default');
```

Quand on restaure les informations de configuration, il est important de les restaurer avec la même clé, et la configuration de cache comme elle était utilisée pour les stocker. Les informations restaurées sont fusionnées en haut de la configuration existante exécutée.

Moteurs de Configuration

CakePHP vous permet de charger des configurations provenant de plusieurs sources et formats de données différents et vous donne accès à un système extensible pour créer vos propres moteurs de configuration [104]. Les moteurs inclus dans CakePHP sont :

- JsonConfig [105]
- IniConfig [106]
- PhpConfig [107]

Par défaut, votre application utilisera `PhpConfig`.

Désactiver les tables génériques

Bien qu'utiliser les classes génériques de Table (aussi appeler les « auto-tables ») soit pratique lorsque vous développez rapidement de nouvelles applications, les tables génériques rendent le debug plus difficile dans certains cas.

Vous pouvez vérifier si une requête a été générée à partir d'une table générique via le DebugKit, dans le panneau SQL. Si vous avez encore des difficultés à diagnostiquer un problème qui pourrait être causé par les auto-tables, vous pouvez lancer une exception quand CakePHP utilise implicitement une `Cake\ORM\Table` générique plutôt que la vraie classe du Model :

```
// Dans votre fichier bootstrap.php
use Cake\Event\EventManager;
use Cake\Http\Exception\InternalErrorException;

$isCakeBakeShellRunning = (PHP_SAPI === 'cli' && isset($argv[1]) && $argv[1] === 'bake');
if (!$isCakeBakeShellRunning) {
    EventManager::instance()->on('Model.initialize', function($event) {
        $subject = $event->getSubject();
        if (get_class($subject) === 'Cake\ORM\Table') {
            $msg = sprintf(
                'Missing table class or incorrect alias when registering table class for␣
↪database table %s.',
                $subject->getTable());
            throw new InternalErrorException($msg);
        }
    });
}
```

104. https://api.cakephp.org/4.x/interface-Cake.Core.Configure.ConfigEngineInterface.html
105. https://api.cakephp.org/4.x/class-Cake.Core.Configure.Engine.JsonConfig.html
106. https://api.cakephp.org/4.x/class-Cake.Core.Configure.Engine.IniConfig.html
107. https://api.cakephp.org/4.x/class-Cake.Core.Configure.Engine.PhpConfig.html

CHAPITRE 8

Routing

class Cake\Routing**RouterBuilder**

Le Routing est une fonctionnalité qui fait correspondre les URLs aux actions du controller. En définissant des routes, vous pouvez séparer la façon dont votre application est intégrée de la façon dont ses URLs sont structurées.

Le Routing dans CakePHP englobe aussi l'idée de routing inversé, où un tableau de paramètres peut être transformé en une URL. En utilisant le routing inversé, vous pouvez reconstruire la structure d'URL de votre application sans mettre à jour tous vos codes.

Tour Rapide

Cette section va vous apprendre les utilisations les plus habituelles du Router de CakePHP. Typiquement si vous voulez afficher quelque chose en page d'accueil, vous ajoutez ceci au fichier **config/routes.php** :

```
/** @var \Cake\Routing\RouteBuilder $routes */
$routes->connect('/', ['controller' => 'Articles', 'action' => 'index']);
```

Ceci va exécuter la méthode `index` dans `ArticlesController` quand la page d'accueil de votre site est visitée. Parfois vous avez besoin de routes dynamiques qui vont accepter plusieurs paramètres, ce sera par exemple le cas d'une route pour voir le contenu d'un article :

```
$routes->connect('/articles/*', ['controller' => 'Articles', 'action' => 'view']);
```

La route ci-dessus accepte toute URL qui ressemble à `/articles/15` et appelle la méthode `view(15)` dans `ArticlesController`. En revanche, ceci ne va pas empêcher les visiteurs d'accéder à une URLs ressemblant à `/articles/foobar`. Si vous le souhaitez, vous pouvez restreindre certains paramètres grâce à une expression régulière :

```
// En utilisant l'interface fluide
$routes->connect(
    '/articles/{id}',
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ['controller' => 'Articles', 'action' => 'view'],
)
->setPatterns(['id' => '\d+'])
->setPass(['id']);

// En passant un tableau d'options
$routes->connect(
    '/articles/{id}',
    ['controller' => 'Articles', 'action' => 'view'],
    ['id' => '\d+', 'pass' => ['id']]
);
```

Dans l'exemple précédent, le caractère jocker `*` est remplacé par un placeholder `{id}`. Utiliser les placeholders nous permet de valider les parties de l'URL, dans ce cas, nous utilisons l'expression régulière `\d+` pour que seuls les chiffres fonctionnent. Finalement, nous disons au Router de traiter le placeholder `id` comme un argument de fonction pour la fonction `view()` en spécifiant l'option `pass`. Vous pourrez en voir plus sur leur utilisation plus tard.

Le Router de CakePHP peut aussi faire correspondre les routes en reverse. Cela signifie qu'à partir d'un tableau contenant des paramètres similaires, il est capable de générer une chaîne URL :

```
use Cake\Routing\Router;

echo Router::url(['controller' => 'Articles', 'action' => 'view', 'id' => 15]);
// Va afficher
/articles/15
```

Les routes peuvent aussi être labellisées avec un nom unique, cela vous permet de rapidement leur faire référence lors de la construction des liens plutôt que de spécifier chacun des paramètres de routing :

```
// Dans le fichier routes.php
$routes->connect(
    '/upgrade',
    ['controller' => 'Subscriptions', 'action' => 'create'],
    ['_name' => 'upgrade']
);

use Cake\Routing\Router;

echo Router::url(['_name' => 'upgrade']);
// Va afficher
/upgrade
```

Pour aider à garder votre code de router « DRY », le router apporte le concept de "scopes". Un scope (une étendue) défini un segment de chemin commun, et optionnellement des routes par défaut. Toute route connectée à l'intérieur d'un scope héritera du chemin et des routes par défaut du scope qui la contient :

```
$routes->scope('/blog', ['plugin' => 'Blog'], function (RouteBuilder $routes) {
    $routes->connect('/', ['controller' => 'Articles']);
});
```

Le route ci-dessus matchera `/blog/` et renverra `Blog\Controller\ArticlesController::index()`.

Le squelette d'application contient quelques routes pour vous aider à commencer. Une fois que vous avez ajouté vos propres routes, vous pouvez retirer les routes par défaut si vous n'en avez pas besoin.

Connecter les Routes

Pour garder votre code *DRY*, vous pouvez utiliser les "routing scopes". Les scopes de Routing permettent non seulement de garder votre code DRY mais aident aussi le Router à optimiser son travail. Comme vous l'avez vu précédemment. Cette méthode va par défaut vers le scope `/`. Pour créer un scope et connecter certaines routes, nous allons utiliser la méthode `scope()` :

```
// Dans config/routes.php
use Cake\Routing\RouteBuilder;
use Cake\Routing\Route\DashedRoute;

$routes->scope('/', function (RouteBuilder $routes) {
    // Connect the generic fallback routes.
    $routes->fallbacks(DashedRoute::class);
});
```

La méthode `connect()` prend jusqu'à trois paramètres : l'URL que vous souhaitez faire correspondre, les valeurs par défaut pour les éléments de votre route, et les options de route. Ces options inlcuent fréquemment des règles d'expressions régulières pour aider le router à faire correspondre les éléments dans l'URL.

Le format basique pour une définition de route est :

```
$routes->connect(
    '/url/template',
    ['targetKey' => 'targetValue'],
    ['option' => 'matchingRegex']
);
```

Le premier paramètre est utilisé pour dire au router quelle sorte d'URL vous essayez de contrôler. L'URL est une chaîne normale délimitée par des slashes, mais peut aussi contenir une wildcard (*) ou *Les Eléments de Route*. Utiliser une wildcard dit au router que vous êtes prêt à accepter tout argument supplémentaire fourni. Les Routes sans un * ne matchent que le modèle exact de pattern fourni.

Une fois que vous avez spécifié une URL, vous utilisez les deux derniers paramètres de `connect()` pour dire à CakePHP que faire avec la requête une fois qu'elle a été matchée. La deuxième paramètre définit la route "cible". Il peut être défini soit comme un tableau, soit comme chaîne de destination. Quelques exemples de routes cibles sont :

```
// Cible sous forme de tableau vers un contrôleur de l'application
$routes->connect(
    '/users/view/*',
    ['controller' => 'Users', 'action' => 'view']
);
$routes->connect('/users/view/*', 'Users::view');

// Cible sous forme de tableau vers un contrôleur préfixé de plugin
$routes->connect(
    '/admin/cms/articles',
    ['prefix' => 'Admin', 'plugin' => 'Cms', 'controller' => 'Articles', 'action' =>
↪'index']
);
$routes->connect('/admin/cms/articles', 'Cms.Admin/Articles::index');
```

La première route que nous connectons correspond aux URL commençant par `/users/view` et fait correspondre ces requêtes à `UsersController->view()`. Le dernier `/*` indique au routeur pour passer tous les segments supplémentaires comme arguments de méthode. Par exemple, `/users/view/123` serait mappé à `UsersController->view(123)`.

L'exemple ci-dessus illustre également les chaînes cibles. Les chaînes cibles fournissent une manière compacte de définir la destination d'une route. Les chaînes cibles ont la syntaxe suivante :

```
[Plugin].[Prefix]/[Controller]::[action]
```

Quelques exemples de chaînes cibles sont :

```
// Contrôleur d'application
'Bookmarks::view'

// Contrôleur d'application possédant un préfix
Admin/Bookmarks::view

// Contrôleur de plugin
Cms.Articles::edit

// Contrôleur de plugin possédant un préfix
Vendor/Cms.Management/Admin/Articles::view
```

Auparavant, nous avons utilisé l'étoile greedy (`/*`) pour capturer des segments de chemin supplémentaires, il y aussi la syntaxe de l'étoile trailing (`/**`). Utiliser une étoile double trailing, va capturer le reste de l'URL en tant qu'argument unique passé. Ceci est utile quand vous voulez utilisez un argument qui incluait un `/` dedans :

```
$routes->connect(
    '/pages/**',
    ['controller' => 'Pages', 'action' => 'show']
);
```

L'URL entrante de `/pages/the-example-/-and-proof` résulterait en un argument unique passé `the-example-/-and-proof`.

Vous pouvez utiliser le deuxième paramètre de `connect()` pour fournir tout les paramètres de routing qui formeront alors des valeurs par défaut de la route :

```
$routes->connect(
    '/government',
    ['controller' => 'Pages', 'action' => 'display', 5]
);
```

Cet exemple montre comment vous pouvez utiliser le deuxième paramètre de `connect()` pour définir les paramètres par défaut. Si vous construisez un site qui propose des produits pour différentes catégories de clients, vous pourriez considérer la création d'une route. Cela vous permet de vous lier à `/government` plutôt qu'à `/pages/display/5`.

Une utilisation classique du routing consiste à créer des segments d'URL qui ne correspondent pas aux noms de vos contrôleurs ou de vos modèles. Imaginons qu'au lieu de vouloir accéder à une URL `/users/some_action/5`, vous souhaitiez y accéder via `/cooks/une_action/5`. Pour ce faire, vous devriez configurer la route suivante :

```
$routes->connect(
    '/cooks/{action}/*', ['controller' => 'Users']
);
```

Cela dit au Router que toute URL commençant par `/cooks/` devrait être envoyée au `UsersController`. L'action appelée dépendra de la valeur du paramètre `{action}`. En utilisant *Les Eléments de Route*, vous pouvez créer des routes variables, qui acceptent des entrées utilisateur ou des variables. La route ci-dessus utilise aussi l'étoile greedy. L'étoile greedy indique au `Router` que cette route devrait accepter tout argument de position supplémentaire donné. Ces arguments seront rendus disponibles dans le tableau *Arguments Passés*.

Quand on génère les URLs, les routes sont aussi utilisées. Utiliser `['controller' => 'Users', 'action' => 'some_action', 5]` en URL va sortir `/cooks/some_action/5` si la route ci-dessus est la première correspondante trouvée.

Les routes connectées jusque là fonctionneront avec n'importe quel verbe HTTP. Si vous souhaitez construire une API REST, vous aurez probablement besoin de faire correspondre des actions HTTP à des méthodes de controller différentes. Le `RouteBuilder` met à disposition des méthodes qui rendent plus facile la définition de routes pour des verbes HTTP spécifiques :

```
// Crée une route qui ne répondra qu'aux requêtes GET.
$routes->get(
    '/cooks/{id}',
    ['controller' => 'Users', 'action' => 'view'],
    'users:view'
);

// Crée une route qui ne répondra qu'aux requêtes PUT
$routes->put(
    '/cooks/{id}',
    ['controller' => 'Users', 'action' => 'update'],
    'users:update'
);
```

Les méthodes ci-dessus mappent la même URL à des actions différentes en fonction du verbe HTTP utilisé. Les requêtes GET pointeront sur l'action "view" tandis que les requêtes PUT pointeront sur l'action "update". Les méthodes suivantes sont disponibles pour les verbes :

- GET
- POST
- PUT
- PATCH
- DELETE
- OPTIONS
- HEAD

Toutes ces méthodes retournent une instance de Route ce qui vous permet d'utiliser les *setters fluides* pour configurer plus précisément vos routes.

Les Eléments de Route

Vous pouvez spécifier vos propres éléments de route et ce faisant cela vous donne le pouvoir de définir des emplacements dans l'URL où les paramètres pour les actions du controller doivent se trouver. Quand une requête est faite, les valeurs pour ces éléments de route se trouvent dans `$this->request->getParam()` dans le controller. Quand vous définissez un élément de route personnalisé, vous pouvez spécifier en option une expression régulière - ceci dit à CakePHP comment savoir si l'URL est correctement formée ou non. Si vous choisissez de ne pas fournir une expression régulière, tout caractère autre que `/` sera traité comme une partie du paramètre :

```
$routes->connect(
    '/{controller}/{id}',
    ['action' => 'view']
)->setPatterns(['id' => '[0-9]+']);

$routes->connect(
    '/{controller}/{id}',
    ['action' => 'view'],
    ['id' => '[0-9]+']
);
```

Cet exemple simple montre comment créer une manière rapide de voir les modèles à partir de tout contrôleur en élaborant une URL qui ressemble à `/controllername/{id}`. L'URL fournie à `connect()` spécifie deux éléments de route : `{controller}` et `{id}`. L'élément `{controller}` est l'élément de route par défaut de CakePHP, donc le router sait comment matcher et identifier les noms de contrôleurs dans les URLs. L'élément `{id}` est un élément de route personnalisé, et doit être clarifié plus loin en spécifiant une expression régulière correspondante dans le troisième paramètre de `connect()`.

CakePHP ne produit pas automatiquement d'urls en minuscule avec des tirets quand vous utilisez le paramètre `{controller}`. Si vous avez besoin de ceci, l'exemple ci-dessus peut être réécrit en :

```
use Cake\Routing\Route\DashedRoute;

// Crée un builder avec une classe de Route différente.
$routes->scope('/', function (RouteBuilder $routes) {
    $routes->setRouteClass(DashedRoute::class);
    $routes->connect('/{controller}/{id}', ['action' => 'view'])
        ->setPatterns(['id' => '[0-9]+']);

    $routes->connect(
        '/{controller}/{id}',
        ['action' => 'view'],
        ['id' => '[0-9]+']
    );
});
```

La classe spéciale `DashedRoute` va s'assurer que les paramètres `{controller}` et `{plugin}` sont correctement mis en minuscule et avec des tirets.

> **Note**
>
> Les Patrons utilisés pour les éléments de route ne doivent pas contenir de groupes capturés. S'ils le font, le Router ne va pas fonctionner correctement.

Une fois que cette route a été définie, la requête `/apples/5` est la même que celle requêtant `/apples/view/5`. Les deux appelleraient la méthode `view()` de ApplesController. A l'intérieur de la méthode `view()`, vous aurez besoin d'accéder à l'ID passé à `$this->request->getParam('id')`.

Si vous avez un unique controller dans votre application et que vous ne voulez pas que le nom du controller apparaisse dans l'URL, vous pouvez mapper toutes les URLs aux actions dans votre controller. Par exemple, pour mapper toutes les URLs aux actions du controller `home`, par ex avoir des URLs comme `/demo` à la place de `/home/demo`, vous pouvez faire ce qui suit :

```
$routes->connect('/{action}', ['controller' => 'Home']);
```

Si vous souhaitez fournir une URL non sensible à la casse, vous pouvez utiliser les modificateurs en ligne d'expression régulière :

```
$routes->connect(
    '/{userShortcut}',
    ['controller' => 'Teachers', 'action' => 'profile', 1],
)->setPatterns(['userShortcut' => '(?i:principal)']);
```

Un exemple de plus, et vous serez un pro du routing :

```
$routes->connect(
    '/{controller}/{year}/{month}/{day}',
    ['action' => 'index']
)->setPatterns([
    'year' => '[12][0-9]{3}',
    'month' => '0[1-9]|1[012]',
    'day' => '0[1-9]|[12][0-9]|3[01]'
]);
```

C'est assez complexe, mais montre comme les routes peuvent vraiment devenir puissantes. L'URL fournie a quatre éléments de route. Le premier nous est familier : c'est une route par défaut qui dit à CakePHP d'attendre un nom de controller.

Ensuite, nous spécifions quelques valeurs par défaut. Quel que soit le controller, nous voulons que l'action `index()` soit appelée.

Finalement, nous spécifions quelques expressions régulières qui vont matcher les années, mois et jours sous forme numérique. Notez que les parenthèses (le groupe de capture) ne sont pas supportées dans les expressions régulières. Vous pouvez toujours spécifier des alternatives, comme dessus, mais vous ne pouvez pas les grouper avec les parenthèses.

Une fois définie, cette route va matcher `/articles/2007/02/01`, `/articles/2004/11/16`, gérant les requêtes pour les actions `index()` de leurs controllers respectifs, avec les paramètres de date dans `$this->request->getParam()`.

Eléments de Routes réservés

Il y a plusieurs éléments de route qui ont une signification spéciale dans CakePHP, et ne devraient pas être utilisés à moins que vous ne souhaitiez spécifiquement utiliser leur signification.

- `controller` Utilisé pour nommer le controller pour une route.
- `action` Utilisé pour nommer l'action de controller pour une route.
- `plugin` Utilisé pour nommer le plugin dans lequel un controller est localisé.
- `prefix` Utilisé pour *Prefix de Routage*.
- `_ext` Utilisé pour *Routage des extensions de fichiers*.
- `_base` Défini à `false` pour retirer le chemin de base de l'URL générée. Si votre application n'est pas dans le répertoire racine, cette option peut être utilisée pour générer les URLs qui sont "liées à cake".
- `_scheme` Défini pour créer les liens sur les schémas différents comme *webcal* ou *ftp*. Par défaut, au schéma courant.
- `_host` Définit l'hôte à utiliser pour le lien. Par défaut à l'hôte courant.
- `_port` Définit le port si vous avez besoin de créer les liens sur des ports non-standards.
- `_full` Si à `true`, la valeur de `App.fullBaseUrl` vue dans *Configuration Générale* sera ajoutée devant les URL générées.
- `#` Vous permet de définir les fragments de hash d'URL.
- `_https` Défini à `true` pour convertir l'URL générée à https, ou `false` pour forcer http.
- `_method` Definit la méthode HTTP à utiliser. Utile si vous travaillez avec *Créer des Routes RESTful*.
- `_name` Nom de route. Si vous avez configuré les routes nommées, vous pouvez utiliser cette clé pour les spécifier.

Configurer les Options de Route

Il y a de nombreuses options de routes qui peuvent être définies pour chaque route. Après avoir connecté une route, vous pouvez utiliser ses méthodes de construction fluide pour la configurer. Ces méthodes remplacent la majorité des clés du paramètre `$options` de la méthode `connect()` :

```
$routes->connect(
    '/{lang}/articles/{slug}',
    ['controller' => 'Articles', 'action' => 'view'],
)
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Autorise les requêtes GET & POST.
->setMethods(['GET', 'POST'])

// Match seulement le sous-domaine 'blog'
->setHost('blog.example.com')

// Définit l'élément de la route qui devrait être converti en argument
->setPass(['slug'])

// Définit les patterns de correspondance pour les éléments de route
->setPatterns([
    'slug' => '[a-z0-9-_]+',
    'lang' => 'en|fr|es',
])

// Autorise également l'extension JSON
->setExtenions(['json'])

// Définit 'lang' pour être un paramètre persistant
->setPersist(['lang']);
```

Passer des Paramètres à l'Action

Quand vous connectez les routes en utilisant *Les Eléments de Route* vous voudrez peut-être que des éléments routés soient passés comme arguments à la place. L'option `pass` défini une liste des éléments de route qui doivent également être rendu disponibles en tant qu'arguments passés aux fonctions du contrôleur :

```
// src/Controller/BlogsController.php
public function view($articleId = null, $slug = null)
{
    // du code ici...
}

// routes.php
Router::scope('/', function ($routes) {
    $routes->connect(
        '/blog/{id}-{slug}', // E.g. /blog/3-CakePHP_Rocks
        ['controller' => 'Blogs', 'action' => 'view']
    )
    // Défini les éléments de route dans le template de route
    // à passer en tant qu'arguments à la fonction. L'ordre est
    // important car cela fera simplement correspondre `$id` and `$slug`
    // avec le premier et le second paramètre (respectivement).
    ->setPass(['id', 'slug'])
    // Defini un pattern que `id` doit avoir.
    ->setPatterns([
        'id' => '[0-9]+',
    ]);
});
```

Maintenant, grâce aux possibilités de routing inversé, vous pouvez passer dans le tableau d'URL comme ci-dessous et CakePHP sait comment former l'URL comme définie dans les routes :

```
// view.php
// ceci va retourner un lien vers /blog/3-CakePHP_Rocks
echo $this->Html->link('CakePHP Rocks', [
    'controller' => 'Blog',
    'action' => 'view',
    'id' => 3,
    'slug' => 'CakePHP_Rocks'
]);

// Vous pouvez aussi utiliser des paramètres indexés numériquement.
echo $this->Html->link('CakePHP Rocks', [
    'controller' => 'Blog',
    'action' => 'view',
    3,
    'CakePHP_Rocks'
]);
```

Utilisation du Routage de Chemin

Nous avons parlé des cibles de chaîne ci-dessus. La même chose fonctionne également pour la génération d'URL en utilisant `Router::pathUrl()` :

```
echo Router::pathUrl('Articles::index');
// donnera par exemple: /articles

echo Router::pathUrl('MyBackend.Admin/Articles::view', [3]);
// donnera par exemple: /admin/my-backend/articles/view/3
```

Astuce

Le support IDE pour la saisie semi-automatique du routage de chemin peut être activé avec CakePHP IdeHelper Plugin [108].

Utiliser les Routes Nommées

Parfois vous trouvez que taper tous les paramètres de l'URL pour une route est trop verbeux, ou bien vous souhaitez tirer avantage des améliorations de la performance que les routes nommées permettent. Lorsque vous connectez les routes, vous pouvez spécifier une option `_name`, cette option peut être utilisée par le routing inversé pour identifier la route que vous souhaitez utiliser :

```
// Connecter une route avec un nom.
$routes->connect(
    '/login',
    ['controller' => 'Users', 'action' => 'login'],
    ['_name' => 'login']
);

// Nommage d'une route liée à un verbe spécifique
$routes->post(
    '/logout',
```

(suite sur la page suivante)

108. https://github.com/dereuromark/cakephp-ide-helper

(suite de la page précédente)

```
    ['controller' => 'Users', 'action' => 'logout'],
    'logout'
);

// Génère une URL en utilisant une route nommée.
$url = Router::url(['_name' => 'logout']);

// Génère une URL en utilisant une route nommée,
// avec certains args query string
$url = Router::url(['_name' => 'login', 'username' => 'jimmy']);
```

Si votre template de route contient des éléments de route comme `{controller}`, vous aurez besoin de fournir ceux-ci comme options de `Router::url()`.

> **Note**
>
> Les noms de Route doivent être uniques pour l'ensemble de votre application. Le même `_name` ne peut être utilisé deux fois, même si les noms apparaissent dans un scope de routing différent.

Quand vous construisez vos noms de routes, vous voudrez probablement coller à certaines conventions pour les noms de route. CakePHP facilite la construction des noms de route en vous permettant de définir des préfixes de nom dans chaque scope :

```
$routes->scope('/api', ['_namePrefix' => 'api:'], function (RouteBuilder $routes) {
    // le nom de cette route sera `api:ping`
    $routes->get('/ping', ['controller' => 'Pings'], 'ping');
});
// Génère une URL correspondant à la route 'ping'
Router::url(['_name' => 'api:ping']);

// Utilisation du namePrefix avec plugin()
$routes->plugin('Contacts', ['_namePrefix' => 'contacts:'], function (RouteBuilder
↪$routes) {
    // Connecte les routes.
});

// Ou avec prefix()
$routes->prefix('Admin', ['_namePrefix' => 'admin:'], function (RouteBuilder $routes) {
    // Connecte les routes.
});
```

Vous pouvez aussi utiliser l'option `_namePrefix` dans les scopes imbriqués et elle fonctionne comme vous pouvez vous y attendre :

```
$routes->plugin('Contacts', ['_namePrefix' => 'contacts:'], function (RouteBuilder
↪$routes) {
    $routes->scope('/api', ['_namePrefix' => 'api:'], function (RouteBuilder $routes) {
        // Le nom de cette route sera `contacts:api:ping`
        $routes->get('/ping', ['controller' => 'Pings'], 'ping');
    });
});
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Génère une URL correspondant à la route 'ping'
Router::url(['_name' => 'contacts:api:ping']);
```

Les routes connectées dans les scopes nommés auront seulement des noms ajoutés si la route est aussi nommée. Les routes sans nom ne se verront pas appliquées `_namePrefix`.

Prefix de Routage

static Cake\Routing\RouterBuilder::**prefix**(*$name*, *$callback*)

De nombreuses applications nécessitent une section d'administration dans laquelle les utilisateurs privilégiés peuvent faire des modifications. Ceci est souvent réalisé grâce à une URL spéciale telle que `/admin/users/edit/5`. Dans CakePHP, les préfixes de routage peuvent être activés en utilisant la méthode de portée (scope) `prefix` :

```
use Cake\Routing\Route\DashedRoute;

$routes->prefix('Admin', function (RouteBuilder $routes) {
    // Toutes les routes ici seront préfixées avec `/admin`, et
    // l'élément de route `'prefix' => 'Admin'` sera ajouté qui
    // sera requis lors de la génération d'URL pour ces routes
    $routes->fallbacks(DashedRoute::class);
});
```

Les préfixes sont mappés aux sous-espaces de noms dans l'espace de nom `Controller` de votre application. En ayant des préfixes en tant que controller séparés, vous pouvez créer des contrôleurs plus petits et/ou plus simples. Les comportements communs aux controllers préfixés et non-préfixés peuvent être encapsulés via l'héritage, les *Components (Composants)*, ou les traits. En utilisant notre exemple des utilisateurs, accéder à l'url `/admin/users/edit/5` devrait appeler la méthode `edit()` de notre `App\Controller\Admin\UsersController` en passant 5 comme premier paramètre. Le fichier de vue utilisé serait **templates/Admin/Users/edit.php**.

Vous pouvez faire correspondre l'URL /admin à votre action `index()` du controller Pages en utilisant la route suivante :

```
$routes->prefix('Admin', function (RouteBuilder $routes) {
    // Parce que vous êtes dans le scope admin, vous n'avez pas besoin
    // d'inclure le prefix /admin ou l'élément de route admin.
    $routes->connect('/', ['controller' => 'Pages', 'action' => 'index']);
});
```

Quand vous créez des routes préfixées, vous pouvez définir des paramètres de route supplémentaires en utilisant l'argument `$options` :

```
$routes->prefix('Admin', ['param' => 'value'], function (RouteBuilder $routes) {
    // Routes connectées ici sont préfixées par '/admin' et
    // ont la clé 'param' de routing définie.
    $routes->connect('/{controller}');
});
```

Les préfixes de plusieurs mots sont par défaut convertis en utilisant l'inflexion en tirets (dasherize), c'est-à-dire que `MyPrefix` serait mappé sur `my-prefix` dans l'URL. Assurez-vous de définir un chemin d'accès pour ces préfixes si vous souhaitez utiliser un format différent comme par exemple le soulignement :

```
$routes->prefix('MyPrefix', ['path' => '/my_prefix'], function (RouteBuilder $routes) {
    // Les routes connectées ici sont préfixées par '/my_prefix'
```

(suite sur la page suivante)

(suite de la page précédente)

```
    $routes->connect('/{controller}');
});
```

Vous pouvez aussi définir les préfixes dans les scopes de plugin :

```
$routes->plugin('DebugKit', function (RouteBuilder $routes) {
    $routes->prefix('Admin', function (RouteBuilder $routes) {
        $routes->connect('/{controller}');
    });
});
```

Ce qui est au-dessus va créer un template de route de type `/debug-kit/admin/{controller}`. La route connectée aura les éléments de route `plugin` et `prefix` définis.

Quand vous définissez des préfixes, vous pouvez imbriquer plusieurs préfixes si besoin :

```
$routes->prefix('Manager', function (RouteBuilder $routes) {
    $routes->prefix('Admin', function (RouteBuilder $routes) {
        $routes->connect('/{controller}/{action}');
    });
});
```

Ce qui est au-dessus va créer un template de route de type `/manager/admin/{controller}/{action}`. La route connectée aura l'élément de route `prefix` défini à `Manager/Admin`.

Le préfixe actuel sera disponible à partir des méthodes du controller avec `$this->request->getParam('prefix')`

Quand vous utilisez les routes préfixées, il est important de définir l'option `prefix`, et d'utiliser le même format CamelCased que celui utilisé dans la méthode `prefix()`. Voici comment construire ce lien en utilisant le helper HTML :

```
// Aller vers une route préfixée.
echo $this->Html->link(
    'Manage articles',
    ['prefix' => 'Manager/Admin', 'controller' => 'Articles', 'action' => 'add']
);

// Enlever un prefix
echo $this->Html->link(
    'View Post',
    ['prefix' => false, 'controller' => 'Articles', 'action' => 'view', 5]
);
```

Note

Vous devez connecter les routes préfixées *avant* de connecter les routes fallback.

Création de liens vers des routes de préfixe

Vous pouvez créer des liens qui pointent vers un préfixe, en ajoutant la clé de préfixe à votre tableau d'URL :

```
echo $this->Html->link(
    'New admin todo',
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ['prefix' => 'Admin', 'controller' => 'TodoItems', 'action' => 'create']
);
```

Lorsque vous utilisez l'imbrication, vous devez les chaîner ensemble :

```
echo $this->Html->link(
    'New todo',
    ['prefix' => 'Admin/MyPrefix', 'controller' => 'TodoItems', 'action' => 'create']
);
```

Cela serait lié à un contrôleur avec l'espace de noms `App\Controller\Admin\MyPrefix` et le chemin de fichier `src/Controller/Admin/MyPrefix/TodoItemsController.php`.

> **Note**
>
> Le préfixe est CamelCased ici, même si le résultat du routage est en pointillés. La route elle-même fera l'inflexion si nécessaire.

Routing des Plugins

static `Cake\Routing\RouterBuilder::`**`plugin`**(*$name*, *$options = []*, *$callback*)

Les routes des *Plugins* doivent être créées en utilisant la méthode `plugin()`. Cette méthode crée un nouveau scope pour les routes de plugin :

```
$routes->plugin('DebugKit', function (RouteBuilder $routes) {
    // Les routes connectées ici sont préfixées par '/debug_kit' et ont
    // l'élément de route plugin défini à 'DebugKit'.
    $routes->connect('/{controller}');
});
```

Lors de la création des scopes de plugin, vous pouvez personnaliser le chemin de l'élément avec l'option `path` :

```
$routes->plugin('DebugKit', ['path' => '/debugger'], function (RouteBuilder $routes) {
    // Les routes connectées ici sont préfixées par '/debugger' et ont
    // l'élément de route plugin défini à 'DebugKit'.
    $routes->connect('/{controller}');
});
```

Lors de l'utilisation des scopes, vous pouvez imbriquer un scope de plugin dans un scope de prefix :

```
$routes->prefix('Admin', function (RouteBuilder $routes) {
    $routes->plugin('DebugKit', function (RouteBuilder $routes) {
        $routes->connect('/{controller}');
    });
});
```

Le code ci-dessus va créer une route similaire à `/admin/debug-kit/{controller}`. Elle aura les éléments de route `prefix` et `plugin` définis. Référez-vous à la section *Routes de Plugins* pour avoir plus d'informations sur comment construire des routes de plugin.

Créer des Liens vers des Routes de Plugins

Vous pouvez créer des liens qui pointent vers un plugin, en ajoutant la clé `plugin` au tableau de l'URL :

```
echo $this->Html->link(
    'New todo',
    ['plugin' => 'Todo', 'controller' => 'TodoItems', 'action' => 'create']
);
```

Inversement, si la requête active est une requête de plugin et que vous souhaitez créer un lien qui n'a pas de plugin, vous pouvez faire ceci :

```
echo $this->Html->link(
    'New todo',
    ['plugin' => null, 'controller' => 'Users', 'action' => 'profile']
);
```

En définissant `'plugin' => null`, vous dites au Router que vous souhaitez créer un lien qui n'appartient pas à un plugin.

Routing Favorisant le SEO

Certains développeurs préfèrent utiliser des tirets dans les URLs, car cela semble donner un meilleur classement dans les moteurs de recherche. La classe `DashedRoute` fournit à votre application la possibilité de créer des URLs avec des tirets pour vos plugins, contrôleurs, et les noms d'action en `camelCase`.

Par exemple, si nous avons un plugin `ToDo` avec un controller `TodoItems` et une action `showItems()`, la route générée sera `/to-do/todo-items/show-items` avec le code qui suit :

```
use Cake\Routing\Route\DashedRoute;

$routes->plugin('ToDo', ['path' => 'to-do'], function (RouteBuilder $routes) {
    $routes->fallbacks(DashedRoute::class);
});
```

Matching des Méthodes HTTP Spécifiques

Les routes peuvent « matcher » des méthodes HTTP spécifiques en utilisant les méthodes spécifiques :

```
$routes->scope('/', function (RouteBuilder $routes) {
    // Cette route matchera seulement les requêtes POST.
    $routes->post(
        '/reviews/start',
        ['controller' => 'Reviews', 'action' => 'start']
    );

    // Matcher plusieurs verbes
    $routes->connect(
        '/reviews/start',
        [
            'controller' => 'Reviews',
            'action' => 'start',
        ]
    )->setMethods(['POST', 'PUT']);
});
```

Vous pouvez « matcher » plusieurs méthodes HTTP en fournissant un tableau. Puisque que l'option `_method` est une clé de routage, elle est utilisée à la fois dans le parsing des URL et la génération des URL. Pour générer des URL pour des routes spécifiques, vous devez utiliser la clé `_method` lors de la génération :

```
$url = Router::url([
    'controller' => 'Reviews',
    'action' => 'start',
    '_method' => 'POST',
]);
```

Matching de Noms de Domaine Spécifiques

Les routes peuvent utiliser l'option `_host` pour « matcher » des noms de domaines spécifiques. Vous pouvez utiliser la wildcard `*.` pour « matcher » n'importe quelle sous-domaine :

```
$routes->scope('/', function (RouteBuilder $routes) {
    // Cette route ne va "matcher" que sur le domaine http://images.example.com
    $routes->connect(
        '/images/default-logo.png',
        ['controller' => 'Images', 'action' => 'default']
    )->setHost('images.example.com');

    // Cette route matchera sur tous les sous-domaines http://*.example.com
    $routes->connect(
        '/images/old-logo.png',
        ['controller' => 'Images', 'action' => 'oldLogo']
    )->setHost('*.example.com');
});
```

L'option `_host` est également utilisée dans la génération d'URL. Si votre option `_host` spécifie un domaine exact, ce domaine sera inclus dans l'URL générée. Cependant, si vous utilisez un caractère générique, vous devrez fournir le `_host` paramètre lors de la génération d'URL :

```
// Si vous avez cette route
$routes->connect(
    '/images/old-logo.png',
    ['controller' => 'Images', 'action' => 'oldLogo']
)->setHost('images.example.com');

// Vous aurez besoin de ceci pour générer l'URL correspondante
echo Router::url([
    'controller' => 'Images',
    'action' => 'oldLogo',
    '_host' => 'images.example.com',
]);
```

Routing des Extensions de Fichier

static `Cake\Routing\RouterBuilder::`**extensions**(*string|array|null $extensions*, *$merge = true*)

Pour manipuler différentes extensions de fichier avec vos routes, vous pouvez définir vos extensions en utilisant la méthode `Cake\Routing\RouteBuilder::setExtensions()` :

```
$routes->scope('/', function (RouteBuilder $routes) {
    $routes->setExtensions(['json', 'xml']);
});
```

Ceci affectera **toutes** les routes qui seront connectées **après** cet appel, à `setExtensions()` en incluant celles qui ont été connectées dans des scopes imbriqués.

Pour restreindre les extensions à un *scope* spécifique, vous pouvez les définir en utilisant la méthode `Cake\Routing\RouteBuilder::extensions()`.

> **Note**
>
> Le réglage des extensions devrait être la première chose que vous devriez faire dans un scope, car les extensions seront appliquées uniquement aux routes qui sont définies **après** la déclaration des extensions.
>
> Lorsque vous définissez des routes dans le même scope mais dans deux appels différents, les extensions ne seront pas héritées d'un appel à l'autre.

En utilisant des extensions, vous dites au router de supprimer toutes les extensions de fichiers correspondant, puis d'analyser le reste. Si vous souhaitez créer une URL comme `/page/title-of-page.html` vous devriez créer un scope comme ceci :

```
$routes->scope('/page', function (RouteBuilder $routes) {
    $routes->setExtensions(['json', 'xml', 'html']);
    $routes->connect(
        '/{title}',
        ['controller' => 'Pages', 'action' => 'view']
    )->setPass(['title']);
});
```

Ensuite, pour créer des liens, utilisez simplement :

```
$this->Html->link(
    'Link title',
    ['controller' => 'Pages', 'action' => 'view', 'title' => 'super-article', '_ext' =>
↪'html']
);
```

Les extensions de fichier sont utilisées par le *Request Handling (Gestion des requêtes)* qui fait la commutation des vues automatiquement en se basant sur les types de contenu.

Middleware appliqué à une Route

Bien que les middlewares puissent être appliqués à toute votre application, appliquer les middlewares à des "scopes" de routing offre plus de flexibilité puisque vous pouvez appliquer des middlewares seulement où ils sont nécessaires permettant à vos middlewares de ne pas nécessiter de logique spécifique sur le comment/où il doit s'appliquer.

> **Note**
>
> Le middleware appliqué sera exécuté par RoutingMiddleware, normalement à la fin de la liste des middleware de votre application.

Avant qu'un middleware ne puisse être appliqué à un scope, il a besoin d'être enregistré dans la collection de routes :

```
// dans config/routes.php
use Cake\Http\Middleware\CsrfProtectionMiddleware;
use Cake\Http\Middleware\EncryptedCookieMiddleware;

$routes->scope('/', function (RouteBuilder $routes) {
    $routes->registerMiddleware('csrf', new CsrfProtectionMiddleware());
    $routes->registerMiddleware('cookies', new EncryptedCookieMiddleware());
});
```

Une fois enregistré, le middleware peut être appliqué à des scopes spécifiques :

```
$routes->scope('/cms', function (RouteBuilder $routes) {
    // Enable CSRF & cookies middleware
    $routes->applyMiddleware('csrf', 'cookies');
    $routes->get('/articles/{action}/*', ['controller' => 'Articles'])
});
```

Dans le cas où vous auriez des "scopes" imbriqués, les « sous » scopes hériteront des middlewares apppliqués dans le scope contenant :

```
$routes->scope('/api', function (RouteBuilder $routes) {
    $routes->applyMiddleware('ratelimit', 'auth.api');
    $routes->scope('/v1', function (RouteBuilder $routes) {
        $routes->applyMiddleware('v1compat');
        // Définissez vos routes
    });
});
```

Dans l'exemple ci-dessus, les routes définies dans `/v1` auront les middlewares "ratelimit", "auth.api", and "v1compat" appliqués. Si vous ré-ouvrez un scope, les middlewares appliqués aux routes dans chaque scopes seront isolés :

```
$routes->scope('/blog', function (RouteBuilder $routes) {
    $routes->applyMiddleware('auth');
    // Connecter les actions qui nécessitent l'authentification aux 'blog' ici
});
$routes->scope('/blog', function (RouteBuilder $routes) {
    // Connecter les actions publiques pour le 'blog' ici
});
```

Dans l'exemple ci-dessus, les 2 utilisations du scope `/blog` ne partagent pas les middlewares. Par contre, les 2 scopes hériteront des middlewares définis dans le scope qui les contient.

Grouper les Middlewares

Pour vous aider à garder votre code DRY (Do not Repeat Yourself), les middlewares peuvent être combinés en groupes. Une fois créés, les groupes peuvent être appliqués comme des middlewares :

```
$routes->registerMiddleware('cookie', new EncryptedCookieMiddleware());
$routes->registerMiddleware('auth', new AuthenticationMiddleware());
$routes->registerMiddleware('csrf', new CsrfProtectionMiddleware());
$routes->middlewareGroup('web', ['cookie', 'auth', 'csrf']);

// Appliquer le groupe
$routes->applyMiddleware('web');
```

Créer des Routes RESTful

Le router aide à générer des routes RESTful pour vos controllers. Les routes RESTful sont utiles lorsque vous créez des points de terminaison (endpoint) d'API pour vos applications. Si nous voulions autoriser l'accès REST à un contrôleur de recette, nous ferions quelque chose comme ceci :

```
//Dans config/routes.php

 $routes->scope('/', function (RouteBuilder $routes) {
    $routes->setExtensions(['json']);
    $routes->resources('Recipes');
});
```

La première ligne définit un certain nombre de routes par défaut pour l'accès REST où la méthode spécifie le format du résultat souhaité (par exemple, xml, json, rss). Ces routes sont sensibles aux méthodes de requêtes HTTP.

HTTP format	URL.format	Action du contrôleur appelée
GET	/recipes.format	RecipesController : :index()
GET	/recipes/123.format	RecipesController : :view(123)
POST	/recipes.format	RecipesController : :add()
PUT	/recipes/123.format	RecipesController : :edit(123)
PATCH	/recipes/123.format	RecipesController : :edit(123)
DELETE	/recipes/123.format	RecipesController : :delete(123)

Note

La valeur par défaut du modèle pour les ID de ressource ne reconnait que des entiers ou des UUID. Si vos ID sont différents, vous devrez fournir une expression régulière via l'option `id`. Par exemple. `$builder->resources('Recettes', ['id' => '. *']).`

La classe Router de CakePHP utilise un nombre différent d'indicateurs pour détecter la méthode HTTP utilisée. Voici la liste dans l'ordre de préférence :

1. La variable de POST `_method`
2. Le header `X_HTTP_METHOD_OVERRIDE`
3. Le header `REQUEST_METHOD`

La variable POST `_method` est utile dans l'utilisation d'un navigateur comme client REST (ou tout ce qui peut faire du POST). Il suffit de configurer la valeur de `_method` avec le nom de la méthode de requête HTTP que vous souhaitez émuler.

Créer des Routes de Ressources Imbriquées

Une fois que vous avez connecté une ressource dans un scope, vous pouvez aussi connecter des routes pour des sous-ressources. Les routes de sous-ressources seront préfixées par le nom de la ressource originale et par son paramètre id. Par exemple :

```
$routes->scope('/api', function (RouteBuilder $routes) {
    $routes->resources('Articles', function (RouteBuilder $routes) {
        $routes->resources('Comments');
    });
});
```

Le code ci-dessus va générer une ressource de route pour `articles` et `comments`. Les routes des `comments` vont ressembler à ceci :

```
/api/articles/{article_id}/comments
/api/articles/{article_id}/comments/{id}
```

Vous pouvez récupérer le champs `article_id` de `CommentsController` de cette façon :

```
$this->request->getParam('article_id');
```

Par défaut les ressources de routes sont connectées au même préfixe que celles de leur scope. Si vous avez à la fois des contrôleurs de ressources imbriqués et non imbriqués, vous pouvez utiliser un contrôleur différent dans chaque contexte en utilisant des préfixes :

```
$routes->scope('/api', function (RouteBuilder $routes) {
    $routes->resources('Articles', function (RouteBuilder $routes) {
        $routes->resources('Comments', ['prefix' => 'Articles']);
    });
});
```

L'exemple ci-dessus mapperait le champs "Comments" vers `App\Controller\Articles\CommentsController`. Une séparation des contrôleurs vous permet de simplifier la logique. Les préfixes créés de cette manière sont compatibles avec *Prefix de Routage*.

Note

Vous pouvez imbriquer autant de ressources que vous le souhaitez, mais il n'est pas recommandé d'imbriquer plus de 2 ressources ensembles.

Limiter la Création des Routes

Par défaut, CakePHP va connecter 6 routes pour chaque ressource. Si vous souhaitez connecter uniquement des routes spécifiques à une ressource, vous pouvez utiliser l'option `only` :

```
$routes->resources('Articles', [
    'only' => ['index', 'view']
]);
```

Le code ci-dessus devrait créer uniquement les routes de ressource pour la lecture. Les noms de route sont `create`, `update`, `view`, `index` et `delete`.

Changer les Actions du Controller

Vous devrez peut-être modifier le nom des actions du controller qui sont utilisés lors de la connexion des routes. Par exemple, si votre action `edit()` est nommée `put()`, vous pouvez utiliser la clé `actions` pour renommer vos actions :

```
$routes->resources('Articles', [
    'actions' => ['update' => 'put', 'create' => 'add']
]);
```

Le code ci-dessus va utiliser la méthode `put()` pour l'action `edit()`, et `add()` au lieu de `create()`.

Mapper des Routes de Ressource Supplémentaires

Vous pouvez mapper des méthodes de ressource supplémentaires en utilisant l'option `map` :

```
$routes->resources('Articles', [
   'map' => [
       'deleteAll' => [
           'action' => 'deleteAll',
           'method' => 'DELETE'
       ]
   ]
]);
// Ceci connecterait /articles/deleteAll
```

En plus des routes par défaut, ceci connecterait aussi une route pour */articles/delete-all*. Par défaut le segment de chemin va matcher le nom de la clé. Vous pouvez utiliser la clé "path" à l'intérieur de la définition de la ressource pour personnaliser le nom de chemin :

```
$routes->resources('Articles', [
    'map' => [
        'updateAll' => [
            'action' => 'updateAll',
            'method' => 'PUT',
            'path' => '/update-many'
        ],
    ]
]);
// Ceci connecterait /articles/update-many
```

Si vous définissez "only" et "map", assurez-vous que vos méthodes mappées sont aussi dans la liste "only".

Router vers une Ressource Préfixée

Les routes vers des ressources peuvent être connectées à des controllers ayant des préfixes de routage en connectant des routes à l'intérieur d'un scope préfixé, ou en utilisant l'option `prefix` :

```
$routes->resources('Articles', [
    'prefix' => 'Api',
]);
```

Classes de Route Personnalisée pour les Ressources

Vous pouvez spécifier la clé `connectOptions` dans le tableau `$options` de la fonction `resources()` pour fournir une configuration personnalisée utilisée par `connect()` :

```
$routes->scope('/', function (RouteBuilder $routes) {
    $routes->resources('Books', [
        'connectOptions' => [
            'routeClass' => 'ApiRoute',
        ]
    ];
});
```

Inflection de l'URL pour les Routes Ressource

Par défaut le fragment d'URL pour les contrôleurs dont le nom est composé de plusieurs mots est la forme en underscore du nom du controller. Par exemple, le fragment d'URL pour `BlogPosts` serait **/blog-posts**.

Vous pouvez spécifier un type d'inflection alternatif en utilisant l'option `inflect` :

```
$routes->scope('/', function (RouteBuilder $routes) {
    $routes->resources('BlogPosts', [
        'inflect' => 'underscore' // Utilisera ``Inflector::underscore()``
    ]);
})
```

Ce qui est au-dessus va générer des URLs de style **/blog_posts***.

Changer le chemin d'un élément

Par défaut, les ressources de routes utilisent le nom de ressource ayant subi une inflexion en guise de segment d'URL. Vous pouvez définir un segment d'URL personnalisé à l'aide de l'option `path` :

```
$routes->scope('/', function (RouteBuilder $routes) {
    $routes->resources('BlogPosts', ['path' => 'posts']);
});
```

Arguments Passés

Les arguments passés sont des arguments supplémentaires ou des segments du chemin qui sont utilisés lors d'une requête. Ils sont souvent utilisés pour transmettre des paramètres aux méthodes de vos controllers :

```
http://localhost/calendars/view/recent/mark
```

Dans l'exemple ci-dessus, `recent` et `mark` sont tous deux des arguments passés à `CalendarsController::view()`. Les arguments passés sont transmis aux contrôleurs de trois manières. D'abord comme arguments de la méthode de l'action appelée, deuxièmement en étant accessibles dans `$this->request->getParam('pass')` sous la forme d'un tableau indexé numériquement. Enfin, il y a `$this->passedArgs` disponible de la même façon que par `$this->request->getParam('pass')`. Lorsque vous utilisez des routes personnalisées, il est également possible de forcer des paramètres particuliers comme étant des paramètres passés.

Si vous alliez visiter l'URL mentionné précédemment, et que vous aviez une action de contrôleur qui ressemblait à cela :

```
class CalendarsController extends AppController
{
    public function view($arg1, $arg2)
    {
        debug(func_get_args());
    }
}
```

Vous auriez le résultat suivant :

```
Array
(
    [0] => recent
```

(suite sur la page suivante)

(suite de la page précédente)

```
    [1] => mark
)
```

La même donnée est aussi disponible dans `$this->request->getParam('pass')` dans vos contrôleurs, vues, et helpers. Les valeurs dans le tableau pass sont indicées numériquement basé sur l'ordre dans lequel elles apparaissent dans l'URL appelée :

```
debug($this->request->getParam('pass'));
```

Le résultat des 2 debug() du dessus serait :

```
Array
(
    [0] => recent
    [1] => mark
)
```

Quand vous générez des URLs, en utilisant un *tableau de routing*, vous ajoutez des arguments passés en valeurs sans clés de type chaîne dans le tableau :

```
['controller' => 'Articles', 'action' => 'view', 5]
```

Comme 5 poss !de une clé numérique, il est traité comme un argument passé.

Générer des URLs

static Cake\Routing\RouterBuilder::**url**(*$url = null*, *$full = false*)

static Cake\Routing\RouterBuilder::**reverse**(*$params*, *$full = false*)

La génération d'URL ou le routing inversé est une fonctionnalité dans CakePHP qui est utilisée pour vous permettre de changer votre structure d'URL sans avoir à modifier tout votre code.

Si vous créez des URLs en utilisant des chaînes de caractères comme :

```
$this->Html->link('View', '/articles/view/' . $id);
```

Et ensuite plus tard, vous décidez que `/articles` devrait vraiment être appelé "posts" à la place, vous devrez aller dans toute votre application en renommant les URLs. Cependant, si vous définissiez votre lien comme :

```
//`link()` utilise Router::url() en interne et accepte un tableau de routage

$this->Html->link(
    'View',
    ['controller' => 'Articles', 'action' => 'view', $id]
);
```

ou :

```
//'Router::reverse()' fonctionne sur le tableau de paramètres de requête
//et produira une entrée valide pour la méthode `link()` : une url
//sous forme de chaîne de caractères.
```

(suite sur la page suivante)

(suite de la page précédente)

```
$requestParams = Router::getRequest()->getAttributes('params');
$this->Html->link('View', Router::reverse($requestParams));
```

Ensuite quand vous décidez de changer vos URLs, vous pouvez le faire en définissant une route. Cela changerait à la fois le mapping d'URL entrant, ainsi que les URLs générés.

Le choix de la technique est déterminé par la façon dont vous pouvez prédire les éléments du tableau de routage.

Utilisation de `Router::url()`

`Router::url()` vous permet d'utiliser des *Tableaux de routage* dans les situations où les éléments de tableau requis sont fixes ou facilement déduits.

Il fournira un routage inversé lorsque l'URL de destination est bien définie :

```
$this->Html->link(
    'View',
    ['controller' => 'Articles', 'action' => 'view', $id]
);
```

Il est également utile lorsque la destination est inconnue mais suit un modèle bien défini

```
$this->Html->link(
    'View',
    ['controller' => $controller, 'action' => 'view', $id]
);
```

Les éléments qui possèdent des clés numériques sont traités comme *Arguments Passés*.

Quand vous utilisez les URLs en tableau, vous pouvez définir les paramètres chaîne de la requête et les fragments de document en utilisant les clés spéciales :

```
$routes->url([
    'controller' => 'Articles',
    'action' => 'index',
    '?' => ['page' => 1],
    '#' => 'top'
]);

// Cela générera une URL comme:
/articles/index?page=1#top
```

Vous pouvez également utiliser n'importe quel élément spécial de route lorsque vous générez des URLs :

- `_ext` Utilisé pour *Routing des Extensions de Fichier* .
- `_base` Défini à `false` pour retirer le chemin de base de l'URL générée. Si votre application n'est pas dans le répertoire racine, cette option peut être utilisée pour générer les URLs qui sont "liées à cake".
- `_scheme` Défini pour créer les liens sur les schémas différents comme *webcal* ou *ftp*. Par défaut, au schéma courant.
- `_host` Définit l'hôte à utiliser pour le lien. Par défaut à l'hôte courant.
- `_port` Définit le port si vous avez besoin de créer les liens sur des ports non-standards.
- `_method` Définit le verbe HTTP à utiliser pour cette URL.
- `_full` Si à `true`, la valeur de `App.fullBaseUrl` vue dans *Configuration Générale* sera ajoutée devant les URL générées.
- `_https` Défini à `true` pour convertir l'URL générée à https, ou `false` pour forcer http.
- `_method` Definit la méthode HTTP à utiliser. Utile si vous travaillez avec *Créer des Routes RESTful*.

— `_name` Nom de route. Si vous avez configuré les routes nommées, vous pouvez utiliser cette clé pour les spécifier.

Utilisation de `Router::reverse()`

`Router::reverse()` vous permet d'utiliser les : ref : *request-parameters* dans les cas où l'URL courante modifiée sert de base à celle de destination mais que les éléments de l'URL courantes ne sont pas prévisibles.

À titre d'exemple, imaginez un blog permettant aux utilisateurs de créer des **Articles** et **Commentaires**, et de marquer les deux comme *publié* ou *brouillon*. Les deuxpages d'index pourrait inclure l'ID utilisateur. L'URL **Commentaires** pourrait également inclure un identifiant d'article pour identifier l'article auquel le commentaire fait référence.

Voici les URL correspondant à ce scénario :

```
 /articles/index/42
/comments/index/42/18
```

Lorsque l'auteur utilise ces pages, il serait pratique d'inclure des liens qui permettent d'afficher la page avec tous les résultats, publiés uniquement, ou à l'état brouillon seulement.

Pour garder le code DRY, il serait préférable d'inclure les liens via un élément :

```
// element/filter_published.php

$params = $this->getRequest()->getAttribute('params');

/* prépare l'url pour l'état 'Brouillon' */
$params = Hash::insert($params, '?.published', 0);
echo $this->Html->link(__('Brouillon'), Router::reverse($params));

/* prépare l'url pour l'état for 'Publié' */
$params = Hash::insert($params, '?.published', 1);
echo $this->Html->link(__('Publié'), Router::reverse($params));

/* prépare l'url pour tous les articles */
$params = Hash::remove($params, '?.published');
echo $this->Html->link(__('Tous'), Router::reverse($params));
```

Les liens générés par ces appels de méthode incluraient un ou deux passages paramètres en fonction de la structure de l'URL actuelle. Et le code fonctionnerait pour toute URL future, par exemple, si vous commencez à utiliser pathPrefixes ou si vous prévoyez de passer plus de paramètres.

Tableaux de Routages vs Paramètrees de Requête

La différence significative entre les deux tableaux et leur utilisation dans ces les méthodes de routage inversé sont dans la manière dont elles incluent les paramètres passés.

Les tableaux de routage incluent les paramètres fournis en tant que valeurs sans clé dans le tableau :

```
$url = [
    'controller' => 'Articles',
    'action' => 'View',
    $id, //a pass parameter
    'page' => 3, //un argument de requête (query)
];
```

Les paramètres de requête incluent les paramètres fournis dans la clé "pass" du tableau :

```
$url = [
    'controller' => 'Articles',
    'action' => 'View',
    'pass' => [$id], //the pass parameters
    '?' => 'page' => 3, //les arguments de la requête (query)
];
```

Il est donc possible, si vous le souhaitez, de convertir les paramètres de la requête en un tableau de routage ou vice versa.

Générer des URL de ressources

La classe `Asset` fournit des méthodes pour générer des URL vers les fichiers css, javascript, images et autres fichiers statiques de votre application :

```
use Cake\Routing\Asset;

// Génère une URL pointant vers APP/webroot/js/app.js
$js = Asset::scriptUrl('app.js');

// Génère une URL pointant vers APP/webroot/css/app.css
$css = Asset::cssUrl('app.css');

// Génère une URL pointant vers APP/webroot/image/logo.png
$img = Asset::imageUrl('logo.png');

// Génère une URL pointant vers APP/webroot/files/upload/photo.png
$file = Asset::url('files/upload/photo.png');
```

Les méthodes ci-dessus acceptent également un tableau d'options comme deuxième paramètre :

- `fullBase` Ajoute l'url complète incluant le nom de domaine.
- `pathPrefix` Indique le préfixe pour les url relatives.
- `plugin` Vous pouvez indiquer `false` pour éviter que les chemins ne soient traités comme des ressources appartenant à un plugin.
- `timestamp` Remplace la valeur de `Asset.timestamp` définie dans la configuration (Configure). Mettez-le à `false` pour désactiver la génération des timestamp. Mettez-le à `true` pour générer les timestamp quand debug est à `true`. Mettez-le à `'force'` pour forcer la génération des timestamps indépendemment de la valeur du paramètre debug.

```
// Génère http://example.org/img/logo.png
$img = Asset::url('logo.png', ['fullBase' => true]);

// Génère /img/logo.png?1568563625
// Pour lequel le timestamp correspond à la date de dernière modification du fichier
$img = Asset::url('logo.png', ['timestamp' => true]);
```

Pour générer des URL de ressources pour les fichiers dans les plugins, utilisez la *syntaxe de plugin* :

```
// Génère `/debug_kit/img/cake.png`
$img = Asset::imageUrl('DebugKit.cake.png');
```

Routing de Redirection

Le routing de redirection permet de créer des statuts HTTP de redirection 30x pour les routes entrantes et les faire pointer vers des URLs différentes. C'est utile lorsque vous souhaitez informer les applications clientes qu'une ressource a été déplacée et que vous ne voulez pas exposer deux URLs pour le même contenu.

Les routes de redirection sont différentes des routes normales car elles effectuent une redirection d'en-tête si une correspondance est trouvée. La redirection peut se produire vers une destination au sein de votre application ou un emplacement à extérieur :

```
$routes->scope('/', function (RouteBuilder $routes) {
    $routes->redirect(
        '/home/*',
        ['controller' => 'Articles', 'action' => 'view'],
        ['persist' => true]
        // Ou ['persist'=>['id']] pour la valeur par défaut du routage
        // quand l'action 'view' attend $id comme argument.
    );
})
```

Redirige `/home/*` vers `/articles/view` et passe les paramètres vers `/articles/view`. Utiliser un tableau comme destination de redirection vous permet d'utiliser différentes routes pour définir où la chaine URL devrait être redirigée. Vous pouvez rediriger vers des destinations externes en utilisant des chaines URLs pour destination :

```
$routes->scope('/', function (RouteBuilder $routes) {
    $routes->redirect('/articles/*', 'https://google.com', ['status' => 302]);
});
```

Cela redirigerait `/articles/*` vers `https://google.com` avec un statut HTTP 302.

Routage des Entités

Le routage d'entité vous permet d'utiliser une entité, un tableau ou un objet `ArrayAccess` comme source des paramètres de routage. Cela vous permet de refactoriser vos routes plus facilement et de générer des URL avec moins de code. Par exemple, si vous commencez avec une route qui ressemble à :

```
$routes->get(
    '/view/{id}',
    ['controller' => 'Articles', 'action' => 'view'],
    'articles:view'
);
```

Vous pouvez générer une URL vers cette route comme suit :

```
// $article est une entité dans le contexte local.
Router::url(['_name' => 'articles:view', 'id' => $article->id]);
```

Plus tard, vous souhaiterez peut-être exposer le slug de l'article dans l'URL à des fins de référencement (SEO). Pour ce faire, vous devez mettre à jour partout où vous générez une URL vers la route `articles:vue`, ce qui peut prendre un certain temps. Si nous utilisons des routes d'entité, nous transmettons l'entité entière de l'article à la génération d'URL, ce qui nous permet d'éviter tout travail supplémentaire lorsque les URL nécessitent plus de paramètres :

```
use Cake\Routing\Route\EntityRoute;
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Créez des routes d'entité pour le reste du contexte.
$routes->setRouteClass(EntityRoute::class);

// Créez une route comme précédemment.
$routes->get(
    '/view/{id}/{slug}',
    ['controller' => 'Articles', 'action' => 'view'],
    'articles:view'
);
```

Maintenant, nous pouvons générer des URL en utilisant la clé `_entity` :

```
Router::url(['_name' => 'articles:view', '_entity' => $article]);
```

Cela extraira à la fois la propriété `id` et la propriété `slug` de l'entité fournie.

Classes Route Personnalisées

Les classes de route personnalisées vous permettent d'étendre et modifier la manière dont les routes individuelles parsent les requêtes et gèrent le routing inversé. Les classes de route suivent quelques conventions :

- Les classes de Route doivent se trouver dans le namespace `Routing\Route` de votre application ou plugin.
- Les classes de Route doivent étendre `Cake\Routing\Route\Route`.
- Les classes de Route doivent implémenter au moins un des méthodes `match()` et/ou `parse()`.

La méthode `parse()` est utilisée pour parser une URL entrante. Elle doit générer un tableau de paramètres de requêtes qui peuvent être résolus en contrôleur & action. Renvoyez `null` pour indiquer une erreur de correspondance.

La méthode `match()` est utilisée pour faire correspondre un tableau de paramètres d'URL et créer une chaine URL. Si les paramètres d'URL ne correspondent pas, `false` doit être renvoyé.

Vous pouvez utiliser votre classe de route personnalisée lors de la création d'une route en utilisant l'option `routeClass` :

```
$routes->connect(
    '/{slug}',
    ['controller' => 'Articles', 'action' => 'view'],
    ['routeClass' => 'SlugRoute']
);

// Ou en définissant la routeClass dans votre scope.
$routes->scope('/', function (RouteBuilder $routes) {
    $routes->setRouteClass('SlugRoute');
    $routes->connect(
        '/{slug}',
        ['controller' => 'Articles', 'action' => 'view']
    );
});
```

Cette route créera une instance de `SlugRoute` et vous permettra d'implémenter une gestion des paramètres personnalisée. Vous pouvez utiliser les classes routes des plugins en utilisant la *syntaxe de plugin* standard.

Classe de Route par Défaut

static Cake\Routing\RouterBuilder::**setRouteClass**(*$routeClass = null*)

Si vous voulez utiliser une autre classe de route pour toutes vos routes en plus de la `Route` par défaut, vous pouvez faire ceci en appelant `RouterBuilder::setRouteClass()` avant de définir la moindre route et éviter de spécifier l'option `routeClass` pour chaque route. Par exemple en utilisant :

```
use Cake\Routing\Route\DashedRoute;

$routes->setRouteClass(DashedRoute::class);
```

Cela provoquera l'utilisation de la classe `DashedRoute` pour toutes les routes suivantes. Appeler la méthode sans argument va retourner la classe de route courante par défaut.

Méthode Fallbacks

Cake\Routing\RouterBuilder::**fallbacks**(*$routeClass = null*)

La méthode fallbacks (de repli) est un raccourci simple pour définir les routes par défaut. La méthode utilise la classe de route passée pour les règles définies ou, si aucune classe n'est passée, la classe retournée par `RouterBuilder::setRouteClass()` sera utilisée.

Appelez fallbacks comme ceci :

```
use Cake\Routing\Route\DashedRoute;

$routes->fallbacks(DashedRoute::class);
```

Est équivalent à ces appels explicites :

```
use Cake\Routing\Route\DashedRoute;

$routes->connect('/{controller}', ['action' => 'index'], ['routeClass' =>␣
↪DashedRoute::class]);
$routes->connect('/{controller}/{action}/*', [], ['routeClass' => DashedRoute::class]);
```

> **Note**
>
> Utiliser la classe route par défaut (`Route`) avec fallbacks, ou toute route avec les éléments `{plugin}` et/ou `{controller}` résultera en des URL incompatibles.

Créer des Paramètres d'URL Persistants

En utilisant les fonctions de filtre, vous pouvez vous immiscer dans le process de génération d'URL. Les fonctions de filtres sont appelées *avant* que les URLs ne soient vérifiées via les routes, cela vous permet donc de préparer les URLs avant le routing.

Les fonctions de callback de filtre doivent attendre les paramètres suivants :
- `$params` Le paramètre d'URL à traiter.
- `$request` La requête actuelle (une instance de `Cake\Http\ServerRequest`).

La fonction filtre d'URL doit *toujours* retourner les paramètres même s'ils n'ont pas été modifiés.

Les filtres d'URL vous permettent d'implémenter des fonctionnalités telles que l'utilisation de paramètres d'URL persistants :

```
Router::addUrlFilter(function (array $params, ServerRequest $request) {
    if ($request->getParam('lang') && !isset($params['lang'])) {
        $params['lang'] = $request->getParam('lang');
    }

    return $params;
});
```

Le fonctions de filtres sont appliquées dans l'ordre dans lequel elles sont connectées.

Un autre cas lorsque l'on souhaite changer une route en particulier à la volée (pour les routes de plugin par exemple) :

```
Router::addUrlFilter(function (array $params, ServerRequest $request) {
    if (empty($params['plugin']) || $params['plugin'] !== 'MyPlugin' || empty($params[
↪'controller'])) {
        return $params;
    }
    if ($params['controller'] === 'Languages' && $params['action'] === 'view') {
        $params['controller'] = 'Locations';
        $params['action'] = 'index';
        $params['language'] = $params[0];
        unset($params[0]);
    }

    return $params;
});
```

Transformera la route suivante :

```
Router::url(['plugin' => 'MyPlugin', 'controller' => 'Languages', 'action' => 'view', 'es
↪']);
```

en ceci :

```
Router::url(['plugin' => 'MyPlugin', 'controller' => 'Locations', 'action' => 'index',
↪'language' => 'es']);
```

Avertissement

Si vous utilisez les fonctionnalités de mise en cache routing-middleware vous devez définir les filtres d'URL dans le `bootstrap()` de votre application car les filtres ne font pas partie des données mises en cache.

CHAPITRE 9

Les Objets Request & Response

Les objets `ServerRequest` et `Response` fournissent une abstraction autour de la requête et des réponses HTTP. L'objet `ServerRequest` dans CakePHP vous permet de faire une introspection de la requête entrante, tandis que l'objet `Response` vous permet de créer sans effort des réponses HTTP à partir de vos controllers.

ServerRequest

class `Cake\Http\`**`ServerRequest`**

`ServerRequest` est l'objet requête utilisé par défaut dans CakePHP. Il centralise un certain nombre de fonctionnalités pour interroger et interagir avec les données demandées. Pour chaque requête, une `ServerRequest` est créée et passée en référence aux différentes couches de l'application que la requête de données utilise. Par défaut la requête est assignée à `$this->request`, et est disponible dans les Controllers, Cells, Vues et Helpers. Vous pouvez aussi y accéder dans les Components en utilisant la référence du controller. Certaines des tâches que `ServerRequest` permet sont les suivantes :

- Transformer les tableaux GET, POST, et FILES en structures de données avec lesquelles vous êtes familiers.
- Fournir une introspection de l'environnement se rapportant à la demande. Des informations comme les d'entêtes (headers) envoyés, l'adresse IP du client et les informations des sous-domaines/domaines sur lesquels le serveur de l'application tourne.
- Fournit un accès aux paramètres de la requête à la fois en tableaux indicés et en propriétés d'un objet.

L'objet ServerRequest de CakePHP implémente l'interface PSR-7 ServerServerRequestInterface [109] facilitant l'utilisation des librairies en-dehors de CakePHP.

Paramètres de la Requête

`ServerRequest` propose les paramètres de routing avec la méthode `getParam()` :

```
$controllerName = $this->request->getParam('controller');
```

Pour obtenir tous les paramètres de routage sous forme de tableau, utilisez getAttribute () :

109. https://www.php-fig.org/psr/psr-7/

```
$parameters = $this->request->getAttribute('params');
```

Tous les éléments de route *Les Eléments de Route* sont accessibles à travers cette interface.

En plus des éléments de routes *Les Eléments de Route*, vous avez souvent besoin d'accéder aux arguments passés *Arguments Passés*. Ceux-ci sont aussi tous les deux disponibles dans l'objet `request` :

```
// Arguments passés
$passedArgs = $this->request->getParam('pass');
```

Tous vous fournissent un accès aux arguments passés. Il y a de nombreux paramètres importants et utiles que CakePHP utilise en interne qu'on peut aussi trouver dans les paramètres de routing :

- `plugin` Le plugin gérant la requête. Aura une valeur nulle quand il n'y a pas de plugins.
- `controller` Le controller gérant la requête courante.
- `action` L'action gérant la requête courante.
- `prefix` Le préfixe pour l'action courante. Voir *Prefix de Routage* pour plus d'informations.

Accéder aux Paramètres Querystring

`Cake\Http\ServerRequest::getQuery`(*$name*, *$default = null*)

Les paramètres Querystring peuvent être lus en utilisant la méthode `getQuery()` :

```
// l'URL est /posts/index?page=1&sort=title
$page = $this->request->getQuery('page');
```

Vous pouvez soit directement accéder à la propriété demandée, soit vous pouvez utiliser `getQuery()` pour lire l'URL requêtée sans erreur. Toute clé qui n'existe pas va retourner `null` :

```
$foo = $this->request->getQuery('valeur_qui_n_existe_pas');
// $foo === null

// Vous pouvez également définir des valeurs par défaut
$foo = $this->request->getQuery('n_existe_pas', 'valeur par défaut');
```

Si vous souhaitez accéder à tous les paramètres de requête, vous pouvez utiliser `getQueryParams()` :

```
$query = $this->request->getQueryParams();
```

Données du Corps de la Requête

`Cake\Http\ServerRequest::getData`(*$name*, *$default = null*)

Toutes les données POST sont accessibles en utilisant `Cake\Http\ServerRequest::getData()`. Par exemple :

```
// Un input avec un attribut de nom égal à 'title' est accessible via
$title = $this->request->getData('title');
```

Vous pouvez utiliser des noms séparés par des points pour accéder aux données imbriquées. Par exemple :

```
$value = $this->request->getData('adresse.nom_de_rue');
```

Pour toute clé qui n'existe pas, la valeur par `$default` sera retournée :

```
$foo = $this->request->getData('Valeur.qui.n.existe.pas');
// $foo == null
```

Vous pouvez également utiliser body-parser-middleware pour analyser le corps de la requête de différents types de contenu dans un tableau de sortie, de sorte qu'il soit accessible via `ServerRequest::getData()`.

Si vous souhaitez accéder à tous les paramètres de requête, vous pouvez utiliser `getQueryParams()` :

```
$query = $this->request->getQueryParams();
```

Envoyer des fichiers

Les fichiers téléchargés sont accessibles via les données du corps de la requête, en utilisant la méthode *Cake\Http\ServerRequest::getData()* décrite ci-dessus. Par exemple, un fichier correspondant au nom `attachment`, peut être accédé comme ceci :

```
$attachment = $this->request->getData('attachment');
```

Par défaut, les téléchargements de fichiers sont représentés dans les données de requête comme des objets qui implémentent \Psr\Http\Message\UploadedFileInterface [110]. Dans l'actuelle implémentation, la variable `$attachment` dans l'exemple ci-dessus contiendrait par défaut une instance de `\Laminas\Diactoros\UploadedFile`.

L'accès aux détails du fichier téléchargé est assez simple, voici comment obtenir les mêmes données que celles fournies par le tableau de téléchargement de fichier des anciennes versions de cakePHP :

```
$name = $attachment->getClientFilename();
$type = $attachment->getClientMediaType();
$size = $attachment->getSize();
$tmpName = $attachment->getStream()->getMetadata('uri');
$error = $attachment->getError();
```

Le déplacement du fichier téléchargé de son emplacement temporaire vers l'emplacement cible souhaité ne nécessite pas d'accéder manuellement au fichier temporaire, à la place cela peut être facilement fait en utilisant les méthodes `moveTo()` des objets :

```
$attachment->moveTo($targetPath);
```

Dans un environnement HTTP, la méthode `moveTo()` validera automatiquement si le fichier est un fichier téléchargé, et lancera une exception si nécessaire. Dans un environnement CLI, où le concept de téléchargement de fichiers n'existe pas, il permettra de déplacer le fichier que vous avez référencé indépendamment de ses origines, ce qui rend possible le test des téléchargements de fichiers.

Pour revenir à l'utilisation des tableaux de téléchargement de fichiers des versions antérieures, définissez la valeur de configuration `App.uploadedFilesAsObjects` à `false`, par exemple dans votre fichier `config/app.php` :

```
return [
    // ...
    'App' => [
        // ...
        'uploadedFilesAsObjects' => false,
    ],
    // ...
];
```

110. https://www.php-fig.org/psr/psr-7/#16-uploaded-files

Avec l'option désactivée, les téléchargements de fichiers sont représentés dans les données de la requête sous forme de tableaux, avec une structure normalisée qui reste la même y compris pour pour les entrées/noms imbriqués, ce qui est différent de la façon dont PHP les représente dans la variable `$ _FILES` (reportez-vous au *manuel PHP <https ://www.php.net/manual/en/features.file-upload.php>* __ pour plus d'informations), c'est-à-dire que la valeur `$attachment` ressemblerait à quelque chose comme ceci :

```
[
    'name' => 'attachment.txt',
    'type' => 'text/plain',
    'size' => 123,
    'tmp_name' => '/tmp/hfz6dbn.tmp'
    'error' => 0
]
```

Astuce

Les fichiers téléchargés sont également accessibles en tant qu'objets séparément des données de requête via les méthodes *Cake\Http\ServerRequest::getUploadedFile()* et *Cake\Http\ServerRequest::getUploadedFiles()*. Ces méthodes renverront toujours des objets, indépendamment de la configuration `App.uploadedFilesAsObjects`.

`Cake\Http\ServerRequest::getUploadedFile($path)`

Renvoie le fichier téléchargé à un chemin spécifique. Le chemin utilise la même syntaxe de point (dot) que la méthode *Cake\Http\ServerRequest::getData()* :

```
$attachment = $this->request->getUploadedFile('attachment');
```

Contrairement à *Cake\Http\ServerRequest::getData()*, *Cake\Http\ServerRequest::getUploadedFile()* ne renvoie des données que lorsqu'un téléchargement de fichier réel existe pour le chemin donné, s'il existe des données de corps de requête régulières, non liées à un fichier, correspondant au chemin donné, alors cette méthode retournera `null`, comme elle le ferait pour tout chemin inexistant.

`Cake\Http\ServerRequest::getUploadedFiles()`

Renvoie tous les fichiers téléchargés dans une structure de tableau normalisée. Pour l'exemple ci-dessus avec le nom d'entrée de fichier `attachement`, la structure ressemblerait à :

```
[
      'attachment' => object(Laminas\Diactoros\UploadedFile) {
          // ...
      }
]
```

`Cake\Http\ServerRequest::withUploadedFiles(array $files)`

Cette méthode définit les fichiers téléchargés de l'objet de requête, elle accepte un tableau d'objets qui implémentent \Psr\Http\Message\UploadedFileInterface [111]. Elle va remplacer tous les fichiers téléchargés éventuellement existants :

```
$files = [
    'MyModel' => [
        'attachment' => new \Laminas\Diactoros\UploadedFile(
```

(suite sur la page suivante)

111. https://www.php-fig.org/psr/psr-7/#16-uploaded-files

(suite de la page précédente)

```
            $streamOrFile,
            $size,
            $errorStatus,
            $clientFilename,
            $clientMediaType
        ),
        'anotherAttachment' => new \Laminas\Diactoros\UploadedFile(
            '/tmp/hfz6dbn.tmp',
            123,
            \UPLOAD_ERR_OK,
            'attachment.txt',
            'text/plain'
        ),
    ],
];

$this->request = $this->request->withUploadedFiles($files);
```

Note

Les fichiers téléchargés qui ont été ajoutés à la demande via cette méthode ne seront *pas* disponibles dans les données du corps de la requête, c'est-à-dire que vous ne pouvez pas les récupérer via *Cake\Http\ServerRequest::getData()* ! Si vous en avez besoin également dans les données de la requête, vous devez les définir via `Cake\Http\ServerRequest::withData()` ou `Cake\Http\ServerRequest::withParsedBody()`.

Accéder aux Données PUT, PATCH ou DELETE

`Cake\Http\ServerRequest::input`(*$callback*[, *$options*])

Quand vous construisez des services REST, vous acceptez souvent des données requêtées sur des requêtes `PUT` et `DELETE`. Toute donnée de corps de requête `application/x-www-form-urlencoded` va automatiquement être parsée et définie dans `$this->data` pour les requêtes `PUT` et `DELETE`. Si vous acceptez les données JSON ou XML, regardez la section ci-dessous pour voir comment vous pouvez accéder aux corps de ces requêtes.

Lorsque vous accédez aux données d'entrée, vous pouvez les décoder avec une fonction optionnelle. Cela peut être utile quand vous devez interagir avec du contenu de requête XML ou JSON. Les paramètres supplémentaires pour la fonction de décodage peuvent être passés comme arguments à `input()` :

```
$jsonData = $this->request->input('json_decode');
```

Variables d'Environnement (à partir de $_SERVER et $_ENV)

`Cake\Http\ServerRequest::env`(*$key*, *$value = null*)

`ServerRequest::env()` est un wrapper pour la fonction globale `env()` et agit comme un getter/setter pour les variables d'environnement sans avoir à modifier les variables globales `$_SERVER` et `$_ENV` :

```
// Obtenir l'host
$host = $this->request->env('HTTP_HOST');

// Définir une valeur, généralement utile pour les tests.
$this->request->env('REQUEST_METHOD', 'POST');
```

Pour accéder à toutes les variables d'environnement dans une requête, utilisez `getServerParams()` :

```
$env = $this->request->getServerParams();
```

Données XML ou JSON

Les applications employant *REST* échangent souvent des données dans des corps de requête post non encodés en URL. Vous pouvez lire les données entrantes dans n'importe quel format en utilisant *input()*. En fournissant une fonction de décodage, vous pouvez recevoir le contenu dans un format déserializé :

```
// Obtenir les données encodées JSON soumises par une action PUT/POST
$jsonData = $this->request->input('json_decode');
```

Certaines méthodes de desérialization requièrent des paramètres supplémentaires quand elles sont appelées, comme le paramètre de type "comme tableau" de `json_decode`. Si vous voulez convertir du XML en objet DOMDocument, *input()* supporte aussi le passage de paramètres supplémentaires :

```
// Obtenir les données encodées en XML soumises avec une action PUT/POST
$data = $this->request->input('Cake\Utility\Xml::build', ['return' => 'domdocument']);
```

Informations du Chemin

L'objet `ServerRequest` fournit aussi des informations utiles sur les chemins dans votre application. Les attributs `base` et `webroot` sont utiles pour générer des URLs et déterminer si votre application est ou n'est pas dans un sous-dossier. Les attributs que vous pouvez utiliser sont :

```
// Suppose que la requête URL courante est /subdir/articles/edit/1?page=1

// Contient /subdir/articles/edit/1?page=1
$here = $request->getRequestTarget();

// Contient /subdir
$base = $request->getAttribute('base');

// Contient /subdir/
$base = $request->getAttribute('webroot');
```

Vérifier les Conditions de la Requête

`Cake\Http\ServerRequest::is`(*$type*, *$args...*)

L'objet `ServerRequest` fournit une façon d'inspecter différentes conditions de la requête. En utilisant la méthode `is()`, vous pouvez vérifier un certain nombre de conditions, ainsi qu'inspecter d'autres critères de la requête spécifique à l'application :

```
$isPost = $this->request->is('post');
```

Vous pouvez aussi étendre les détecteurs de la requête qui sont disponibles, en utilisant *Cake\Http\ServerRequest::addDetector()* pour créer de nouveaux types de détecteurs. Il y a différents types de détecteurs que vous pouvez créer :

- Comparaison avec valeur d'environnement - Compare l'égalité de la valeur extraite à partir de *env()* avec la valeur fournie.
- **Comparaison de la valeur d'en-tête - Si l'en-tête spécifié existe avec la** valeur spécifiée, la fonction appelable renvoie true.

— Comparaison de valeur avec motif - Vous permet de comparer la valeur extraite de *env()* avec une expression régulière.
— Comparaison basée sur les options - Utilise une liste d'options pour créer une expression régulière. Les appels suivants pour ajouter un détecteur d'option déjà défini, vont fusionner les options.
— Les détecteurs de Callback - Vous permettent de fournir un type "callback" pour gérer la vérification. Le callback va recevoir l'objet `ServerRequest` comme seul paramètre.

`Cake\Http\ServerRequest::addDetector($name, $options)`

Quelques exemples seraient :

```
// Ajouter un détecteur d'environnement.
$this->request->addDetector(
    'post',
    ['env' => 'REQUEST_METHOD', 'value' => 'POST']
);

// Ajouter un détecteur de valeur avec motif.
$this->request->addDetector(
    'iphone',
    ['env' => 'HTTP_USER_AGENT', 'pattern' => '/iPhone/i']
);

// Ajouter un détecteur d'options
$this->request->addDetector('internalIp', [
    'env' => 'CLIENT_IP',
    'options' => ['192.168.0.101', '192.168.0.100']
]);

// Ajouter un détecteur d'en-tête avec comparaison de valeurs
$this->request->addDetector('fancy', [
    'env' => 'CLIENT_IP',
    'header' => ['X-Fancy' => 1]
]);

// Ajouter un détecteur d'en-tête avec comparaison appelable
$this->request->addDetector('fancy', [
    'env' => 'CLIENT_IP',
    'header' => ['X-Fancy' => function ($value, $header) {
        return in_array($value, ['1', '0', 'yes', 'no'], true);
    }]
]);

// Ajouter un détecteur de callback. Doit être un callable valide.
$this->request->addDetector(
    'awesome',
    function ($request) {
        return $request->getParam('awesome');
    }
);

// Ajouter un détecteur qui utilise des arguments supplémentaires.
$this->request->addDetector(
    'csv',
    [
```

(suite sur la page suivante)

(suite de la page précédente)

```
        'accept' => ['text/csv'],
        'param' => '_ext',
        'value' => 'csv',
    ]
);
```

Il y a plusieurs détecteurs intégrés que vous pouvez utiliser :

- `is('get')` Vérifie si la requête courante est un GET.
- `is('put')` Vérifie si la requête courante est un PUT.
- `is('patch')` Vérifie si la requête courante est un PATCH.
- `is('post')` Vérifie si la requête courante est un POST.
- `is('delete')` Vérifie si la requête courante est un DELETE.
- `is('head')` Vérifie si la requête courante est un HEAD.
- `is('options')` Vérifie si la requête courante est OPTIONS.
- `is('ajax')` Vérifie si la requête courante vient d'un X-Requested-With = XMLHttpRequest.
- `is('ssl')` Vérifie si la requête courante est via SSL.
- `is('flash')` Vérifie si la requête courante a un User-Agent de Flash.
- `is('json')` Vérifie si la requête a l'extension "json" ajoutée et si elle accepte le mimetype "application/json".
- `is('xml')` Vérifie si la requête a l'extension "xml" ajoutée et si elle accepte le mimetype "application/xml" ou "text/xml".

`ServerRequest` inclut aussi des méthodes comme *`Cake\Http\ServerRequest::domain()`*, *`Cake\Http\ServerRequest::subdomains()`* et *`Cake\Http\ServerRequest::host()`* qui facilitent la vie des applications avec sous-domaines.

Données de Session

Pour accéder à la session pour une requête donnée, utilisez la méthode `getSession()` ou l'attribut `session` :

```
$session = $this->request->getSession();
$session = $this->request->getAttribute('session');

$userName = $session->read('Auth.User.name');
```

Pour plus d'informations, consultez la documentation *Sessions* sur la façon d'utiliser l'objet `Session`.

Hôte et Nom de Domaine

`Cake\Http\ServerRequest::domain(`*$tldLength = 1*`)`

Retourne le nom de domaine sur lequel votre application tourne :

```
// Affiche 'example.org'
echo $request->domain();
```

`Cake\Http\ServerRequest::subdomains(`*$tldLength = 1*`)`

Retourne un tableau avec les sous-domaines sur lequel votre application tourne :

```
// Retourne ['my', 'dev'] pour 'my.dev.example.org'
$subdomains = $request->subdomains();
```

`Cake\Http\ServerRequest::host()`

Retourne l'hôte sur lequel votre application tourne :

```
// Affiche 'my.dev.example.org'
echo $request->host();
```

Lire la Méthode HTTP

`Cake\Http\ServerRequest::getMethod()`

Retourne le type de méthode HTTP avec lequel la requête a été faite :

```
// Affiche POST
echo $request->getMethod();
```

Restreindre les Méthodes HTTP qu'une Action Accepte

`Cake\Http\ServerRequest::allowMethod($methods)`

Définit les méthodes HTTP autorisées. Si elles ne correspondent pas, elle va lancer une `MethodNotAllowedException`. La réponse 405 va inclure l'en-tête `Allow` nécessaire avec les méthodes passées :

```
public function delete()
{
    // Accepter uniquement les demandes POST et DELETE
    $this->request->allowMethod(['post', 'delete']);
    ...
}
```

Lire les en-têtes HTTP

Ces méthodes vous permettent d'accéder à n'importe quel en-tête `HTTP_*` qui a été utilisé dans la requête. Par exemple :

```
// Récupère le header dans une chaîne
$userAgent = $this->request->getHeaderLine('User-Agent');

// Récupère un tableau contenant toutes les valeurs.
$acceptHeader = $this->request->getHeader('Accept');

// Vérifie l'existence d'un header
$hasAcceptHeader = $this->request->hasHeader('Accept');
```

Du fait que certaines installations d'Apache ne rendent pas le header `Authorization` accessible, CakePHP le rend disponible via des méthodes spécifiques.

`Cake\Http\ServerRequest::referer($local = true)`

Retourne l'adresse référente de la requête.

`Cake\Http\ServerRequest::clientIp()`

Retourne l'adresse IP du visiteur.

Faire Confiance aux Headers de Proxy

Si votre application est derrière un load balancer ou exécutée sur un service cloud, vous voudrez souvent obtenir l'hôte de load balancer, le port et le schéma dans vos requêtes. Souvent les load balancers vont aussi envoyer des en-têtes `HTTP-X-Forwarded-*` avec les valeurs originales. Les en-têtes forwardés ne seront pas utilisés par CakePHP directement. Pour que l'objet request utilise les en-têtes, définissez la propriété `trustProxy` à `true` :

```
$this->request->trustProxy = true;

// Ces méthodes utiliseront maintenant les en-têtes du proxy.
$port = $this->request->port();
$host = $this->request->host();
$scheme = $this->request->scheme();
$clientIp = $this->request->clientIp();
```

Une fois que les proxys sont approuvés, la méthode `clientIp()` utilisera la *dernière* adresse IP dans l'en-tête `X-Forwarded-For`. Si votre application est derrière plusieurs proxies, vous pouvez utiliser `setTrustedProxies()` pour définir les adresses IP des proxies sous votre contrôle :

```
request->setTrustedProxies(['127.1.1.1', '127.8.1.3']);
```

Une fois les proxys approuvés, `clientIp()` utilisera la première adresse IP de l'en-tête `X-Forwarded-For` à condition que ce soit la seule valeur qui ne provienne pas d'un proxy approuvé.

Vérifier les En-têtes Acceptés

`Cake\Http\ServerRequest::accepts`(*$type = null*)

Trouve les types de contenu que le client accepte ou vérifie s'il accepte un type particulier de contenu.

Récupère tous les types :

```
$accepts = $this->request->accepts();
```

Vérifie pour un unique type :

```
$acceptsJson = $this->request->accepts('application/json');
```

static `Cake\Http\ServerRequest::acceptLanguage`(*$language = null*)

Obtenir toutes les langues acceptées par le client, ou alors vérifier si une langue spécifique est acceptée.

Obtenir la liste des langues acceptées :

```
$acceptsLanguages = $this->request->acceptLanguage();
```

Vérifier si une langue spécifique est acceptée :

```
$acceptsFrench = $this->request->acceptLanguage('fr-fr');
```

Lire des Cookies

Les cookies de la requête peuvent être lus à travers plusieurs méthodes :

```
// Récupère la valeur du cookie, ou null si le cookie n'existe pas
$rememberMe = $this->request->getCookie('remember_me');
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Lit la valeur ou retourne le défaut (qui est 0 ici)
$rememberMe = $this->request->getCookie('remember_me', 0);

// Récupère tous les cookies dans un tableau
$cookies = $this->request->getCookieParams();

// Récupère une instance de CookieCollection
$cookies = $this->request->getCookieCollection()
```

Référez-vous à la documentation de *Cake\Http\Cookie\CookieCollection* pour savoir comment travailler avec les collections de cookies.

Fichiers uploadés

Les requêtes exposent les données du fichier téléchargé dans `getData()` ou ``getUploadedFiles()` comme objets implémentant l'interface``UploadedFileInterface`` :

```
// Récupère une liste des objets UploadedFile
$files = $request->getUploadedFiles();

// Lire les données du fichier.
$files[0]->getStream();
$files[0]->getSize();
$files[0]->getClientFileName();

// Déplacer le fichier.
$files[0]->moveTo($targetPath);
```

Manipuler les URIs

Les requêtes contiennent un objet URI, qui contient des méthodes pour interagir avec l'URI demandée :

```
// Récupère l'URI
$uri = $request->getUri();

// Extrait les données de l'URI.
$path = $uri->getPath();
$query = $uri->getQuery();
$host = $uri->getHost();
```

Response

class Cake\Http**Response**

Cake\Http\Response est la classe de réponse par défaut dans CakePHP. Elle encapsule un nombre de fonctionnalités et de caractéristiques pour la génération de réponses HTTP dans votre application. Elle aide aussi à tester des objets factices (mocks/stubs), vous permettant d'inspecter les en-têtes qui vont être envoyés. *Cake\Http\ServerRequest*, *Cake\Http\Response* consolide un certain nombre de méthodes qu'on pouvait trouver avant dans `Controller`, *RequestHandlerComponent* et `Dispatcher`. Les anciennes méthodes sont dépréciées en faveur de l'utilisation de *Cake\Http\Response*.

`Response` fournit une interface pour envelopper les tâches de réponse communes liées, telles que :

— Envoyer des en-têtes pour les redirections.
— Envoyer des en-têtes de type de contenu.
— Envoyer n'importe quel en-tête.
— Envoyer le corps de la réponse.

Gérer les Types de Contenu

`Cake\Http\Response::withType($contentType = null)`

Vous pouvez contrôler le Content-Type des réponses de votre application en utilisant `Cake\Http\Response::withType()`. Si votre application a besoin de gérer les types de contenu qui ne sont pas construits dans Response, vous pouvez faire correspondre ces types avec `setTypeMap()` comme ceci :

```
// Ajouter un type vCard
$this->response->setTypeMap('vcf', ['text/v-card']);

// Configurer la réponse de Type de Contenu pour vcard.
$this->response = $this->response->withType('vcf');
```

Habituellement, vous voudrez faire correspondre des types de contenu supplémentaires dans le callback `beforeFilter()` de votre controller afin que vous puissiez tirer parti de la fonctionnalité de commutation automatique de vue de `RequestHandlerComponent`, si vous l'utilisez.

Envoyer des fichiers

`Cake\Http\Response::withFile($path, $options = [])`

Il y a des moments où vous souhaitez envoyer des fichiers en réponse à vos demandes. Vous pouvez accomplir cela en utilisant `Cake\Http\Response::withFile()` :

```
public function sendFile($id)
{
    $file = $this->Attachments->getFile($id);
    $response = $this->response->withFile($file['path']);
    // Renvoie la réponse pour empêcher le contrôleur d'essayer
    // de rendre une vue
    return $response;
}
```

Comme indiqué dans l'exemple ci-dessus, vous devez transmettre le chemin du fichier à la méthode. CakePHP enverra un en-tête de type de contenu approprié s'il s'agit d'un type de fichier connu répertorié dans *Cake\Http\Response : :$_mimeTypes*. Vous pouvez ajouter de nouveaux types avant d'appeler `Cake\Http\Response::withFile()` en utilisant la méthode `Cake\Http\Response::withType()`.

Si vous le souhaitez, vous pouvez également forcer le téléchargement d'un fichier au lieu de l'afficher dans le navigateur en spécifiant les options :

```
$response = $this->response->withFile(
    $file['path'],
    ['download' => true, 'name' => 'foo']
);
```

Les options prises en charge sont :

name
Le nom vous permet de spécifier un autre nom de fichier à envoyer l'utilisateur.

download
Une valeur booléenne indiquant si les en-têtes doivent être définis pour forcer le téléchargement.

Envoyer une Chaîne de Caractères comme Fichier

Vous pouvez répondre avec un fichier qui n'existe pas sur le disque, par exemple si vous voulez générer un pdf ou un ics à la volée à partir d'une chaine :

```
public function sendIcs()
{
   $icsString = $this->Calendars->generateIcs();
   $response = $this->response;

   // Injecter le contenu de la chaîne dans le corps de la réponse
   $response = $response->withStringBody($icsString);

   $response = $response->withType('ics');

   // Force le téléchargement de fichier en option
   $response = $response->withDownload('filename_for_download.ics');

   // Renvoie la réponse pour empêcher le contrôleur d'essayer
   // de rendre une vue
   return $response;
}
```

Les fonctions de rappel (callbacks) peuvent également renvoyer le corps en tant que chaîne de caractères :

```
$path = '/some/file.png';
$this->response->body(function () use ($path) {
    return file_get_contents($path);
});
```

Définir les En-têtes

`Cake\Http\Response::withHeader`(*$header*, *$value*)

La définition de headers se fait avec la méthode *`Cake\Http\Response::withHeader()`*. Comme toutes les méthodes de l'interface PSR-7, cette méthode retourne une nouvelle instance avec le nouvel header :

```
// Ajoute/remplace un header
$response = $response->withHeader('X-Extra', 'My header');

// Définit plusieurs headers
$response = $response->withHeader('X-Extra', 'My header')
    ->withHeader('Location', 'http://example.com');

// Ajoute une valeur à un header existant
$response = $response->withAddedHeader('Set-Cookie', 'remember_me=1');
```

Les headers ne sont pas envoyés dès que vous les définissez. Ils sont stockés jusqu'à ce que la réponse soit émise par `Cake\Http\Server`.

Vous pouvez maintenant utiliser la méthode `Cake\Http\Response::withLocation()` pour définir ou obtenir directement le header « redirect location ».

Définir le Corps de la réponse

`Cake\Http\Response::withStringBody($string)`

Pour définir une chaîne comme corps de réponse, écrivez ceci :

```
// Définit une chaîne dans le corps
$response = $response->withStringBody('My Body');

// Si vous souhaitez une réponse JSON
$response = $response->withType('application/json')
    ->withStringBody(json_encode(['Foo' => 'bar']));
```

`Cake\Http\Response::withBody($body)`

Pour définir le corps de la réponse, utilisez la méthode `withBody()` qui est fournie par le `Laminas\Diactoros\MessageTrait` :

```
$response = $response->withBody($stream);
```

Assurez-vous que `$stream` est un objet de type `Psr\Http\Message\StreamInterface`. Concernant la manière de créer un nouveau stream, voyez ci-dessous.

Vous pouvez également « *streamer* » les réponses depuis des fichiers en utilisant des streams `Laminas\Diactoros\Stream` :

```
// Pour "streamer" depuis un fichier
use Laminas\Diactoros\Stream;

$stream = new Stream('/path/to/file', 'rb');
$response = $response->withBody($stream);
```

Vous pouvez aussi streamer des réponses depuis un callback en utilisant un `CallbackStream`. C'est utile si vous avez des ressources comme des images, des fichiers CSV ou des fichiers PDF à streamer au client :

```
// Streamer depuis un callback
use Cake\Http\CallbackStream;

// Création d'une image
$img = imagecreate(100, 100);
// ...

$stream = new CallbackStream(function () use ($img) {
    imagepng($img);
});
$response = $response->withBody($stream);
```

Définir le Character Set

`Cake\Http\Response::withCharset($charset)`

Cette méthode permet de définir le charset qui sera utilisé dans la réponse :

```
$this->response = $this->response->withCharset('UTF-8');
```

Interagir avec le Cache du Navigateur

`Cake\Http\Response::withDisabledCache()`

Parfois, vous avez besoin de forcer les navigateurs à ne pas mettre en cache les résultats de l'action d'un controller. *Cake\Http\Response::withDisabledCache()* est justement prévue pour cela :

```
public function index()
{
    // Désactive le cache
    $this->response = $this->response->withDisabledCache();
}
```

Avertissement

Désactiver le cache à partir de domaines SSL pendant que vous essayez d'envoyer des fichiers à Internet Explorer peut entraîner des erreurs.

`Cake\Http\Response::withCache($since, $time = '+1 day')`

Vous pouvez aussi dire aux clients que vous voulez qu'ils mettent en cache des réponses. En utilisant *Cake\Http\Response::withCache()* :

```
public function index()
{
    // Autoriser la mise en cache
    $this->response = $this->response->withCache('-1 minute', '+5 days');
}
```

Ce qui est au-dessus indiquera aux clients de mettre en cache la réponse résultante pendant 5 jours, espérant ainsi accélérer l'expérience de vos visiteurs. La méthode `withCache()` définit valeur `Last-Modified` en premier argument. L'entête `Expires` et `max-age` sont définis en se basant sur le second paramètre. Le Cache-Control est défini aussi à `public`.

Configuration fine du Cache HTTP

L'une des meilleures méthodes et des plus simples pour rendre votre application plus rapide est d'utiliser le cache HTTP. Selon ce modèle de mise en cache, vous êtes seulement tenu d'aider les clients à décider s'ils doivent utiliser une copie de la réponse mise en cache en définissant quelques propriétés d'en-têtes comme la date de mise à jour et la balise entity de réponse.

Plutôt que d'avoir à coder la logique de mise en cache et de sa désactivation (rafraîchissement) une fois que les données ont changé, HTTP utilise deux méthodes, l'expiration et la validation qui sont habituellement beaucoup plus simples à utiliser.

En dehors de l'utilisation de *Cake\Http\Response::withCache()*, vous pouvez également utiliser d'autres méthodes pour régler finement les en-têtes de cache HTTP et ainsi tirer profit du cache du navigateur ou du proxy inverse.

L'En-tête de Contrôle du Cache

`Cake\Http\Response::withSharable($public, $time = null)`

Utilisé par la méthode méthode d'expiration, cet en-tête contient de multiples indicateurs qui peuvent changer la façon dont les navigateurs ou les proxies utilisent le contenu mis en cache. Un en-tête `Cache-Control` peut ressembler à ceci :

```
Cache-Control: private, max-age=3600, must-revalidate
```

La classe `Response` vous aide à configurer cet en-tête avec quelques méthodes utiles qui vont produire un en-tête final `Cache Control` valide. La première est la méthode `withSharable()`, qui indique si une réponse peut être considérée comme partageable pour différents utilisateurs ou clients. Cette méthode contrôle en fait la partie *public* ou *private* de cet en-tête. Définir une réponse en *private* indique que tout ou partie de celle-ci est prévue pour un unique utilisateur. Pour tirer profit des mises en cache partagées, il est nécessaire de définir la directive de contrôle en *public*.

Le deuxième paramètre de cette méthode est utilisé pour spécifier un `max-age` pour le cache qui est le nombre de secondes après lesquelles la réponse n'est plus considérée comme récente :

```
public function view()
{
    ...
    // Définit le Cache-Control en public pour 3600 secondes
    $this->response = $this->response->withSharable(true, 3600);
}

public function mes_donnees()
{
    ...
    // Définit le Cache-Control en private pour 3600 secondes
    $this->response = $this->response->withSharable(false, 3600);
}
```

`Response` expose des méthodes séparées pour la définition de chaque component dans l'en-tête de `Cache-Control`.

L'En-tête d'Expiration

`Cake\Http\Response::withExpires($time)`

Vous pouvez définir l'en-tête `Expires` avec une date et un temps après lesquels la réponse n'est plus considérée comme à jour. Cet en-tête peut être défini en utilisant la méthode `withExpires()` :

```
public function view()
{
    $this->response = $this->response->withExpires('+5 days');
}
```

Cette méthode accepte aussi une instance `DateTime` ou toute chaîne de caractère qui peut être parsée par la classe `DateTime`.

L'En-tête Etag

`Cake\Http\Response::withEtag($tag, $weak = false)`

La validation du Cache dans HTTP est souvent utilisée quand le contenu change constamment et demande à l'application de générer seulement les contenus de la réponse si le cache n'est plus à jour. Sous ce modèle, le client continue de stocker les pages dans le cache, mais au lieu de l'utiliser directement, il demande à l'application à chaque fois si les ressources ont changé ou non. C'est utilisé couramment avec des ressources statiques comme les images et autres ressources.

La méthode `withEtag()` (appelée balise d'entité) est une chaîne de caractère qui identifie de façon unique les ressources requêtées comme le fait un checksum pour un fichier, afin de déterminer si elle correspond à une ressource du cache.

Pour réellement tirer profit de l'utilisation de cet en-tête, vous devez soit appeler manuellement la méthode `isNotModified()` ou inclure le *Request Handling (Gestion des requêtes)* dans votre controlleur :

```
public function index()
{
    $articles = $this->Articles->find('all')->all();

    // Somme de contrôle simple du contenu de l'article.
    // Vous devriez utiliser une implémentation plus efficace
    // dans une application du monde réel.
    $checksum = md5(json_encode($articles));

    $response = $this->response->withEtag($checksum);
    if ($response->isNotModified($this->request)) {
        return $response;
    }

    $this->response = $response;
    // ...
}
```

Note

La plupart des utilisateurs proxy devront probablement penser à utiliser l'en-tête Last Modified plutôt que Etags pour des raisons de performance et de compatibilité.

L'En-tête Last-Modified

`Cake\Http\Response::withModified($time)`

De même, avec la méthode consistant à valider du cache HTTP, vous pouvez définir l'en-tête `Last-Modified` pour indiquer la date et l'heure à laquelle la ressource a été modifiée pour la dernière fois. Définir cet en-tête aide CakePHP à indiquer à ces clients si la réponse a été modifiée ou n'est pas basée sur leur cache.

Pour réellement tirer profit de l'utilisation de cet en-tête, vous devez soit appeler manuellement la méthode `isNotModified()` ou inclure le *Request Handling (Gestion des requêtes)* dans votre controlleur :

```
public function view()
{
    $article = $this->Articles->find()->first();
    $response = $this->response->withModified($article->modified);
    if ($response->isNotModified($this->request)) {
        return $response;
    }
    $this->response;
    // ...
}
```

L'En-tête Vary

`Cake\Http\Response::withVary($header)`

Dans certains cas, vous voudrez offrir différents contenus en utilisant la même URL. C'est souvent le cas quand vous avez une page multilingue ou que vous répondez avec différentes pages HTML selon le navigateur qui requête la

ressource. Dans ces circonstances, vous pouvez utiliser l'en-tête `Vary` :

```
$response = $this->response->withVary('User-Agent');
$response = $this->response->withVary('Accept-Encoding', 'User-Agent');
$response = $this->response->withVary('Accept-Language');
```

Envoyer des Réponses Non-Modifiées

`Cake\Http\Response::isNotModified(Request $request)`

Compare les en-têtes de cache pour l'objet requêté avec l'en-tête du cache de la réponse et determine s'il peut toujours être considéré comme à jour. Si oui, il supprime le contenu de la réponse et envoie l'en-tête *304 Not Modified* :

```
// Dans une action de controller.
if ($this->response->isNotModified($this->request)) {
    return $this->response;
}
```

Définir des Cookies

Des cookies peuvent être ajoutés aux réponses en utilisant soit un tableau, soit un objet `Cake\Http\Cookie\Cookie` :

```
use Cake\Http\Cookie\Cookie;
use DateTime;

// Ajoute un cookie
$this->response = $this->response->withCookie(Cookie::create(
    'remember_me',
    'yes',
    // Toutes les clés sont facultatives
    [
        'expires' => new DateTime('+1 year'),
        'path' => '',
        'domain' => '',
        'secure' => false,
        'http' => false,
    ]
]);
```

Référez-vous à la section *Créer des Cookies* pour savoir comment utiliser l'objet Cookie. Vous pouvez utiliser `withExpiredCookie()` pour envoyer un cookie expiré dans la réponse. De cette manière, le navigateur supprimera son cookie local :

```
$this->response = $this->response->withExpiredCookie(new Cookie('remember_me'));
```

Définir les En-têtes de Requête d'Origine Croisée (Cross Origin Request Headers = CORS)

La méthode `cors()` est utilisée pour définir le Contrôle d'Accès HTTP [112] et ses en-têtes liés au travers d'une interface simple :

112. https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS

```
$this->response = $this->response->cors($this->request)
    ->allowOrigin(['*.cakephp.org'])
    ->allowMethods(['GET', 'POST'])
    ->allowHeaders(['X-CSRF-Token'])
    ->allowCredentials()
    ->exposeHeaders(['Link'])
    ->maxAge(300)
    ->build();
```

Les en-têtes liés au CORS vont seulement être appliqués à la réponse si les critères suivants sont vérifiés :

1. La requête a un en-tête `Origin`.
2. La valeur `Origin` de la requête correspond à une des valeurs autorisées de Origin.

Erreurs Communes avec les Responses Immutables

Les objets responses offrent de nombreuses méthodes qui traitent les responses comme des objets immutables. Les objets immutables permettent de prévenir les effets de bord difficiles à repérer. Malgré leurs nombreux avantages, s'habituer aux objets immutables peut prendre un peu de temps. Toutes les méthodes qui commencent par `with` intéragiront avec la réponse à la manière immutable et retourneront **toujours** une **nouvelle** instance. L'erreur la plus fréquente quand les développeurs travaillent avec les objets immutables est d'oublier de persister l'instance modifiée :

```
$this->response->withHeader('X-CakePHP', 'yes!');
```

Dans le code ci-dessus, la réponse ne contiendra pas le header `X-CakePHP` car la valeur retournée par `withHeader()` n'a pas été persistée. Pour avoir un code fonctionnel, vous devrez écrire :

```
$this->response = $this->response->withHeader('X-CakePHP', 'yes!');
```

CookieCollections

class `Cake\Http\Cookie\`**`CookieCollection`**

Les objets `CookieCollection` sont accessibles depuis les objets Request et Response. Ils vous permettent d'intéragir avec des groupes de cookies en utilisant des patterns immutables, ce qui permet au caractère immutable des Request et des Response d'être préservé.

Créer des Cookies

class `Cake\Http\Cookie\`**`Cookie`**

Les objets `Cookie` peuvent être définis via le constructor ou en utilisant l'interface fluide qui suit les patterns immutables :

```
use Cake\Http\Cookie\Cookie;

// Tous les arguments dans le constructor
$cookie = new Cookie(
    'remember_me', // nom
    1, // valeur
    new DateTime('+1 year'), // durée d'expiration, si applicable
    '/', // chemin, si applicable
```

(suite sur la page suivante)

(suite de la page précédente)

```
    'example.com', // domaine, si applicable
    false, // seulement en mode 'secure' ?
    true // seulement en http ?
);

// En utilisant les méthodes immutables
$cookie = (new Cookie('remember_me'))
    ->withValue('1')
    ->withExpiry(new DateTime('+1 year'))
    ->withPath('/')
    ->withDomain('example.com')
    ->withSecure(false)
    ->withHttpOnly(true);
```

Une fois que vous avez créer un cookie, vous pouvez l'ajouter à une nouvelle `CookieCollection`, ou à une existante :

```
use Cake\Http\Cookie\CookieCollection;

// Crée une nouvelle collection
$cookies = new CookieCollection([$cookie]);

// Ajoute à une collection existante
$cookies = $cookies->add($cookie);

// Supprime un cookie via son nom
$cookies = $cookies->remove('remember_me');
```

> **Note**
>
> Gardez bien à l'esprit que les collections sont immutables et qu'ajouter des cookies dans une collection ou retirer des cookies d'une collection va créer *une nouvelle* collection.

Vous devriez utiliser la méthode `withCookie()` pour ajouter des cookies aux objets `Response` :

```
// Ajoute un cookie
$response = $this->response->withCookie($cookie);

// Remplace la collection de cookies
$response = $this->response->withCookieCollection($cookies);
```

Les cookies ajoutés aux Response peuvent être chiffrés en utilisant le encrypted-cookie-middleware

Lire des Cookies

Une fois que vous avez une instance de `CookieCollection`, vous pouvez accéder aux cookies qu'elle contient :

```
// Vérifie l'existence d'un cookie
$cookies->has('remember_me');

// Récupère le nombre de cookie dans une collection
count($cookies);
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Récupère l'instance d'un cookie
$cookie = $cookies->get('remember_me');
```

Une fois que vous avez un objet `Cookie`, vous pouvez intéragir avec son état et le modifier. Gardez à l'esprit que les cookies sont immutables, donc vous allez devoir mettre à jour la collection si vous modifiez un cookie :

```
// Récupère la valeur
$value = $cookie->getValue()

// Accède à une donnée dans une valeur JSON
$id = $cookie->read('User.id');

// Vérifie l'état
$cookie->isHttpOnly();
$cookie->isSecure();
```

CHAPITRE 10

Controllers (Contrôleurs)

class Cake\Controller**Controller**

Les Controllers sont le "C" dans MVC. Après que le routage a été appliqué et que le bon controller a été trouvé, l'action de votre controller est appelée. Votre controller devra gérer l'interprétation des données requêtées, s'assurer que les bons modèles sont appelés et que la bonne réponse ou vue est rendue. Les controllers peuvent être imaginés comme une couche intercalée entre le Modèle et la Vue. Le mieux est de garder des controllers peu chargés, et des modèles plus fournis. Cela vous aidera à réutiliser votre code et facilitera le test de votre code.

Habituellement, les controllers sont utilisés pour gérer la logique autour d'un seul modèle. Par exemple, si vous construisez un site pour gérer une boulangerie en-ligne, vous aurez sans doute un RecettesController qui gère vos recettes et un IngredientsController qui gère vos ingrédients. Cependant, il est aussi possible d'avoir des controllers qui fonctionnent avec plus d'un modèle. Dans CakePHP, un controller est nommé d'après le modèle principal qu'il gère.

Les controllers de votre application sont des classes qui étendent la classe CakePHP `AppController`, qui hérite elle-même de la classe *`Controller`* du cœur. La classe `AppController` peut être définie dans **src/Controller/AppController.php** et elle devra contenir les méthodes partagées par tous les controllers de votre application.

Les controllers peuvent inclure un certain nombre de méthodes qui gèrent les requêtes. Celles-ci sont appelées des *actions*. Par défaut, chaque méthode publique dans un controller est une action accessible via une URL. Une action est responsable de l'interprétation des requêtes et de la création de la réponse. Habituellement, les réponses sont sous forme de vue rendue, mais il y a aussi d'autres façons de créer des réponses.

Le Controller App

Comme indiqué dans l'introduction, la classe `AppController` est la classe mère de tous les controllers de votre application. `AppController` étend elle-même la classe *`Cake\Controller\Controller`* incluse dans la librairie du cœur de CakePHP. `AppController` est définie dans **src/Controller/AppController.php** comme ceci :

```
namespace App\Controller;
```

(suite sur la page suivante)

(suite de la page précédente)

```
use Cake\Controller\Controller;

class AppController extends Controller
{
}
```

Les attributs et méthodes de controller créés dans `AppController` seront disponibles dans tous les controllers de votre application. Les Components (que vous découvrirez plus loin) sont plus appropriés pour du code utilisé dans la plupart des controllers (mais pas nécessairement tous).

Vous pouvez utiliser `AppController` pour charger les components qui seront utilisés dans tous les controllers de votre application. CakePHP fournit une méthode `initialize()` qui est appelée à la fin du constructeur du Controller pour ce type d'utilisation :

```
namespace App\Controller;

use Cake\Controller\Controller;

class AppController extends Controller
{

    public function initialize(): void
    {
        // Active toujours le component CSRF.
        $this->loadComponent('Csrf');
    }

}
```

En plus de la méthode `initialize()`, l'ancienne propriété `$components` vous permettra aussi de déclarer les components qui doivent être chargés. Bien que les règles d'héritage en orienté objet s'appliquent, les components et les helpers utilisés par un controller sont traités spécifiquement. Dans ces cas, les valeurs de la propriété de `AppController` sont fusionnées avec les tableaux de la classe de controller enfant. Les valeurs dans la classe enfant surchargeront toujours celles de `AppController`.

Déroulement d'une Requête

Quand une requête est faîte dans une application CakePHP, les classes `Cake\Routing\Router` et `Cake\Routing\Dispatcher` de CakePHP utilisent la fonctionnalité *Connecter les Routes* pour trouver et créer le bon controller. La requête de données est encapsulée dans un objet request. CakePHP met toutes les informations importantes de la requête dans la propriété `$this->request`. Consultez la section *ServerRequest* pour plus d'informations sur l'objet request de CakePHP.

Les Actions du Controller

Les actions du Controller sont responsables de la conversion des paramètres de la requête dans une réponse pour le navigateur/utilisateur faisant la requête. CakePHP utilise des conventions pour automatiser le processus et retirer quelques codes boiler-plate que vous auriez besoin d'écrire autrement.

Par convention, CakePHP rend une vue avec une version inflectée du nom de l'action. Revenons à notre boulangerie en ligne par exemple, notre RecipesController pourrait contenir les actions `view()`, `share()`, et `search()`. Le controller serait trouvé dans **src/Controller/RecipesController.php** et contiendrait :

```
// src/Controller/RecipesController.php

class RecipesController extends AppController
{
    public function view($id)
    {
        //la logique de l'action va ici.
    }

    public function share($customerId, $recipeId)
    {
        //la logique de l'action va ici.
    }

    public function search($query)
    {
        //la logique de l'action va ici.
    }
}
```

Les fichiers de template pour ces actions seraient **templates/Recipes/view.php**, **templates/Recipes/share.php**, et **templates/Recipes/search.php**. Le nom du fichier de vue est par convention le nom de l'action en minuscules et avec des underscores.

Les actions du Controller utilisent généralement `Controller::set()` pour créer un contexte que `View` utilise pour afficher la couche de vue. Du fait des conventions que CakePHP utilise, vous n'avez pas à créer et rendre la vue manuellement. Au lieu de ça, une fois qu'une action du controller est terminée, CakePHP va gérer le rendu et la livraison de la Vue.

Si pour certaines raisons, vous voulez éviter le comportement par défaut, vous pouvez retourner un objet de *Cake\Http\Response* de l'action avec la response complètement créée.

Afin que vous utilisiez efficacement le controller dans votre propre application, nous allons détailler certains des attributs et méthodes du cœur fournis par les controllers de CakePHP.

Interactions avec les Vues

Les Controllers interagissent avec les vues de plusieurs façons. Premièrement, ils sont capables de passer des données aux vues, en utilisant `Controller::set()`. Vous pouvez aussi décider quelle classe de vue utiliser, et quel fichier de vue doit être rendu à partir du controller.

Définir les Variables de View

`Cake\Controller\Controller::set`(*string $var*, *mixed $value*)

La méthode `Controller::set()` est la principale façon utilisée pour transmettre des données de votre controller à votre vue. Une fois `Controller::set()` utilisée, la variable de votre controller devient accessible dans la vue :

```
// Dans un premier temps vous passez les données depuis le controller:

$this->set('couleur', 'rose');

// Ensuite vous pouvez les utiliser dans la vue de cette manière:
?>
```

(suite sur la page suivante)

(suite de la page précédente)

```
Vous avez sélectionné un glaçage <?= $couleur; ?> pour le gâteau.
```

La méthode `Controller::set()` peut également prendre un tableau associatif comme premier paramètre. Cela peut souvent être une manière rapide d'affecter en une seule fois un jeu complet d'informations à la vue :

```
$data = [
    'couleur' => 'rose',
    'type' => 'sucre',
    'prix_de_base' => 23.95
];

// donne $couleur, $type, et $prix_de_base
// disponible dans la vue:

$this->set($data);
```

Gardez à l'esprit que les variables de vue sont partagées entre toutes les parties rendues par votre vue. Elles seront disponibles dans toutes les parties de la vue : le modèle, la mise en page et tous les éléments à l'intérieur des deux premiers.

Définir les Options d'une View

Si vous voulez personnaliser la classe de vue, les dossiers de layout/template, les helpers ou le thème qui seront utilisés lors du rendu de la vue, vous pouvez utiliser la méthode `viewBuilder()` pour récupérer un constructeur. Ce constructeur peut être utilisé pour définir les propriétés de la vue avant sa création :

```
$this->viewBuilder()
    ->helpers(['MyCustom'])
    ->theme('Modern')
    ->className('Modern.Admin');
```

Le code ci-dessus montre comment charger des helpers personnalisés, définir un thème et utiliser une classe de vue personnalisée.

Rendre une View

`Cake\Controller\Controller::`**`render`**(*string $view*, *string $layout*)

La méthode `Controller::render()` est automatiquement appelée à la fin de chaque action exécutée par le controller. Cette méthode exécute toute la logique liée à la présentation (en utilisant les variables transmises via la méthode `Controller::set()`), place le contenu de la vue à l'intérieur de son `View::$layout` et transmet le tout à l'utilisateur final.

Le fichier de vue utilisé par défaut est déterminé par convention. Ainsi, si l'action `search()` de notre controller RecipesController est demandée, le fichier de vue situé dans **templates/Recipes/search.php** sera utilisé :

```
namespace App\Controller;

class RecipesController extends AppController
{
// ...
    public function search()
```

(suite sur la page suivante)

(suite de la page précédente)

```
    {
        // Rend la vue située dans  templates/Recipes/search.php
        $this->render();
    }
// ...
}
```

Bien que CakePHP appelle cette fonction automatiquement à la fin de chaque action (à moins que vous n'ayez défini `$this->autoRender` à `false`), vous pouvez l'utiliser pour spécifier un fichier de vue alternatif en précisant le nom d'un fichier de vue en premier argument de la méthode `Controller::render()`.

Si `$view` commence par un "/" on suppose que c'est un fichier de vue ou un élément dont le chemin est relatif au dossier **templates**. Cela permet un affichage direct des éléments, ce qui est très pratique lors d'appels AJAX :

```
// Rend un élément dans templates/element/ajaxreturn.php
$this->render('/element/ajaxreturn');
```

Le second paramètre `$layout` de `Controller::render()` vous permet de spécifier le layout de la vue qui est rendue.

Rendre un Template de Vue Spécifique

Dans votre controller, vous pourriez avoir envie de rendre une vue différente de celle rendue par défaut. Vous pouvez le faire en appelant directement `Controller::render()`. Une fois que vous avez appelé `Controller::render()`, CakePHP n'essaiera pas de re-rendre la vue :

```
namespace App\Controller;

class PostsController extends AppController
{
    public function myAction()
    {
        $this->render('custom_file');
    }
}
```

Cela rendrait **templates/Posts/custom_file.php** au lieu de **templates/Posts/my_action.php**.

Vous pouvez aussi rendre les vues des plugins en utilisant la syntaxe suivante : `$this->render('PluginName.PluginController/custom_file')`. Par exemple :

```
namespace App\Controller;

class PostsController extends AppController
{
    public function my_action()
    {
        $this->render('Users.UserDetails/custom_file');
    }
}
```

Cela rendrait la vue **plugins/Users/templates/UserDetails/custom_file.php**

Négociation de Contenu

Les controllers peuvent définir une liste des classes de vue qu'ils proposent. Une fois l'action du controller terminée, CakePHP utilisera cette liste de vues pour réaliser une négociation de contenu. Ainsi, votre application peut réutiliser la même action de controller pour rendre une vue HTML, ou une réponse JSON ou XML. La liste des classes de vue supportées par un controller est définie par la méthode `viewClasses()` :

```
namespace App\Controller;

use Cake\View\JsonView;
use Cake\View\XmlView;

class PostsController extends AppController
{
    public function viewClasses()
    {
        return [JsonView::class, XmlView::class];
    }
}
```

Si aucune autre vue ne peut être sélectionnée d'après l'en-tête `Accept` des requêtes ou l'extension de routage, la classe basique `View` sera automatiquement utilisée comme classe de repli. Si votre application a besoin d'effectuer une logique différente selon les différents formats de réponse, vous pouvez utiliser `$this->request->is()` pour construire la logique conditionnelle dont vous avez besoin.

> **Note**
>
> Les classes de vue doivent implémenter la méthode statique `contentType()` pour pouvoir participer à la négociation de contenu.

Rediriger vers d'Autres Pages

`Cake\Controller\Controller::redirect`(*string|array $url*, *integer $status*)

La méthode `redirect()` ajoute un en-tête `Location` et définit le code d'état de la réponse et la renvoie. Vous devez renvoyer la réponse créée par `redirect()` pour que CakePHP envoie la redirection au lieu de terminer l'action du contrôleur et rendre la vue.

vous pouvez rediriger en utilisant les valeurs du *tableau de routing* :

```
return $this->redirect([
    'controller' => 'Orders',
    'action' => 'confirm',
    $order->id,
    '?' => [
        'product' => 'pizza',
        'quantity' => 5
    ],
    '#' => 'top'
]);
```

Ou utiliser une URL relative ou absolue :

```
return $this->redirect('/orders/confirm');

return $this->redirect('http://www.example.com');
```

Ou rediriger vers l'URL appelante (referer) :

```
return $this->redirect($this->referer());
```

En utilisant le deuxième paramètre, vous pouvez définir un code d'état pour votre redirection :

```
// Effectue un 301 (moved permanently)
return $this->redirect('/order/confirm', 301);

// Effectue un 303 (see other)
return $this->redirect('/order/confirm', 303);
```

Voir la section *Utiliser des Redirections dans les Events des Components* pour savoir comment rediriger hors de un gestionnaire de cycle de vie.

Rediriger vers une Autre Action du Même Controller

`Cake\Controller\Controller::setAction`(*$action*, *$args...*)

Si vous devez rediriger l'action courante vers une autre action du *même* controller, vous pouvez utiliser `Controller::setAction()` pour mettre à jour l'objet request, modifier le template de vue qui va être rendu et rediriger l'exécution vers l'action nommée :

```
// Depuis l'action delete, vous pouvez rendre
// la liste mise à jour.
$this->setAction('index');
```

Chargement des Modèles Supplémentaires

`Cake\Controller\Controller::fetchTable`(*string $alias*, *array $config = []*)

La fonction `fetchTable()` devient pratique quand vous avez besoin d'utiliser une table autre que la table par défaut du controller :

```
// Dans une méthode de controller.
$recentArticles = $this->fetchTable('Articles')->find('all', [
    'limit' => 5,
    'order' => 'Articles.created DESC'
]);
```

Paginer un Modèle

`Cake\Controller\Controller::paginate`()

Cette méthode est utilisée pour paginer les résultats retournés par vos modèles. Vous pouvez définir les tailles de la page, les conditions à utiliser pour la recherche de ces données et bien plus encore. Consultez la section *pagination* pour plus de détails sur l'utilisation de la pagination.

L'attribut `$paginate` vous permet de de personnaliser le comportement de `paginate()` :

```
class ArticlesController extends AppController
{
    public $paginate = [
        'Articles' => [
            'conditions' => ['published' => 1]
        ]
    ];
}
```

Configurer les Components à Charger

`Cake\Controller\Controller::loadComponent($name, $config = [])`

Dans la méthode `initialize()` de votre Controller, vous pouvez définir tout component que vous voulez charger et toute donnée de configuration pour eux :

```
public function initialize(): void
{
    parent::initialize();
    $this->loadComponent('Csrf');
    $this->loadComponent('Comments', Configure::read('Comments'));
}
```

Cycle de Vie des Callbacks de la Requête

Les controllers de CakePHP lancent plusieurs events/callbacks (méthodes de rappel) que vous pouvez utiliser pour insérer de la logique durant tout le cycle de vie de la requête :

Event List

- `Controller.initialize`
- `Controller.startup`
- `Controller.beforeRedirect`
- `Controller.beforeRender`
- `Controller.shutdown`

Callback des Controllers

Par défaut, les méthodes de rappel (callbacks) suivantes sont connectées aux events liés si les méthodes sont implémentées dans vos controllers.

`Cake\Controller\Controller::beforeFilter(EventInterface $event)`

Cette méthode est appelée pendant l'event `Controller.initialize` qui se produit avant chaque action du controller. C'est un endroit pratique pour vérifier le statut d'une session ou les permissions d'un utilisateur.

> **Note**
>
> La méthode beforeFilter() sera appelée pour les actions manquantes.

Retourner une réponse à partir d'une méthode `beforeFilter` ne va pas empêcher l'appel des autres écouteurs du même event. Vous devez explicitement *stopper l'event*.

`Cake\Controller\Controller::beforeRender(EventInterface $event)`

Cette méthode est appelée pendant l'event `Controller.beforeRender` qui se produit après l'action du controller mais avant que la vue ne soit rendue. Ce callback n'est pas souvent utilisé, mais peut-être nécessaire si vous appelez *render()* manuellement à la fin d'une action donnée.

`Cake\Controller\Controller::afterFilter(EventInterface $event)`

Cette méthode est appelée pendant l'event `Controller.shutdown` qui se produit après chaque action du controller, et après que l'affichage soit terminé. C'est la dernière méthode du controller qui est exécutée.

En plus des callbacks des controllers, les *Components (Composants)* fournissent aussi un ensemble similaire de callbacks.

N'oubliez pas d'appeler les callbacks de `AppController` dans les callbacks des controllers enfant pour avoir de meilleurs résultats :

```
//use Cake\Event\EventInterface;
public function beforeFilter(EventInterface $event)
{
    parent::beforeFilter($event);
}
```

Middleware de Controller

`Cake\Controller\Controller::middleware($middleware, array $options = [])`

Les Middlewares peuvent être définis globalement, dans le scope d'une routine ou dans un contrôleur. Pour définir un middleware pour un contrôleur spécifique, utilisez la méthode `middleware()` depuis l'intérieur de la méthode `initialize()` de votre contrôleur :

```
public function initialize(): void
{
    parent::initialize();

    $this->middleware(function ($request, $handler) {
        // Faire la logique du middleware.

        // Assurez-vous de renvoyer une réponse ou d'appeler handle()
        return $handler->handle($request);
    });
}
```

Les middlewares définis par un contrôleur seront appelés **avant** que `beforeFilter()` les méthodes d'action ne soient appelées.

Plus sur les Controllers

Le Controller Pages

Le squelette d'application officiel de CakePHP est livré avec un controller par défaut **PagesController.php**. C'est un controller simple et optionnel qui permet d'afficher un contenu statique. La page d'accueil que vous voyez juste après l'installation est d'ailleurs générée à l'aide de ce controller et du fichier de vue **templates/Pages/home.php**. Ex : Si vous écrivez un fichier de vue **templates/Pages/a_propos.php**, vous pouvez y accéder en utilisant l'url **http ://exemple.com/pages/a_propos**. Vous pouvez modifier le controller Pages selon vos besoins.

Quand vous « cuisinez » une application avec Composer, le controller Pages est créé dans votre dossier **src/Controller/**.

Components (Composants)

Les Components (Composants) sont des regroupements de logique applicative qui sont partagés entre les controllers. CakePHP est également livré avec un fantastique ensemble de components, que vous pouvez utiliser pour vous aider dans de nombreuses tâches communes. Vous pouvez également créer votre propre component. Si vous vous surprenez à vouloir copier et coller des choses entre vos controllers, alors vous devriez envisager de regrouper celle-ci dans un Component. Créer des components permet de garder un code de controller propre et vous permet de réutiliser du code entre différents controllers.

Pour plus d'informations sur les components intégrés dans CakePHP, consultez le chapitre de chaque component :

Authentification

`class AuthComponent`(*ComponentCollection $collection*, *array $config = []*)

Identifier, authentifier et autoriser des utilisateurs constitue une partie courante de nombreuses applications Web. Le component Auth de CakePHP fournit un moyen modulaire d'accomplir cette tâche. Le component Auth vous permet de combiner l'authentification des objets, l'autorisation des objets pour créer un moyen souple pour permettre l'identification et le contrôle des autorisations de l'utilisateur.

Obsolète depuis la version 4.0.0 : Le composant AuthComponent est obsolète à partir de la version 4.0.0 et sera remplacé par les plugins authorization [113] et authentication [114].

Lectures Suggérées Avant de Continuer

La Configuration de l'authentification nécessite quelques étapes, notamment la définition d'une table users, la création d'un model, du controller et des vues, etc..

Tout ceci est couvert étape par étape dans le *Tutorial du Blog*.

Si vous cherchez des solutions existantes pour l'authentification et / ou l'autorisation pour CakePHP, allez jeter un oeil à la section Authentication and Authorization [115] de la CakePHP Awesome List.

Authentification

L'authentification est le processus d'identification des utilisateurs par des identifiants de connexion définis et permet de s'assurer que l'utilisateur est bien celui qu'il prétend être. En général, cela se fait à travers un nom d'utilisateur et un mot de passe, qui sont comparés à une liste d'utilisateurs connus. Dans CakePHP, il y a plusieurs façons intégrées pour l'authentification des utilisateurs enregistrés dans votre application.

- `FormAuthenticate` vous permet d'authentifier les utilisateurs sur la base de formulaire de donnée POST. Habituellement il s'agit d'un formulaire de connexion où les utilisateurs entrent des informations.
- `BasicAuthenticate` vous permet d'identifier les utilisateurs en utilisant l'authentification Basic HTTP.
- `DigestAuthenticate` vous permet d'identifier les utilisateurs en utilisant l'authentification Digest HTTP.

Par défaut Le component Auth (`AuthComponent`) utilise `FormAuthenticate`.

Choisir un Type d'Authentification

En général, vous aurez envie d'offrir l'authentification par formulaire. C'est le plus facile pour les utilisateurs utilisant un navigateur Web. Si vous construisez une API ou un service web, vous aurez peut-être à envisager l'utilisation de l'authentification de base ou l'authentification Digest. L'élément clé qui différencie l'authentification digest de l'authentification basic est la plupart du temps liée à la façon dont les mots de passe sont gérés. Avec l'authentification basic, le nom d'utilisateur et le mot de passe sont transmis en clair sur le serveur. Cela rend l'authentification de base

113. https://book.cakephp.org/authorization/
114. https://book.cakephp.org/authentication/
115. https://github.com/FriendsOfCake/awesome-cakephp/blob/master/README.md#authentication-and-authorization

non appropriée pour des applications sans SSL, puisque vous exposeriez sensiblement vos mots de passe. L'authentification Digest utilise un hachage condensé du nom d'utilisateur, mot de passe, et quelques autres détails. Cela rend l'authentification Digest plus appropriée pour des applications sans cryptage SSL.

Vous pouvez également utiliser des systèmes d'authentification comme OpenID, mais openid ne fait pas parti du cœur de CakePHP.

Configuration des Gestionnaires d'Authentification

Vous configurez les gestionnaires d'authentification en utilisant la config `authenticate`. Vous pouvez configurer un ou plusieurs gestionnaires pour l'authentification. L'utilisation de plusieurs gestionnaires d'authentification vous permet de supporter les différentes méthodes de connexion des utilisateurs. Quand les utilisateurs se connectent, les gestionnaires d'authentification sont utilisés dans l'ordre selon lequel ils ont été déclarés. Une fois qu'un gestionnaire est capable d'identifier un utilisateur, les autres gestionnaires ne seront pas utilisés. Inversement, vous pouvez mettre un terme à toutes les authentifications en levant une exception. Vous devrez traiter toutes les exceptions levées, et les gérer comme désiré.

Vous pouvez configurer le gestionnaire d'authentification dans la méthode `beforeFilter()` ou dans la méthode `initialize()`. Vous pouvez passer l'information de configuration dans chaque objet d'authentification en utilisant un tableau :

```
// Configuration simple
$this->Auth->setConfig('authenticate', ['Form']);

// Passer la configuration
$this->Auth->setConfig('authenticate', [
    'Basic' => ['userModel' => 'Members'],
    'Form' => ['userModel' => 'Members']
]);
```

Dans le deuxième exemple vous pourrez noter que nous avons à déclarer la clé `userModel` deux fois. Pour vous aider à garder un code « propre », vous pouvez utiliser la clé `all`. Cette clé spéciale vous permet de définir les réglages qui sont passés à chaque objet attaché. La clé `all` est aussi utilisée comme cela `AuthComponent::ALL` :

```
// Passer la configuration en utilisant 'all'
$this->Auth->setConfig('authenticate', [
    AuthComponent::ALL => ['userModel' => 'Members'],
    'Basic',
    'Form'
]);
```

Dans l'exemple ci-dessus, à la fois `Form` et `Basic` prendront les paramétrages définis dans la clé « all ». Tous les paramètres transmis à un objet d'authentification particulier remplaceront la clé correspondante dans la clé "all". Les objets d'authentification supportent les clés de configuration suivante.

- `fields` Les champs à utiliser pour identifier un utilisateur. Vous pouvez utiliser les mots clés `username` et `password` pour spécifier respectivement les champs de nom d'utilisateur et de mot de passe.
- `userModel` Le nom du model de la table users, par défaut Users.
- `finder` la méthode finder à utiliser pour récupérer l'enregistrement de l'utilisateur. "all" par défaut.
- `passwordHasher` La classe de hashage de mot de Passe. Par défaut à `Default`.

Pour configurer les différents champs de l'utilisateur dans la méthode `initialize()` :

```
public function initialize(): void
{
    parent::initialize();
    $this->loadComponent('Auth', [
```

(suite sur la page suivante)

(suite de la page précédente)

```
        'authenticate' => [
            'Form' => [
                'fields' => ['username' => 'email', 'password' => 'passwd']
            ]
        ]
    ]);
}
```

Ne mettez pas d'autre clés de configuration de Auth (comme `authError`, `loginAction`, ...) au sein d'élément `authenticate` ou `Form`. Ils doivent se trouver au même niveau que la clé d'authentification. La configuration ci-dessus avec d'autres configurations ressemblerait à quelque chose comme :

```
public function initialize(): void
{
    parent::initialize();
    $this->loadComponent('Auth', [
        'loginAction' => [
            'controller' => 'Users',
            'action' => 'login',
            'plugin' => 'Users'
        ],
        'authError' => 'Vous croyez vraiment que vous pouvez faire cela?',
        'authenticate' => [
            'Form' => [
                'fields' => ['username' => 'email']
            ]
        ],
        'storage' => 'Session'
    ]);
}
```

En plus de la configuration courante, l'authentification de base prend en charge les clés suivantes :

— `realm` Le domaine en cours d'authentification. Par défaut à `env('SERVER_NAME')`.

En plus de la configuration courante, l'authentification Digest prend en charge les clés suivantes :

— `realm` Le domaine en cours d'authentification. Par défaut à servername.
— `nonce` Un nom à usage unique utilisé pour l'authentification. Par défaut à `uniqid()`.
— `qop` Par défaut à auth, pas d'autre valeur supportée pour le moment.
— `opaque` Une chaîne qui doit être retournée à l'identique par les clients. Par Défaut à `md5($config['realm'])`.

Note

Pour récupérer l'enregistrement utilisateur, la requête à la base de données est faite seulement sur le champ « username ». La vérification du mot de passe est faite via PHP. Ceci est nécessaire car les algorithmes de hash comme bcrypt (qui est utilisé par défaut) génèrent un nouveau hash à chaque fois, et ce, pour la même chaîne de caractères. Ceci entraîne l'impossibilité de faire une simple comparaison de chaînes via SQL pour vérifier si le mots de passe correspond.

Personnaliser la Requête de Recherche

Vous pouvez personnaliser la requête utilisée pour chercher l'utilisateur en utilisant l'option `finder` dans la configuration de la classe d'authentification :

```
public function initialize(): void
{
    parent::initialize();
    $this->loadComponent('Auth', [
        'authenticate' => [
            'Form' => [
                'finder' => 'auth'
            ]
        ],
    ]);
}
```

Cela nécessitera que votre table `UsersTable` ait une méthode `findAuth()`. Dans l'exemple ci-dessous, la requête est modifiée pour récupérer uniquement les champs et ajouter une condition. Vous devez vous assurer que vous avez fait un select sur les champs pour lesquels vous souhaitez authentifier un utilisateur, par exemple `username` et `password` :

```
public function findAuth(\Cake\ORM\Query $query, array $options)
{
    $query
        ->select(['id', 'username', 'password'])
        ->where(['Users.active' => 1]);

    return $query;
}
```

Identifier les Utilisateurs et les Connecter

`AuthComponent::identify()`

Vous devez appeler manuellement `$this->Auth->identify()` pour connecter un utilisateur en utilisant les clés fournies dans la requête. Ensuite utilisez `$this->Auth->setUser()` pour connecter l'utilisateur et sauvegarder les infos de l'utilisateur dans la session par exemple.

Quand les utilisateurs s'identifient, les objets d'identification sont vérifiés dans l'ordre où ils ont été attachés. Une fois qu'un objet peut identifier un utilisateur, les autres objets ne sont pas vérifiés. Une simple fonction de connexion pourrait ressembler à cela :

```
public function login()
{
    if ($this->request->is('post')) {
        $user = $this->Auth->identify();
        if ($user) {
            $this->Auth->setUser($user);

            return $this->redirect($this->Auth->redirectUrl());
        } else {
            $this->Flash->error(__("Nom d'utilisateur ou mot de passe incorrect"));
        }
    }
}
```

Le code ci-dessus va d'abord tenter d'identifier un utilisateur en utilisant les données POST. En cas de succès, nous définissons les informations de l'utilisateur dans la session afin qu'elle persiste au cours des requêtes et redirige en cas

de succès vers la dernière page visitée ou vers une URL spécifiée dans la config `loginRedirect`. Si la connexion est un échec, un message flash est défini.

Avertissement

`$this->Auth->setUser($data)` connectera l'utilisateur avec les données postées. Elle ne va pas réellement vérifier l'identité avec une classe d'authentification.

Rediriger les Utilisateurs Après Connexion

`AuthComponent::redirectUrl()`

Après avoir connecté un utilisateur, vous voudrez généralement le rediriger vers l'endroit d'où il vient. Passez une URL pour définir la destination vers laquelle l'utilisateur doit être redirigé après s'être connecté.

Si aucun paramètre n'est passé, l'URL retournée suivra les règles suivantes :

- Retourne l'URL normalisée du paramètre URL redirect s'il est présent et qu'il pointe sur le même domaine que celui de l'application.
- S'il n'y a pas de valeur en session ou en paramètres URL et que la clé `loginRedirect` faisait partie de la configuration de `AuthComponent`, la valeur de `loginRedirect` est retournée.
- S'il n'y a pas de valeur de redirection et que la clé `loginRedirect` n'a pas été configurée, `/` est retournée.

Création de Systèmes d'Authentification Stateless

Les authentifications basic et digest sont des schémas d'authentification sans état (stateless) et ne nécessitent pas un POST initial ou un form. Si vous utilisez seulement les authentificateurs basic/digest, vous n'avez pas besoin d'action login dans votre controller. L'authentication stateless va re-vérifier les autorisations de l'utilisateur à chaque requête, ceci crée un petit surcoût mais permet aux clients de se connecter sans utiliser les cookies et rend AuthComponent plus adapté pour construire des APIs.

Pour des authentificateurs stateless, la config `storage` doit être définie à `Memory` pour que AuthComponent n'utilise pas la session pour stocker l'enregistrement utilisateur. Vous pouvez aussi définir la config `unauthorizedRedirect` à `false` pour que AuthComponent lance une `ForbiddenException` plutôt que le comportement par défaut qui est de rediriger vers la page référente.

Les objets d'authentification peuvent implémenter une méthode `getUser()` qui peut être utilisée pour supporter les systèmes de connexion des utilisateurs qui ne reposent pas sur les cookies. Une méthode getUser typique regarde l'environnement de la requête (request/environnement) et utilise les informations contenues pour confirmer l'identité de l'utilisateur. L'authentification HTTP Basic utilise par exemple `$_SERVER['PHP_AUTH_USER']` et `$_SERVER['PHP_AUTH_PW']` pour les champs username et password.

Note

Dans le cas ou l'authentification ne fonctionne pas telle qu'espérée, vérifiez si les requêtes sont exécutées (voir `BaseAuthenticate::_query($username)`). Dans le cas où aucune requête n'est exécutée, vérifiez si `$_SERVER['PHP_AUTH_USER']` et `$_SERVER['PHP_AUTH_PW']` sont renseignés par le serveur web. Si vous utilisez Apache avec PHP-FastCGI, vous devrez peut être ajouter cette ligne dans le **.htaccess** de votre webroot :

```
RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization},L]
```

Pour chaque requête, ces valeurs, `PHP_AUTH_USER` et `PHP_AUTH_PW`) sont utilisées pour ré-identifier l'utilisateur et s'assurer que c'est un utilisateur valide. Comme avec les méthodes d'authentification de l'objet `authenticate()`, la méthode `getuser()` devrait retourner un tableau d'information utilisateur en cas de succès et `false` en cas d'échec :

```
public function getUser(ServerRequest $request)
{
    $username = env('PHP_AUTH_USER');
    $pass = env('PHP_AUTH_PW');

    if (empty($username) || empty($pass)) {
        return false;
    }

    return $this->_findUser($username, $pass);
}
```

Le contenu ci-dessus montre comment vous pourriez mettre en œuvre la méthode getUser pour les authentifications HTTP Basic. La méthode `_findUser()` fait partie de `BaseAuthenticate` et identifie un utilisateur en se basant sur un nom d'utilisateur et un mot de passe.

Utiliser l'Authentification Basic

L'Authentification Basic vous permet de créer une authentification stateless qui peut être utilisée pour des applications en intranet ou pour des scénarios d'API simple. Les données d'identification de l'authentification Basic seront revérifiés à chaque requête.

> **Avertissement**
>
> L'authentification Basic transmet les données d'identification en clair. Vous devez utiliser HTTPS quand vous utilisez l'authentification Basic.

Pour utiliser l'authentification basic, vous devez configurer AuthComponent :

```
$this->loadComponent('Auth', [
    'authenticate' => [
        'Basic' => [
            'fields' => ['username' => 'username', 'password' => 'api_key'],
            'userModel' => 'Users'
        ],
    ],
    'storage' => 'Memory',
    'unauthorizedRedirect' => false
]);
```

Ici nous voulons utiliser le username + clé API pour nos champs, et utiliser le model Users.

Créer des clés d'API pour une Authentification Basic

Comme le HTTP basic envoie les données d'identification en clair, il n'est pas sage que les utilisateurs envoient leur mot de passe de connexion. A la place, une clé d'API opaque est généralement utilisée. Vous pouvez générer de façon aléatoire ces tokens d'API en utilisant les libraries de CakePHP :

```
namespace App\Model\Table;

use Cake\Auth\DefaultPasswordHasher;
use Cake\Utility\Text;
```

(suite sur la page suivante)

(suite de la page précédente)

```
use Cake\Event\EventInterface;
use Cake\ORM\Table;
use Cake\Utility\Security;

class UsersTable extends Table
{
    public function beforeSave(EventInterface $event)
    {
        $entity = $event->getData('entity');

        if ($entity->isNew()) {
            $hasher = new DefaultPasswordHasher();

            // Génère un 'token' pour l'API
            $entity->api_key_plain = Security::hash(Security::randomBytes(32), 'sha256',␣
↪false);

            // Bcrypt the token so BasicAuthenticate can check
            // it during login.
            $entity->api_key = $hasher->hash($entity->api_key_plain);
        }

        return true;
    }
}
```

Ce qui est au-dessus va générer un hash aléatoire pour chaque utilisateur quand il est sauvegardé. Le code ci-dessus fait l'hypothèse que vous avez deux `api_key` - pour stocker la clé API hashée, et `api_key_plain` - vers la version en clair de la clé API, donc vous pouvez l'afficher à l'utilisateur plus tard. Utiliser une clé plutôt qu'un mot de passe, signifie que même en HTTP en clair, vos utilisateurs peuvent utiliser un token opaque plutôt que leur mot de passe original. Il est aussi sage d'inclure la logique permettant aux clés API d'être régénérées lors de la requête d'un utilisateur.

Utiliser l'Authentification Digest

L'authentification Digest est un modèle qui améliore la sécurité par rapport à l'authentification basic, puisque les certificats d'identification de l'utilisateur ne sont jamais envoyés dans l'en-tête de la requête. A la place, un hash est envoyé.

Pour utiliser l'authentification digest, vous devez configurer AuthComponent :

```
$this->loadComponent('Auth', [
    'authenticate' => [
        'Digest' => [
            'fields' => ['username' => 'username', 'password' => 'digest_hash'],
            'userModel' => 'Users'
        ],
    ],
    'storage' => 'Memory',
    'unauthorizedRedirect' => false
]);
```

Ici nous utilisons le username + digest_hash pour nos champs, et nous utilisons le model Users.

Hasher les Mots de Passe pour l'Authentification Digest

Comme l'authentification Digest nécessite un mot de passe hashé au format défini par la RFC, afin de correctement hasher un mot de passe pour pouvoir l'utiliser avec l'authentification Digest, vous devez utiliser la fonction de hashage de mot de passe spéciale dans `DigestAuthenticate`. Si vous allez combiner l'authentification digest avec une autre stratégie d'authentication, il est aussi recommandé que vous stockiez le mot de passe digest dans une colonne séparée du mot de passe standard hashé :

```
namespace App\Model\Table;

use Cake\Auth\DigestAuthenticate;
use Cake\Event\Event;
use Cake\ORM\Table;

class UsersTable extends Table
{
    public function beforeSave(Event $event)
    {
        $entity = $event->getData('entity');

        // Make a password for digest auth.
        $entity->digest_hash = DigestAuthenticate::password(
            $entity->username,
            $entity->plain_password,
            env('SERVER_NAME')
        );

        return true;
    }
}
```

Les mots de passe pour l'authentification digest ont besoin d'un peu plus d'informations que les autres mots de passe hashés, selon la RFC sur l'authentification digest.

> **Note**
>
> Le troisième paramètre de DigestAuthenticate : :password() doit correspondre à la valeur de config "realm" définie quand DigestAuthentication a été configurée dans AuthComponent : :$authenticate. Celle-ci est `env('SCRIPT_NAME')` par défaut. Vous pouvez souhaiter utiliser une chaîne static si vous voulez des hashs cohérents dans plusieurs environnements.

Créer des Objets d'Authentification Personnalisés

Comme les objets d'authentification sont modulaires, vous pouvez créer des objets d'authentification personnalisés pour votre application ou plugins. Si par exemple vous vouliez créer un objet d'authentification OpenID, dans **src/Auth/OpenidAuthenticate.php**, vous pourriez mettre ce qui suit :

```
namespace App\Auth;

use Cake\Auth\BaseAuthenticate;
use Cake\Http\ServerRequest;
use Cake\Http\Response;
```

(suite sur la page suivante)

(suite de la page précédente)

```
class OpenidAuthenticate extends BaseAuthenticate
{
    public function authenticate(ServerRequest $request, Response $response)
    {
        // Faire les trucs d'OpenID ici.
        // Retourne un tableau de l user si ils peuvent authentifier
        // l utilisateur
        // Retourne false dans le cas contraire
    }
}
```

Les objets d'authentification devraient retourner `false` s'ils ne peuvent identifier l'utilisateur et un tableau d'information utilisateur s'ils le peuvent. Il n'est pas utile d'étendre `BaseAuthenticate`, simplement votre objet d'identification doit implémenter `Cake\Event\EventListenerInterface`. La class `BaseAuthenticate` fournit un nombre de méthode très utiles communément utilisées. Vous pouvez aussi implémenter une méthode `getUser()` si votre objet d'identification doit supporter des authentifications sans cookie ou sans état (stateless). Regardez les sections portant sur l'authentification digest et basic plus bas pour plus d'information.

`AuthComponent` lance maintenant deux événements``Auth.afterIdentify`` et `Auth.logout` respectivement après qu'un utilisateur a été identifié et avant qu'un utilisateur ne soit déconnecté. Vous pouvez définir une fonction de callback pour ces événements en retournant un tableau de mapping depuis la méthode `implementedEvents()` de votre classe d'authentification :

```
public function implementedEvents()
{
    return [
        'Auth.afterIdentify' => 'afterIdentify',
        'Auth.logout' => 'logout'
    ];
}
```

Utilisation d'Objets d'Authentification Personnalisés

Une fois votre objet d'authentification créé, vous pouvez les utiliser en les incluant dans le tableau d'authentification AuthComponents :

```
$this->Auth->config('authenticate', [
    'Openid', // objet d'authentification de app
    'AuthBag.Openid', // objet d'identification de plugin.
]);
```

Note

Notez qu'en utilisant la notation simple, il n'y a pas le mot "Authenticate" lors de l'instantiation de l'objet d'authentification. A la place, si vous utilisez les namespaces, vous devrez définir le namespace complet de la classe (y compris le mot "Authenticate").

Gestion des Requêtes non Authentifiées

Quand un utilisateur non authentifié essaie d'accéder à une page protégée en premier, la méthode `unauthenticated()` du dernier authentificateur dans la chaîne est appelée. L'objet d'authentification peut gérer la réponse d'envoi ou la redirection appropriée en retournant l'objet response pour indiquer qu'aucune action suivante n'est nécessaire du fait de l'ordre dans lequel vous spécifiez l'objet d'authentification dans les propriétés de `authenticate`.

Si l'authentificateur retourne null, *AuthComponent* redirige l'utilisateur vers l'action login. Si c'est une requête ajax et `ajaxLogin` est spécifiée, cet element est rendu sinon un code de statut HTTP 403 est retourné.

Afficher les Messages Flash de Auth

Pour afficher les messages d'erreur de session que Auth génère, vous devez ajouter les lignes de code suivante dans votre layout. Ajoutez les deux lignes suivantes au fichier **templates/layouts/default.php** dans la section body :

```
echo $this->Flash->render();
```

Vous pouvez personnaliser les messages d'erreur et les réglages que le component Auth `AuthComponent` utilise. En utilisant `flash`, vous pouvez configurer les paramètres que le component Auth utilise pour envoyer des messages flash. Les clés disponibles sont

- `key` - La clé à utiliser, "default" par défaut.
- `element` - Le nom de l'élément à utiliser pour le rendu. `null` par défaut.
- `params` - Le tableau des paramètres supplémentaires à utiliser, `[]` par défaut.

En plus des paramètres de message flash, vous pouvez personnaliser les autres messages d'erreurs que le component AuthComponent utilise. Dans la partie beforeFilter de votre controller ou dans le paramétrage du component, vous pouvez utiliser `authError` pour personnaliser l'erreur à utiliser quand l'authentification échoue :

```
$this->Auth->config('authError', 'Désolé, vous n'êtes pas autorisés à accéder à cette␣
↪zone.');
```

Parfois, vous voulez seulement afficher l'erreur d'autorisation après que l'user se soit déjà connecté. Vous pouvez supprimer ce message en configurant sa valeur avec le booléen `false`.

Dans le `beforeFilter()` de votre controller ou dans la configuration du component :

```
if (!$this->Auth->user()) {
    $this->Auth->config('authError', false);
}
```

Hachage des Mots de Passe

Vous êtes responsable du hashage des mots de passe avant qu'ils soient stockés dans la base de données, la façon la plus simple est d'utiliser une fonction directrice (setter) dans votre entity User :

```
namespace App\Model\Entity;

use Cake\Auth\DefaultPasswordHasher;
use Cake\ORM\Entity;

class User extends Entity
{

    // ...

    protected function _setPassword($password)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    {
        if (strlen($password) > 0) {
          return (new DefaultPasswordHasher)->hash($password);
        }
    }

    // ...
}
```

AuthComponent est configuré par défaut pour utiliser `DefaultPasswordHasher` lors de la validation des informations d'identification de l'utilisateur si aucune configuration supplémentaire est requise afin d'authentifier les utilisateurs.

`DefaultPasswordHasher` utilise l'algorithme de hashage bcrypt en interne, qui est l'une des solutions les plus fortes pour hasher un mot de passe dans l'industrie. Bien qu'il soit recommandé que vous utilisiez la classe de hash de mot de passe, il se peut que vous gériez une base de données d'utilisateurs dont les mots de passe ont été hashés différemment.

Créer des Classes de Hash de Mot de Passe Personnalisé

Pour utiliser un hasher de mot de passe différent, vous devez créer la classe dans **src/Auth/LegacyPasswordHasher.php** et intégrer les méthodes `hash()` et `check()`. Cette classe doit étendre la classe `AbstractPasswordHasher` :

```
namespace App\Auth;

use Cake\Auth\AbstractPasswordHasher;

class LegacyPasswordHasher extends AbstractPasswordHasher
{

    public function hash($password)
    {
        return sha1($password);
    }

    public function check($password, $hashedPassword)
    {
        return sha1($password) === $hashedPassword;
    }
}
```

Ensuite, vous devez configurer AuthComponent pour utiliser votre propre hasher de mot de passe :

```
public function initialize(): void
{
    parent::initialize();
    $this->loadComponent('Auth', [
        'authenticate' => [
            'Form' => [
                'passwordHasher' => [
                    'className' => 'Legacy',
                ]
            ]
        ]
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ]);
}
```

Supporter des systèmes hérités est une bonne idée mais il est encore mieux de garder votre base de données avec les derniers outils de sécurité. La section suivante va expliquer comment migrer d'un algorithme de hash vers celui par défaut de CakePHP.

Changer les Algorithmes de Hashage

CakePHP fournit un moyen propre de migrer vos mots de passe utilisateurs d'un algorithme vers un autre, ceci est possible avec la classe `FallbackPasswordHasher`. Supposons que vous migriez votre application depuis CakePHP 2.x qui utilise des hash de mot de passe `sha1`, vous pouvez configurer le AuthComponent comme suit :

```
public function initialize(): void
{
    parent::initialize();
    $this->loadComponent('Auth', [
        'authenticate' => [
            'Form' => [
                'passwordHasher' => [
                    'className' => 'Fallback',
                    'hashers' => [
                        'Default',
                        'Weak' => ['hashType' => 'sha1']
                    ]
                ]
            ]
        ]
    ]);
}
```

Le premier nom qui apparait dans la clé `hashers` indique quelle classe est la préférée et elle réservera les autres dans la liste si la vérification n'est pas un succès.

Quand vous utilisez `WeakPasswordHasher`, vous devez définir la valeur de configuration `Security.salt` pour vous assurer que les mots de passe sont bien chiffrés avec cette valeur salt.

Afin de mettre à jour les anciens mot de passe des utilisateurs à la volée, vous pouvez changer la fonction login selon :

```
public function login()
{
    if ($this->request->is('post')) {
        $user = $this->Auth->identify();
        if ($user) {
            $this->Auth->setUser($user);
            if ($this->Auth->authenticationProvider()->needsPasswordRehash()) {
                $user = $this->Users->get($this->Auth->user('id'));
                $user->password = $this->request->getData('password');
                $this->Users->save($user);
            }

            return $this->redirect($this->Auth->redirectUrl());
        }
```

(suite sur la page suivante)

(suite de la page précédente)

```
        ...
    }
}
```

Comme vous pouvez le voir, nous définissons le mot de passe en clair à nouveau pour que la fonction directrice (setter) dans l'entity hashe le mot de passe comme montré dans les exemples précédents et sauvegarde ensuite l'entity.

Connecter les Utilisateurs Manuellement

`AuthComponent::setUser(array $user)`

Parfois, le besoin se fait sentir de connecter un utilisateur manuellement, par exemple juste après qu'il se soit enregistré dans votre application. Vous pouvez faire cela en appelant `$this->Auth->setUser()` avec les données utilisateur que vous voulez pour la "connexion" :

```
public function register()
{
    $user = $this->Users->newEntity($this->request->getData());
    if ($this->Users->save($user)) {
        $this->Auth->setUser($user->toArray());

        return $this->redirect([
            'controller' => 'Users',
            'action' => 'home'
        ]);
    }
}
```

Avertissement

Assurez-vous d'ajouter manuellement le nouveau User id au tableau passé à la méthode de `setUser()`. Sinon vous n'aurez pas l'id utilisateur disponible.

Accéder à l'Utilisateur Connecté

`AuthComponent::user($key = null)`

Une fois que l'utilisateur est connecté, vous avez souvent besoin d'information particulière à propos de l'utilisateur courant. Vous pouvez accéder à l'utilisateur en cours de connexion de la façon suivante :

```
// Depuis l'intérieur du controler
$this->Auth->user('id');
```

Si l'utilisateur courant n'est pas connecté ou que la clé n'existe pas, la valeur null sera retournée.

Déconnexion des Utilisateurs

`AuthComponent::logout()`

Éventuellement, vous aurez besoin d'un moyen rapide pour dés-authentifier les utilisateurs et les rediriger où ils devraient aller. Cette méthode est aussi très pratique si vous voulez fournir un lien "Me déconnecter" à l'intérieur de la zone membres de votre application :

```
public function logout()
{
    return $this->redirect($this->Auth->logout());
}
```

La déconnexion des utilisateurs connectés avec l'authentification Basic ou Digest est difficile à accomplir pour tous les clients. La plupart des navigateurs retiennent les autorisations pendant qu'il restent ouvert. Certains navigateurs peuvent être forcés en envoyant un code 401. Le changement du realm de l'authentification est une autre solution qui fonctionne pour certain clients.

Décider quand lancer l'Authentification

Dans certains cas, vous aurez peut-être envie d'utiliser `$this->Auth->user()` dans la méthode `beforeFilter()`. C'est possible en utilisant la clé de config `checkAuthIn`. Ce qui suit modifie les vérifications initiales d'authentification qui doivent être faites pour un event en particulier :

```
//Définit AuthComponent pour authentifier dans initialize()
$this->Auth->config('checkAuthIn', 'Controller.initialize');
```

La valeur par défaut pour `checkAuthIn` est `'Controller.startup'` - mais en utilisant `'Controller.initialize'`, l'authentification initiale est faite avant la méthode `beforeFilter()`.

Autorisation

L'autorisation est le processus qui permet de s'assurer qu'un utilisateur identifié/authentifié est autorisé à accéder aux ressources qu'il demande. S'il est activé, `AuthComponent` peut vérifier automatiquement des gestionnaires d'autorisations et veiller à ce que les utilisateurs connectés soient autorisés à accéder aux ressources qu'ils demandent. Il y a plusieurs gestionnaires d'autorisations intégrés et vous pouvez créer vos propres gestionnaires pour votre application ou comme faisant partie d'un plugin par exemple.

— `ControllerAuthorize` appelle `isAuthorized()` sur le controller actif et utilise ce retour pour autoriser un utilisateur. C'est souvent le moyen le plus simple d'autoriser les utilisateurs.

> **Note**
>
> Les adaptateurs `ActionsAuthorize` & `CrudAuthorize` disponibles dans CakePHP 2.x ont été déplacés dans un plugin séparé cakephp/acl [116].

Configurer les Gestionnaires d'Autorisation

Vous configurez les gestionnaires d'autorisation en utilisant la clé de config `authorize`. Vous pouvez configurer un ou plusieurs gestionnaires pour l'autorisation. L'utilisation de plusieurs gestionnaires vous donne la possibilité d'utiliser plusieurs moyens de vérifier les autorisations. Quand les gestionnaires d'autorisation sont vérifiés, ils sont appelés dans l'ordre où ils sont déclarés. Les gestionnaires devraient retourner `false`, s'il ne sont pas capable de vérifier les autorisations ou bien si la vérification a échoué. Les gestionnaires devraient retourner `true` s'ils sont capables de vérifier avec succès les autorisations. Les gestionnaires seront appelés dans l'ordre jusqu'à ce que l'un d'entre eux retourne `true`. Si toutes les vérifications échouent, l'utilisateur sera redirigé vers la page d'où il vient. Vous pouvez également stopper les autorisations en levant une exception. Vous aurez besoin d'attraper toutes les exceptions levées et de les traiter.

Vous pouvez configurer les gestionnaires d'autorisations dans l'une des méthodes `beforeFilter()` ou `initialize()` de votre controller. Vous pouvez passer les informations de configuration dans chaque objet d'autorisation en utilisant un tableau :

116. https://github.com/cakephp/acl

```
// paramétrage Basique
$this->Auth->config('authorize', ['Controller']);

// passage de paramètre
$this->Auth->config('authorize', [
    'Actions' => ['actionPath' => 'controllers/'],
    'Controller'
]);
```

Tout comme avec `authenticate`, `authorize`, vous pouvez utiliser la clé `all` pour vous aider à garder un code propre. Cette clé spéciale vous aide à définir les paramètres qui sont passés à chaque objet attaché. La clé `all` est aussi exposée comme `AuthComponent::ALL` :

```
// Passer la configuration en utilisant 'all'
$this->Auth->config('authorize', [
    AuthComponent::ALL => ['actionPath' => 'controllers/'],
    'Actions',
    'Controller'
]);
```

Dans l'exemple ci-dessus, à la fois l'`Action` et le `Controller` auront les paramètres définis pour la clé "all". Chaque paramètre passé à un objet d'autorisation spécifique remplacera la clé correspondante dans la clé "all".

Si un utilisateur authentifié essaie d'aller à une URL pour laquelle il n'est pas autorisé, il est redirigé vers l'URL de référence. Si vous ne voulez pas cette redirection (souvent nécessaire quand vous utilisez un adaptateur d'authentification stateless), vous pouvez définir l'option de configuration `unauthorizedRedirect` à `false`. Cela fait que AuthComponent lance une `ForbiddenException` au lieu de rediriger.

Création d'Objets Authorize Personnalisés

Parce que les objets authorize sont modulables, vous pouvez créer des objets authorize personnalisés dans votre application ou plugins. Si par exemple vous voulez créer un objet authorize LDAP dans **src/Auth/LdapAuthorize.php**, vous pourriez mettre cela :

```
namespace App\Auth;

use Cake\Auth\BaseAuthorize;
use Cake\Http\ServerRequest;

class LdapAuthorize extends BaseAuthorize
{
    public function authorize($user, ServerRequest $request)
    {
        // Faire des choses pour ldap ici.
    }
}
```

Les objets Authorize devraient retourner `false` si l'utilisateur se voit refuser l'accès ou si l'objet est incapable de faire un contrôle. Si l'objet est capable de vérifier l'accès de l'utilisateur, `true` devrait être retourné. Il n'est pas nécessaire d'étendre `BaseAuthorize`, il faut simplement que votre objet authorize implémente la méthode `authorize()`. La classe `BaseAuthorize` fournit un nombre intéressant de méthodes utiles qui sont communément utilisées.

Utilisation d'Objets Authorize Personnalisés

Une fois que vous avez créé votre objet authorize personnalisé, vous pouvez l'utiliser en l'incluant dans le tableau authorize :

```
$this->Auth->config('authorize', [
    'Ldap', // Objet d'autorisation de l'application.
    'AuthBag.Combo', // Objet d'autorisation d'un plugin.
]);
```

Ne pas Utiliser d'Autorisation

Si vous souhaitez ne pas utiliser les objets d'autorisation intégrés et que vous voulez gérer les choses entièrement à l'extérieur du Component Auth (`AuthComponent`), vous pouvez définir `$this->Auth->setConfig('authorize', false);`. Par défaut, le component Auth démarre avec `authorize` à `false`. Si vous n'utilisez pas de schéma d'autorisation, assurez-vous de vérifier les autorisations vous-même dans la partie `beforeFilter()` de votre controller ou avec un autre component.

Rendre des Actions Publiques

`AuthComponent::allow`(*$actions = null*)

Il y a souvent des actions de controller que vous souhaitez laisser entièrement publiques ou qui ne nécessitent pas de connexion utilisateur. Le component Auth (`AuthComponent`) est pessimiste et par défaut interdit l'accès. Vous pouvez marquer des actions comme publique en utilisant `AuthComponent::allow()`. En marquant les actions comme publique, le component Auth ne vérifiera pas la connexion d'un utilisateur, ni n'autorisera la vérification des objets :

```
// Permet toutes les actions
$this->Auth->allow();

// Ne permet que l'action view.
$this->Auth->allow('view');

// Ne permet que les actions view et index.
$this->Auth->allow(['view', 'index']);
```

En l'appellant sans paramètre, vous autorisez toutes les actions à être publiques. Pour une action unique, vous pouvez fournir le nom comme une chaine, sinon utiliser un tableau.

> **Note**
>
> Vous ne devez pas ajouter l'action « login » de votre `UsersController` dans la liste des allow. Le faire entraînera des problèmes sur le fonctionnement normal de `AuthComponent`.

Fabriquer des Actions qui requièrent des Autorisations

`AuthComponent::deny`(*$actions = null*)

Par défaut, toutes les actions nécessitent une authorisation. Cependant, si après avoir rendu les actions publiques, vous voulez révoquer les accès publics, vous pouvez le faire en utilisant `AuthComponent::deny()` :

```
// retire toutes les actions .
$this->Auth->deny();
```

(suite sur la page suivante)

(suite de la page précédente)

```
// retire une action
$this->Auth->deny('add');

// retire un groupe d'actions.
$this->Auth->deny(['add', 'edit']);
```

En l'appellant sans paramètre, cela interdira toutes les actions. Pour une action unique, vous pouvez fournir le nom comme une chaine, sinon utiliser un tableau.

Utilisation de ControllerAuthorize

ControllerAuthorize vous permet de gérer les vérifications d'autorisation dans le callback d'un controller. C'est parfait quand vous avez des autorisations très simples ou que vous voulez utiliser une combinaison models + components pour faire vos autorisations et que vous ne voulez pas créer un objet authorize personnalisé.

Le callback est toujours appelé `isAuthorized()` et devrait retourner un booléen pour indiquer si l'utilisateur est autorisé ou pas à accéder aux ressources de la requête. Le callback est passé à l'utilisateur actif, ainsi il peut donc être vérifié :

```
class AppController extends Controller
{
    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('Auth', [
            'authorize' => 'Controller',
        ]);
    }

    public function isAuthorized($user = null)
    {
        // Chacun des utilisateurs enregistrés peut accéder aux fonctions publiques
        if (!$this->request->getParam('prefix')) {
            return true;
        }

        // Seulement les administrateurs peuvent accéder aux fonctions d'administration
        if ($this->request->getParam('prefix') === 'admin') {
            return (bool)($user['role'] === 'admin');
        }

        // Par défaut n'autorise pas
        return false;
    }
}
```

Le callback ci-dessus fournirait un système d'autorisation très simple où seuls les utilisateurs ayant le rôle d'administrateur pourraient accéder aux actions qui ont le préfixe admin.

Options de Configuration

Les configurations suivantes peuvent toutes être définies soit dans la méthode `initialize()` de votre controller, soit en utilisant `$this->Auth->setConfig()` dans votre `beforeFilter()` :

ajaxLogin
Le nom d'une vue optionnelle d'un élément à rendre quand une requête AJAX est faite avec une session expirée invalide.

allowedActions
Les actions du controller pour lesquelles la validation de l'utilisateur n'est pas nécessaire.

authenticate
Défini comme un tableau d'objets d'identifications que vous voulez utiliser quand les utilisateurs de connectent. Il y a plusieurs objets d'authentification dans le noyau, cf la section *Lectures Suggérées Avant de Continuer*.

authError
Erreur à afficher quand les utilisateurs font une tentative d'accès à un objet ou une action à laquelle ils n'ont pas accès.

Vous pouvez supprimer les messages authError de l'affichage par défaut en mettant cette valeur au booléen `false`.

authorize
Défini comme un tableau d'objets d'autorisation que vous voulez utiliser quand les utilisateurs sont autorisés sur chaque requête, cf la section *Autorisation*

flash
Paramétrage à utiliser quand Auth à besoin de faire un message flash avec `FlashComponent::set()`. Les clés disponibles sont :
— `element` - L'élément à utiliser , par défaut à "default".
— `key` - La clé à utiliser, par défaut à "auth".
— `params` - Un tableau de paramètres supplémentaires à utiliser par défaut à []

loginAction
Une URL (définie comme une chaîne de caractères ou un tableau) pour l'action du controller qui gère les connexions. Par défaut à `/users/login`.

loginRedirect
L" URL (définie comme une chaîne de caractères ou un tableau) pour l'action du controller où les utilisateurs doivent être redirigés après la connexion. Cette valeur sera ignorée si l'utilisateur à une valeur `Auth.redirect` dans sa session.

logoutRedirect
L'action par défaut pour rediriger l'utilisateur quand il se déconnecte. Lorsque le component Auth ne gère pas les redirection post-logout, une URL de redirection sera retournée depuis `AuthComponent::logout()`. Par défaut à `loginAction`.

unauthorizedRedirect
Contrôle la gestion des accès non autorisés. Par défaut, un utilisateur non autorisé est redirigé vers l'URL référente, `loginAction` ou `/`. Si défini à `false`, une exception ForbiddenException est lancée au lieu de la redirection.

storage
Classe de stockage à utiliser pour faire persister les enregistrements utilisateurs. Lors de l'utilisation d'un authenticator personnalisé, vous devriez définir cette option à `Memory`. Par défaut à `Session`. Vous pouvez passer des options de config pour stocker une classe en utilisant le format de tableau. Par exemple, pour utiliser une clé de session personnalisée, vous pouvez définir `storage` avec `['className' => 'Session', 'key' => 'Auth.Admin']`.

checkAuthIn
Le nom de l'event pour lequel les vérifications de l'authentification doivent être faites. Défaut à `Controller.startup`. Vous pouvez le spécifier à `Controller.initialize` si vous souhaitez que les vérifications soient faites avant que l'action `beforeFilter()` du controller soit executée.

Aussi, `$this->Auth->getConfig()` vous permet d'obtenir une valeur de configuration en appelant seulement l'option de configuration :

```
$this->Auth->getConfig('loginAction');

return $this->redirect($this->Auth->getConfig('loginAction'));
```

Utile si vous souhaitez rediriger un utilisateur sur la page `login` par exemple. Sans option, la configuration complète sera retournée.

Tester des Actions Protégées par AuthComponent

Regardez la section *Tester Des Actions Qui Nécessitent Une Authentification* pour avoir des astuces sur la façon de tester les actions de controller qui sont protégées par `AuthComponent`.

Flash

class Cake\Controller\Component**FlashComponent**(*ComponentCollection $collection*, *array $config = []*)

FlashComponent est un moyen de définir des messages de notifications à afficher après avoir traité les données envoyées via un formulaire ou acquitter des informations. CakePHP appelle ces messages des « messages flash ». FlashComponent écrit les messages flash dans `$_SESSION` pour être affichés dans une View en utilisant *FlashHelper*.

Définir les Messages Flash

FlashComponent fournit deux façons de définir des messages flash : sa méthode magique `__call()` et sa méthode `set()`. Pour remplir votre application sémantiquement, la méthode magique `__call()` de FlashComponent vous permet d'utiliser un nom de méthode qui est lié à un element qui se trouve dans le répertoire **templates/element/flash**. Par convention, les méthodes en camelcase vont être liées à un nom d'élément en minuscule et avec des underscores (_) :

```
// Utilise templates/element/flash/success.php
$this->Flash->success('C\'était un succès');

// Utilise templates/element/flash/great_success.php
$this->Flash->greatSuccess('C\'était un grand succès');
```

De façon alternative, pour définir un message sans rendre un element, vous pouvez utiliser la méthode `set()` :

```
$this->Flash->set('Ceci est un message');
```

Les messages Flash sont ajoutés à un tableau en interne. Les appels successifs à `set()` ou `__call()` avec la même clé ajoutera les messages dans `$_SESSION`. Si vous souhaitez écraser les messages existants lors de la définition d'un flash message, mettez l'option `clear` sur `` `true `` lors de la configuration du composant.

Les méthodes `__call()` et `set()` de FlashComponent prennent de façon optionnelle un deuxième paramètre, un tableau d'options :

- `key` Par défaut à "flash". La clé du tableau trouvé sous la clé "Flash" dans la session.
- `element` Par défaut à null, mais il va automatiquement être défini lors de l'utilisation de la méthode magique `__call()`. Le nom d'élément à utiliser pour le rendu.
- `params` Un tableau en option de clés/valeurs pour rendre disponible des variables dans un element.
- `clear` attend un `bool` et permet d'effacer tous les messages de la pile courante et d'en démarrer une nouvelle.

Un exemple de l'utilisation de ces options :

```
// Dans votre Controller
$this->Flash->success("L'utilisateur a été sauvegardé", [
```

(suite sur la page suivante)

(suite de la page précédente)

```
    'key' => 'positive',
    'params' => [
        'name' => $user->name,
        'email' => $user->email
    ]
]);

// Dans votre Vue
<?= $this->Flash->render('positive') ?>

<!-- Dans templates/element/flash/success.php -->
<div id="flash-<?= h($key) ?>" class="message-info success">
    <?= h($message) ?>: <?= h($params['name']) ?>, <?= h($params['email']) ?>.
</div>
```

Notez que le paramètre `element` sera toujours surchargé en utilisant `__call()`. Afin de récupérer un element spécifique d'un plugin, vous devez définir le paramètre `plugin`. Par exemple :

```
// Dans votre Controller
$this->Flash->warning('My message', ['plugin' => 'PluginName']);
```

Le code ci-dessus va utiliser l'element **warning.php** dans **plugins/PluginName/templates/element/flash** pour afficher le message flash.

> **Note**
>
> Par défaut, CakePHP échappe le contenu dans les messages flash pour des raisons de sécurité. Si vous utilisez une requête ou des données d'utilisateur dans vos messages flash, ceux-ci sont échappés et donc sécurisés pour l'affichage. Si vous souhaitez afficher du HTML, vous devez passer un paramètre `escape` et aussi ajuster les templates pour permettre la désactivation de l'échappement quand un tel paramètre est passé.

HTML dans des Messages Flash

Il est possible d'afficher le HTML dans des messages flash en utilisant la clé d'option `'escape'` :

```
$this->Flash->info(sprintf('<b>%s</b> %s', h($highlight), h($message)), ['escape' =>␣
↪false]);
```

Assure-vous de bien échapper l'input manuellement, ensuite. Dans l'exemple ci-dessus, `$highlight` et `$message` sont des inputs non-HTML et donc sont échappés.

Pour plus d'informations sur le rendu de vos messages flash, consultez la section *FlashHelper*.

SecurityComponent (Sécurité)

class SecurityComponent(*ComponentCollection $collection*, *array $config = []*)

Le Component Security apporte une sécurité renforcée à votre application. Il fournit des méthodes pour diverses tâches comme :

- Restreindre les méthodes HTTP que votre application accepte.
- Protection contre la falsification de formulaire.
- Exiger l'utilisation du SSL.
- Limiter les communications croisées dans le controller.

Comme tous les components, il est configuré au travers de plusieurs paramètres configurables. Toutes ces propriétés peuvent être définies directement ou au travers de « méthodes setter » du même nom dans la partie `beforeFilter()` de votre controller.

En utilisant le Component Security vous obtenez automatiquement une protection contre la falsification de formulaire. Des jetons de champs cachés seront automatiquement insérés dans les formulaires et vérifiés par le component Security.

Si vous utilisez la fonctionnalité de protection des formulaires par le component Security et que d'autres components traitent des données de formulaire dans les callbacks `startup()`, assurez-vous de placer le component Security avant ces components dans la méthode `initialize()`.

> **Note**
>
> Quand vous utilisez le component Security vous **devez** utiliser le Helper Form (FormHelper) pour créer vos formulaires. De plus, vous **ne** devez surcharger **aucun** des attributs des champs " « name ». Le component Security regarde certains indicateurs qui sont créés et gérés par le Helper form. (spécialement ceux créés dans *`create()`*) et *`end()`*). La modification dynamique des champs qui lui sont soumis dans une requête POST (ex. désactiver, effacer, créer des nouveaux champs via Javascript) est susceptible de déclencher un black-holing (envoi dans le trou noir) de la requête.
>
> Vous devez toujours vérifier les méthodes HTTP utilisées avant d'exécuter d'autre code. Vous devez *vérifier la méthode HTTP* ou utiliser *`Cake\Http\ServerRequest::allowMethod()`* pour vous assurer que la bonne méthode HTTP est utilisée.

Gestion des callbacks Blackhole

`SecurityComponent::blackHole`(*object $controller*, *string $error = ''*, *SecurityException $exception = null*)

Si une action est restreinte par le component Security, elle devient un black-hole (trou noir), comme une requête invalide qui aboutira à une erreur 400 par défaut. Vous pouvez configurer ce comportement, en définissant l'option de configuration `blackHoleCallback` par une fonction de rappel (callback) dans le controller.

En configurant la fonction de rappel, vous pouvez personnaliser le processus de mise en trou noir (blackhole callback) :

```
public function beforeFilter(Event $event)
{
    $this->Security->setConfig('blackHoleCallback', 'blackhole');
}

public function blackhole($type)
{
    // Gère les erreurs.
}
```

> **Note**
>
> utilisez `$this->Security->config()` pour les versions de CakePHP inférieures à 3.4.0.

Le paramètre `$type` peut avoir les valeurs suivantes :
- "auth" Indique une erreur de validation de formulaire, ou une incohérence controller/action.
- "secure" Indique un problème sur la méthode de restriction SSL.

Restreindre les actions aux actions SSL

`SecurityComponent::requireSecure()`

Définit les actions qui nécessitent une requête SSL-securisée. Prend un nombre indéfini de paramètres. Peut-être appelé sans argument, pour forcer toutes les actions à requérir une SSL-securisée.

`SecurityComponent::requireAuth()`

Définit les actions qui nécessitent un jeton valide généré par le component Security. Prend un nombre indéfini de paramètres. Peut-être appelé sans argument, pour forcer toutes les actions à requérir une authentification valide.

Restreindre les Demandes croisées de Controller

allowedControllers

Une liste de controllers qui peuvent envoyer des requêtes vers ce controller. Ceci peut être utilisé pour contrôler les demandes croisées de controller.

allowedActions

Une liste des actions qui peuvent envoyer des requêtes vers les actions de ce controller. Ceci peut être utilisé pour contrôler les demandes croisées de controller.

Prévention de la Falsification de Formulaire

Par défaut le component Security `SecurityComponent` empêche l'utilisation de la falsification de formulaire. Le `SecurityComponent` va empêcher les choses suivantes :

- Les champs inconnus ne peuvent être ajoutés au formulaire.
- Les champs ne peuvent être retirés du formulaire.
- Les valeurs dans les inputs cachés ne peuvent être modifiées.

La prévention de ces types de falsification est faite de concert avec FormHelper, en recherchant les champs qui sont dans un formulaire. Les valeurs pour les champs cachés sont aussi utilisées. Toutes ces données sont combinées et il en ressort un hash. Quand un formulaire est soumis, SecurityComponent va utiliser les données POSTées pour construire la même structure et comparer le hash.

> **Note**
>
> SecurityComponent **ne** va **pas** empêcher aux options sélectionnées d'être ajoutées/changées. Ni ne va empêcher les options radio d'être ajoutées/changées.

unlockedFields

Définit une liste de champs de formulaire à exclure de la validation POST. Les champs peuvent être déverrouillés dans le component ou avec `FormHelper::unlockField()`. Les champs qui ont été déverrouillés ne sont pas requis faisant parti du POST et les champs cachés déverrouillés n'ont pas leur valeur vérifiée.

validatePost

Défini à `false` pour complètement éviter la validation des requêtes POST, essentiellement éteindre la validation de formulaire.

Les options de configuration ci-dessus peuvent être _set_ via la méthode `setConfig()` ou `config()` si vous utilisez une version de CakePHP avant 3.4.0.

Utilisation

Le component Security est généralement utilisé dans la méthode `beforeFilter()` de votre controller. Vous pouvez spécifier les restrictions de sécurité que vous voulez et le component Security les forcera au démarrage :

```
namespace App\Controller;

use App\Controller\AppController;
use Cake\Event\EventInterface;

class WidgetsController extends AppController
{

    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('Security');
    }

    public function beforeFilter(EventInterface $event)
    {
        if ($this->request->getParam('admin')) {
            $this->Security->requireSecure();
        }
    }
}
```

Cette exemple forcera toutes les actions qui proviennent de la « route » Admin à être effectuées via des requêtes sécurisées :

```
namespace App\Controller;

use App\Controller\AppController;
use Cake\Event\EventInterface;

class WidgetsController extends AppController
{

    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('Security', ['blackHoleCallback' => 'forceSSL']);
    }

    public function beforeFilter(EventInterface $event)
    {
        if ($this->request->getParam('admin')) {
            $this->Security->requireSecure();
        }
    }

    public function forceSSL()
    {
        return $this->redirect('https://' . env('SERVER_NAME') . $this->request->
↪getRequestTarget());
    }
}
```

> **Note**
>
> Utilisez `$this->request-here()` pour les versions de CakePHP avant 3.4.0

Cet exemple forcera toutes les actions qui proviennent de la « route » admin à requérir des requêtes sécurisés SSL. Quand la requête est placée dans un trou noir, elle appellera le callback `forceSSL()` qui redirigera automatiquement les requêtes non sécurisées vers les requêtes sécurisées.

Protection CSRF

CSRF ou Cross Site Request Forgery est une vulnérabilité courante pour les applications Web. Cela permet à un attaquant de capturer et de rejouer une requête, et parfois de soumettre des demandes de données en utilisant les balises images ou des ressources sur d'autres domaines. Pour activer la protection CSRF, utilisez CSRF Middleware.

Désactiver le Component Security pour des Actions Spécifiques

Il peut arriver que vous souhaitiez désactiver toutes les vérifications de sécurité pour une action (ex. ajax request). Vous pouvez « délocker » ces actions en les listant dans `$this->Security->unlockedActions` dans votre `beforeFilter()`. La propriété `unlockedActions` **ne** va **pas** avoir d'effets sur les autres fonctionnalités de `SecurityComponent` :

```
namespace App\Controller;

use App\Controller\AppController;
use Cake\Event\EventInterface;

class WidgetController extends AppController
{

    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('Security');
    }

    public function beforeFilter(EventInterface $event)
    {
         $this->Security->setConfig('unlockedActions', ['edit']);
    }
}
```

> **Note**
>
> Utilisez `$this->Security->config()` pour les versions de CakePHP inférieures à 3.4.0.

Cet exemple désactiverait toutes les vérifications de sécurité pour une action edit.

Pagination

class Cake\Controller\Component**PaginatorComponent**

Obsolète depuis la version 4.4.0 : Le Component paginator est déprécié depuis 4.4.0 et sera supprimé dans 5.0.

L'un des principaux défis pour créer une application flexible et ergonomique est la conception d'une interface utilisateur intuitive. De nombreuses applications ont tendance à augmenter rapidement en taille et en complexité, et les designers comme les programmeurs se retrouvent incapables de faire face à l'affichage de centaines ou de milliers d'enregistrements. Réécrire prend du temps, et les performances et la satisfaction des utilisateurs peuvent en pâtir.

Afficher un nombre raisonnable d'enregistrements par page a toujours été une partie critique dans toutes les applications et cause régulièrement de nombreux maux de tête aux développeurs. CakePHP allège le fardeau des développeurs en fournissant un moyen efficace de paginer les données.

La pagination dans CakePHP se fait par un Component dans le controller, pour faciliter la création des requêtes de pagination. Vous utilisez ensuite *PaginatorHelper* dans vos templates de vue pour générer les contrôles de pagination.

Usage basique

Pour paginer une requête, nous commençons par charger le `PaginatorComponent` :

```
class ArticlesController extends AppController
{
    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('Paginator');
    }
}
```

Une fois qu'il est chargé, nous pouvons paginer une classe de table de l'ORM ou un objet `Query` :

```
public function index()
{
    // Paginer une table de l'ORM.
    $this->set('articles', $this->paginate($this->Articles));

    // Paginer une requête en cours de construction.
    $query = $this->Articles->find('published');
    $this->set('articles', $this->paginate($query));
}
```

Utilisation avancée

Le `PaginatorComponent` peut supporter des cas de figure plus complexes en configurant la propriété `$paginate` du controller ou l'argument `$settings` de `paginate()`. Ces conditions servent de base pour vos requêtes de pagination. On leur ajoute ensuite les paramètres `sort`, `direction`, `limit`, et `page` passés dans l'URL :

```
class ArticlesController extends AppController
{

    public $paginate = [
        'limit' => 25,
        'order' => [
            'Articles.title' => 'asc'
```

(suite sur la page suivante)

(suite de la page précédente)

```
        ]
    ];
}
```

Astuce

L'option par défaut `order` doit être définie dans un tableau.

Cependant vous pouvez passer dans vos paramètres de pagination n'importe quelle option supportée par *find()*, telle que `fields`. Il est souvent plus propre et simple de mettre vos options de pagination dans une *Méthodes Finder Personnalisées*. vous pouvez définir l'utilisation de la pagination du finder en configurant l'option `findType` :

```
class ArticlesController extends AppController
{
    public $paginate = [
        'finder' => 'published',
    ];
}
```

Si votre finder attend d'autres options, vous pouvez les passer comme des valeurs pour le finder :

```
class ArticlesController extends AppController
{

    // trouve les articles selon les tags
    public function tags()
    {
        $tags = $this->request->getParam('pass');

        $optionsPersonnaliseesPourLeFinder = [
            'tags' => $tags
        ];
        // Nous utilisons ici l'argument $settings pour paginate().
        // Mais on pourrait utiliser la même structure dans $this->paginate
        //
        // Notre finder personnalisé s'appelle findTagged dans ArticlesTable.php
        // C'est pourquoi nous utilisons une clé `tagged`.
        // Voici à quoi devrait ressembler notre finder:
        // public function findTagged(Query $query, array $options) {
        $settings = [
            'finder' => [
                'tagged' => $optionsPersonnaliseesPourLeFinder
            ]
        ];

        $articles = $this->paginate($this->Articles, $settings);

        $this->set(compact('articles', 'tags'));
    }
}
```

En plus de définir les valeurs de pagination générales, vous pouvez définir plusieurs jeux de pagination par défaut dans

votre controller. Vous pouvez utiliser le nom de chaque modèle comme clé dans la propriété `$paginate` :

```
class ArticlesController extends AppController
{
    public $paginate = [
        'Articles' => [],
        'Authors' => [],
    ];
}
```

Les tableaux sous les clés `Articles` et `Auteurs` peuvent contenir toutes les clés que `$paginate` peut contenir habituellement.

Une fois que nous avons utilisé `paginate()` pour créer des résultats, des paramètres de pagination seront ajoutés à l'objet `request` du controller. Vous pouvez accéder aux métadonnées de pagination par `$this->request->getAttribute('paging')`.

Pagination Simple

Par défaut, la pagination utilise une requête `count()` pour calculer le nombre de résultats, de manière à pouvoir afficher les liens vers des numéros de pages. Sur de grandes quantités de données, ce décompte peut devenir très coûteux. Dans les situations où n'avez besoin que des boutons "Précédent" et "Suivant", vous pouvez utiliser le paginateur "simple" qui ne lance pas de requête de comptage :

```
public function initialize(): void
{
    parent::initialize();

    // Charger le component de pagination avec la stratégie simple.
    $this->loadComponent('Paginator', [
        'className' => 'Simple',
    ]);
}
```

En utilisant le `SimplePaginator`, vous ne pourrez pas générer des liens vers des numéros de pages, ni des compteurs de données, ni un lien vers la dernière page, ni le nombre total d'enregistrements.

Utiliser Paginator Directement

Si vous devez paginer des données depuis un autre component, vous pouvez utiliser directement PaginatorComponent. Il fournit une API similaire à la méthode du controller :

```
$articles = $this->Paginator->paginate($articleTable->find(), $config);

// Ou
$articles = $this->Paginator->paginate($articleTable, $config);
```

Le premier paramètre doit être l'objet query à partir d'un find sur l'objet table dont vous souhaitez paginer les résultats. Au choix, vous pouvez aussi passer l'objet table et laisser le paginator construire la requête pour vous. Le second paramètre est le tableau de configuration à utiliser pour la pagination. Ce tableau doit avoir la même structure que la propriété `$paginate` dans un controller. Quand on pagine un objet `Query`, l'option `finder` sera ignorée. Vous devez passer la query que vous souhaitez paginer.

Paginer Plusieurs Requêtes

Vous pouvez paginer plusieurs models dans la même action du controller en utilisant l'option `scope` à la fois dans la propriété `$paginate` du controller et dans l'appel à la méthode `paginate()` :

```
// Paginate property
public $paginate = [
    'Articles' => ['scope' => 'article'],
    'Tags' => ['scope' => 'tag']
];

// Dans une action de controller
$articles = $this->paginate($this->Articles, ['scope' => 'article']);
$tags = $this->paginate($this->Tags, ['scope' => 'tag']);
$this->set(compact('articles', 'tags'));
```

L'option `scope` va faire que `PaginatorComponent` va regarder les paramètres de query string scopés. Par exemple, l'URL suivante pourrait être utilisée pour paginer les tags et les articles en même temps :

```
/dashboard?article[page]=1&tag[page]=3
```

Consulter la section *Paginer Plusieurs Résultats* pour savoir comment générer les éléments HTML scopés et les URLS pour la pagination.

Pour paginer plusieurs fois le même modèle dans une même action du controller, vous devez définir un alias du modèle. Consultez *Utiliser le TableLocator* pour plus de détails sur l'utilisation du registre de tables :

```
// Dans une action de controller
$this->paginate = [
    'ArticlesTable' => [
        'scope' => 'published_articles',
        'limit' => 10,
        'order' => [
            'id' => 'desc',
        ],
    ],
    'UnpublishedArticlesTable' => [
        'scope' => 'unpublished_articles',
        'limit' => 10,
        'order' => [
            'id' => 'desc',
        ],
    ],
];

$publishedArticles = $this->paginate(
    $this->Articles->find('all', [
        'scope' => 'published_articles'
    ])->where(['published' => true])
);

// Charge un autre objet table pour permettre de les différencier dans le component de␣
↪pagination
$unpublishedArticlesTable = $this->fetchTable('UnpublishedArticles', [
    'className' => 'App\Model\Table\ArticlesTable',
```

(suite sur la page suivante)

(suite de la page précédente)

```
    'table' => 'articles',
    'entityClass' => 'App\Model\Entity\Article',
]);

$unpublishedArticles = $this->paginate(
    $unpublishedArticlesTable->find('all', [
        'scope' => 'unpublished_articles'
    ])->where(['published' => false])
);
```

Contrôler les Champs Utilisés pour le Tri

Par défaut, vous pouvez trier sur n'importe quelle colonne non virtuelle d'une table. Ce n'est pas toujours souhaitable puisque cela permet aux utilisateurs de trier sur des colonnes non indexées qui peuvent être longues à trier. Vous pouvez définir la liste des champs pouvant être triés en utilisant l'option `sortableFields`. Cette option est nécessaire quand vous voulez trier sur des données associées, ou des champs calculés qui peuvent faire partie de la requête de pagination :

```
public $paginate = [
    'sortableFields' => [
        'id', 'title', 'Users.username', 'created'
    ]
];
```

Toute requête qui tentera de trier les champs qui ne sont pas dans cette liste sera ignorée.

Limiter le Nombre Maximum de Lignes par Page

Le nombre de résultats récupérés pour chaque page peut être configuré par l'utilisateur dans le paramètre `limit`. D'une manière générale, il n'est pas souhaitable que l'utilisateur puisse récupérer toutes les lignes d'un ensemble paginé. L'option `maxLimit` permet que personne ne puisse définir de limite trop haute depuis l'extérieur. Par défaut, CakePHP limite à un maximum de 100 le nombre de lignes par page. Si cette valeur par défaut n'est pas appropriée pour votre application, vous pouvez l'ajuster dans les options de pagination, par exemple en le réduisant à `10` :

```
public $paginate = [
    // Autres clés ici.
    'maxLimit' => 10
];
```

Si le paramètre de limite de la requête est plus grand que cette valeur, elle sera réduite à la valeur `maxLimit`.

Faire des Jointures d'Associations Supplémentaires

Vous pouvez charger des associations supplémentaires depuis la table paginée en utilisant le paramètre `contain` :

```
public function index()
{
    $this->paginate = [
        'contain' => ['Authors', 'Comments']
    ];

    $this->set('articles', $this->paginate($this->Articles));
}
```

Requêtes de Page Inexistante

Quand on essaie d'accéder à une page inexistante, c'est-à-dire lorsque le numéro de page demandé est supérieur au nombre total de pages, PaginatorComponent lance une `NotFoundException`.

Ainsi, vous avez le choix entre laisser la page d'erreur normale s'afficher, ou utiliser un bloc try catch et exécuter des actions appropriées lorsqu'une `NotFoundException` est attrapée :

```
use Cake\Http\Exception\NotFoundException;

public function index()
{
    try {
        $this->paginate();
    } catch (NotFoundException $e) {
        // Faire quelque chose ici, comme rediriger vers la première ou la dernière page.
        // $this->request->getAttribute('paging') vous donnera les infos nécessaires.
    }
}
```

Pagination dans la Vue

Consultez la documentation *PaginatorHelper* pour savoir comment créer des liens de navigation dans la pagination.

Request Handling (Gestion des requêtes)

class RequestHandlerComponent(*ComponentCollection $collection*, *array $config = []*)

Le component Request Handler est utilisé dans CakePHP pour obtenir des informations supplémentaires au sujet des requêtes HTTP qui sont faites à votre application. Vous pouvez l'utiliser pour informer vos controllers des processus AJAX, autant que pour obtenir des informations complémentaires sur les types de contenu que le client accepte et modifie automatiquement dans le layout approprié, quand les extensions de fichier sont disponibles.

Par défaut, le RequestHandler détectera automatiquement les requêtes AJAX en se basant sur le header `X-Requested-With`, qui est utilisé par de nombreuses librairies JavaScript. Quand il est utilisé conjointement avec `Cake\Routing\Router::extensions()`, RequestHandler changera automatiquement le layout et les fichiers de template par ceux qui correspondent à des types de média non-HTML. En outre, s'il existe un helper avec le même nom que l'extension demandée, il sera ajouté au tableau des helpers des Controllers. Enfin, si une donnée XML/JSON est POST'ée vers vos Controllers, elle sera décomposée dans un tableau qui est assigné à `$this->request->getData()`, et pourra alors être accessible comme vous le feriez pour n'importe quelle donnée POST. Afin d'utiliser le Request Handler, il doit être chargé depuis la méthode `initialize()` :

```
class WidgetController extends AppController
{

    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('RequestHandler');
    }

    // suite du controller
}
```

Obtenir des informations sur une requête

Request Handler contient plusieurs méthodes qui fournissent des informations à propos du client et de ses requêtes.

`RequestHandlerComponent::accepts($type = null)`

$type peut être une chaîne, un tableau, ou "null". Si c'est une chaîne, la méthode accepts() renverra `true` si le client accepte ce type de contenu. Si c'est un tableau, accepts() renverra `true` si un des types du contenu est accepté par le client. Si c'est "null", elle renverra un tableau des types de contenu que le client accepte. Par exemple :

```
class ArticlesController extends AppController
{

    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('RequestHandler');
    }

    public function beforeFilter(EventInterface $event)
    {
        if ($this->RequestHandler->accepts('html')) {
            // Execute le code seulement si le client accepte une
            // response HTML  (text/html).
        } elseif ($this->RequestHandler->accepts('xml')) {
            // Execute uniquement le code XML
        }
        if ($this->RequestHandler->accepts(['xml', 'rss', 'atom'])) {
            // Execute si le client accepte l'une des réponses
            // ci-dessus: XML, RSS ou Atom.
        }
    }
}
```

D'autres méthodes de détections du contenu des requêtes :

`RequestHandlerComponent::isXml()`

Renvoie `true` si la requête actuelle accepte les réponses XML.

`RequestHandlerComponent::isRss()`

Renvoie `true` si la requête actuelle accepte les réponses RSS.

`RequestHandlerComponent::isAtom()`

Renvoie `true` si l'appel actuel accepte les réponse Atom, `false` dans le cas contraire.

`RequestHandlerComponent::isMobile()`

Renvoie `true` si le navigateur du client correspond à un téléphone portable, ou si le client accepte le contenu WAP. Les navigateurs mobiles supportés sont les suivants :

- Android
- AvantGo
- BlackBerry
- DoCoMo
- Fennec
- iPad
- iPhone
- iPod

- J2ME
- MIDP
- NetFront
- Nokia
- Opera Mini
- Opera Mobi
- PalmOS
- PalmSource
- portalmmm
- Plucker
- ReqwirelessWeb
- SonyEricsson
- Symbian
- UP.Browser
- webOS
- Windows CE
- Windows Phone OS
- Xiino

`RequestHandlerComponent::isWap()`

Retourne `true` si le client accepte le contenu WAP.

Toutes les méthodes de détection des requêtes précédentes peuvent être utilisées dans un contexte similaire pour filtrer les fonctionnalités destinées à du contenu spécifique. Par exemple, au moment de répondre aux requêtes AJAX, si vous voulez désactiver le cache du navigateur, et changer le niveau de débogage. Cependant, si vous voulez utiliser le cache pour les requêtes non-AJAX., le code suivant vous permettra de le faire :

```
if ($this->request->is('ajax')) {
    $this->response->disableCache();
}
// Continue l'action du controller
```

Décoder Automatiquement les Données de la Requête

Ajoute une requête de décodage de données. Le gestionnaire devrait contenir un callback, et tout autre argument supplémentaire pour le callback. Le callback devrait retourner un tableau de données contenues dans la requête. Par exemple, ajouter un gestionnaire de CSV pourrait ressembler à ceci :

```
class ArticlesController extends AppController
{
    public function initialize(): void
    {
        parent::initialize();
        $parser = function ($data) {
            $rows = str_getcsv($data, "\n");
            foreach ($rows as &$row) {
                $row = str_getcsv($row, ',');
            }

            return $rows;
        };
        $this->loadComponent('RequestHandler', [
            'inputTypeMap' => [
```

(suite sur la page suivante)

(suite de la page précédente)

```
                'csv' => [$parser]
            ]
        ]);
    }
}
```

Vous pouvez utiliser n'importe quel callback [117] pour la fonction de gestion. Vous pouvez aussi passer des arguments supplémentaires au callback, c'est très utile pour les callbacks comme `json_decode` :

```
$this->RequestHandler->config('inputTypeMap.json', ['json_decode', true]);
```

Le contenu ci-dessus créera `$this->request->getData()` un tableau des données d'entrées JSON, sans le `true` supplémentaire vous obtiendrez un jeu d'objets `stdClass`.

Vérifier les Préférences de Content-Type

`RequestHandlerComponent::prefers`(*$type = null*)

Détermine les content-types que le client préfère. Si aucun paramètre n'est donné, le type de contenu le plus approchant est retourné. Si $type est un tableau, le premier type que le client accepte sera retourné. La préférence est déterminée, premièrement par l'extension de fichier analysée par Router, s'il y en avait une de fournie et secondairement, par la liste des content-types définis dans `HTTP_ACCEPT` :

```
$this->RequestHandler->prefers('json');
```

Répondre aux Requêtes

`RequestHandlerComponent::renderAs`(*$controller*, *$type*)

Change le mode de rendu d'un controller pour le type spécifié. Ajoutera aussi le helper approprié au tableau des helpers du controller, s'il est disponible et qu'il n'est pas déjà dans le tableau :

```
// Force le controller à rendre une response xml.
$this->RequestHandler->renderAs($this, 'xml');
```

Cette méthode va aussi tenter d'ajouter un helper qui correspond au type de contenu courant. Par exemple si vous rendez un `rss`, `RssHelper` sera ajouté.

`RequestHandlerComponent::respondAs`(*$type*, *$options*)

Définit l'en-tête de réponse basé sur la correspondance content-type/noms. Cette méthode vous laisse définir un certain nombre de propriétés de response en une seule fois :

```
$this->RequestHandler->respondAs('xml', [
    // Force le téléchargement
    'attachment' => true,
    'charset' => 'UTF-8'
]);
```

`RequestHandlerComponent::responseType`()

Retourne l'en-tête Content-type du type de réponse actuel ou null s'il y en a déjà un de défini.

117. https://php.net/callback

Profiter du cache de validation HTTP

Le model de validation de cache HTTP est l'un des processus utilisé pour les passerelles de cache, aussi connu comme reverse proxies, pour déterminer si elles peuvent servir une copie de réponse stockée au client. D'après ce model, vous bénéficiez surtout d'une meilleur bande passante, mais utilisé correctement vous pouvez aussi gagner en temps de processeur, et ainsi gagner en temps de réponse.

En activant le Component RequestHandler dans votre controller vous validerez le contrôle automatique effectué avant de rendre une vue. Ce contrôle compare l'objet réponse à la requête originale pour déterminer si la réponse n'a pas été modifiée depuis la dernière fois que le client a fait sa demande.

Si la réponse est évaluée comme non modifiée, alors le processus de rendu de vues est arrêter, réduisant le temps processeur. Un `no content` est retourné au client, augmentant la bande passante. Le code de réponse est défini à *304 Not Modified*.

Vous pouvez mettre en retrait ce contrôle automatique en paramétrant `checkHttpCache` à `false` :

```
public function initialize(): void
{
    parent::initialize();
    $this->loadComponent('RequestHandler', [
        'checkHttpCache' => false
    ]);
}
```

Utiliser les ViewClasses personnalisées

Quand vous utilisez JsonView/XmlView, vous aurez envie peut-être de surcharger la serialization par défaut avec une classe View par défaut, ou ajouter des classes View pour d'autres types.

Vous pouvez mapper les types existants et les nouveaux types à vos classes personnalisées. Vous pouvez aussi définir ceci automatiquement en utilisant la configuration `viewClassMap` :

```
public function initialize(): void
{
    parent::initialize();
    $this->loadComponent('RequestHandler', [
        'viewClassMap' => [
            'json' => 'ApiKit.MyJson',
            'xml' => 'ApiKit.MyXml',
            'csv' => 'ApiKit.Csv'
        ]
    ]);
}
```

Protection de Formulaire

class FormProtection(*ComponentCollection $collection*, *array $config = []*)

Le Component FormProtection fournit une protection contre l'altération des données de formulaire.

Comme tous les components, il dispose de plusieurs paramèters de configuration. Chacune de ces propriétés peut être définie directement ou par des *setters* du même nom, dans les méthodes `initialize()` ou `beforeFilter()` de votre controller.

Si vous utilisez d'auutres components qui traitent des données de formulaire dans leurs callbacks `startup()`, veillez à placer le Component FormProtection avant ceux-ci dans votre méthode `initialize()`.

> **Note**
>
> Quand vous utilisez le Component FormProtection vous **devez** utiliser le FormHelper pour créer vos formulaires. De plus, vous **ne devez pas** réécrire les attributs « name » des champs. Le Component FormProtection observe certains indicateurs créés et gérés par le FormHelper (en particulier ceux qui sont créés dans *create()* et *end()*). L'altération dynamique des champs soumis dans une requête POST (par exemple désactiver, supprimer ou créer de nouveaux champs via JavaScript) est susceptible d'entraîner l'échec de validation du jeton de formulaire.

Prévention des altérations de formulaire

Par défaut, le `FormProtectionComponent` empêche certaines altérations de formulaire par les utilisateurs. Il les empêche :

- de modifier l'action du formulaire (URL)
- d'ajouter des champs inconnus
- de supprimer des champs du formulaire
- de modifier des valeurs dans des inputs cachés.

La prévention de ces altérations fonctionne en collaboration avec le `FormHelper`, et consiste à tracer les champs du formulaire. Les valeurs des champs cachés sont également tracées. Toutes ces données sont hachées, et des jetons cachés sont insérés automatiquement dans le formulaire. Quand le formulaire est transmis, le `FormProtectionComponent` utilise les données POST pour reconstruire la même structure et comparer le hash.

> **Note**
>
> Le FormProtectionComponent **n'empêchera pas** l'ajout et la modification d'options dans les select. Il n'empêchera pas non plus l'ajout ou la modification d'options radio.

Utilisation

La configuration du component security se fait généralement dans les callbacks `initialize()` ou `beforeFilter()` du controller.

Les options possibles sont :

validate
: Définir à `false` pour passer complètement la validation des requêtes POST, désactivant de fait l'essentiel de la validation de formulaires.

unlockedFields
: Défini comme une liste de champs à exclure de la validation POST. Les champs peuvent être déverrouillés (*unlocked*) soit dans le Component, soit avec `FormHelper::unlockField()`. Les champs déverrouillés peuvent être absents du POST et les valeurs des champs cachés déverrouillés ne sont pas vérifiées.

unlockedActions
: Actions à exclure des vérifications de validation de POST.

validationFailureCallback
: Callback à appeler en cas d'échec de validation. Doit être une Closure valide. Non défini par défaut, auquel cas une exception est lancée en cas d'échec de validation.

Désactiver les vérifications d'altération de formulaire

```
namespace App\Controller;

use App\Controller\AppController;
use Cake\Event\EventInterface;
```

(suite sur la page suivante)

(suite de la page précédente)

```
class WidgetsController extends AppController
{
    public function initialize(): void
    {
        parent::initialize();

        $this->loadComponent('FormProtection');
    }

    public function beforeFilter(EventInterface $event)
    {
        parent::beforeFilter($event);

        if ($this->request->getParam('prefix') === 'Admin') {
            $this->FormProtection->setConfig('validate', false);
        }
    }
}
```

L'exemple ci-dessus désactiverait la prévention d'altération de formulaire pour les routes préfixées par `Admin`.

Désactiver l'altération de formulaire pour certaines actions

Il peut y avoir des cas dans lesquels vous voudrez désactiver la prévention d'altération de formulaire pour une action (par exemple des requêtes AJAX). Vous pouvez « déverrouiller » ces actions en les listant dans `$this->Security->unlockedActions` dans votre `beforeFilter()` :

```
namespace App\Controller;

use App\Controller\AppController;
use Cake\Event\EventInterface;

class WidgetController extends AppController
{
    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('FormProtection');
    }

    public function beforeFilter(EventInterface $event)
    {
        parent::beforeFilter($event);

        $this->FormProtection->setConfig('unlockedActions', ['edit']);
    }
}
```

Cet exemple désactiverait toutes les vérifications de sécurité pour l'action `edit`.

Gestion des échecs de validation par des callbacks

Si la validation de protection du formulaire échoue, elle renverra par défaut une erreur 400. Vous pouvez configurer ce comportement en définissant l'option de configuration `validationFailureCallback` vers une fonction callback du controller.

En configurant une méthode de callback, vous pouvez personnaliser le mode de fonctionnement de la gestion d'erreur :

```
public function beforeFilter(EventInterface $event)
{
    parent::beforeFilter($event);

    $this->FormProtection->setConfig(
        'validationFailureCallback',
        function (BadRequestException $exception) {
            // Vous pouvez soit renvoyer une instance response, soit lever
            // l'exception reçue en argument.
        }
    );
}
```

Checking HTTP Cache

class CheckHttpCacheComponent(*ComponentCollection $collection*, *array $config = []*)

Le modèle de validation du cache HTTP est un des process utilisés pour les passerelles de cache, aussi connues comme mandataires inversés (*reverse proxies*), pour déterminer si elles peuvent servir à stocker une copie de la réponse au client. Dans ce modèle, vous économisez surtout de la bande passante, mais quand vous l'utilisez correctement vous pouvez également économiser du CPU, et réduire les temps de réponse :

```
// dans un Controller
public function initialize(): void
{
    parent::initialize();

    $this->addComponent('CheckHttpCache');
}
```

Le fait d'activer `CheckHttpCacheComponent` dans votre controller active automatiquement une vérification de `beforeRender`. Cette vérification compare les en-têtes de cache définies dans l'objet response avec les en-têtes de cache envoyées dans la requête, pour déterminer si la réponse n'a pas été modifiée depuis la dernière fois que le client l'a demandée. Les en-têtes de requête utilisées sont les suivantes :

- `If-None-Match` est comparée avec l'en-tête `Etag` de la réponse.
- `If-Modified-Since` est comparée avec l'en-tête `Last-Modified` de la réponse.

Si les en-têtes de la réponse correspondent aux critères des en-têtes de la requête, alors on saute l'étape de rendu de la vue. Cela évite à votre application de générer la vue, ce qui économise de la bande passante et du temps. Lorsque les en-têtes de la réponse correspondent, on renvoie une réponse vide avec le code de status `304 Not Modified`.

Configuration des Components

De nombreux components du cœur nécessitent une configuration. Quleques exemples de composants qui requièrent une configuration sont *SecurityComponent (Sécurité)* et *Request Handling (Gestion des requêtes)*. La configuration pour ces components, et pour les components en général, se fait via `loadComponent()` dans la méthode `initialize()` de votre Controller ou via le tableau `$components` :

```
class PostsController extends AppController
{
    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('RequestHandler', [
            'viewClassMap' => ['json' => 'AppJsonView'],
        ]);
        $this->loadComponent('Security', ['blackholeCallback' => 'blackhole']);
    }
}
```

Vous pouvez configurer les components à la volée en utilisant la méthode `setConfig()`. Souvent, ceci est fait dans la méthode `beforeFilter()` de votre controller. Ceci peut aussi être exprimé comme ceci :

```
public function beforeFilter(EventInterface $event)
{
    $this->RequestHandler->setConfig('viewClassMap', ['rss' => 'MyRssView']);
}
```

Comme les helpers, les components ont des méthodes `getConfig()` et `setConfig()` qui sont utilisées pour récupérer et définir toutes les configurations pour un component :

```
// Lire des données de config.
 $this->RequestHandler->getConfig('viewClassMap');

// Définir la config
$this->Csrf->setConfig('cookieName', 'token');
```

Comme avec les helpers, les components vont automatiquement fusionner leur propriété `$_defaultConfig` avec la configuration du constructeur pour créer la propriété `$_config` qui est accessible avec `getConfig()` et `setConfig()`.

Faire des Alias avec les Components

Un paramètre commun à utiliser est l'option `className`, qui vous autorise à faire des alias des components. Cette fonctionnalité est utile quand vous voulez remplacer `$this->Auth` ou une autre référence habituelle de Component avec une implémentation sur mesure :

```
// src/Controller/PostsController.php
class PostsController extends AppController
{
    public function initialize(): void
    {
        parent::initialize('Auth', [
            'className' => 'MyAuth'
        ]);
    }
}

// src/Controller/Component/MyAuthComponent.php
use Cake\Controller\Component\AuthComponent;

class MyAuthComponent extends AuthComponent
```

(suite sur la page suivante)

(suite de la page précédente)

```
{
    // Ajoutez votre code pour surcharge l'AuthComponent du cœur
}
```

Le code ci-dessus fera un *alias* `MyAuthComponent` de `$this->Auth` dans vos controllers.

> **Note**
>
> Faire un alias à un component remplace cette instance n'importe où où le component est utilisé, en incluant l'intérieur des autres Components.

Charger les Components à la Volée

Vous n'avez parfois pas besoin de rendre le component accessible sur chaque action du controller. Dans ce cas là, vous pouvez le charger à la volée en utilisant la méthode `loadComponent()` à l'intérieur de votre controller :

```
// Dans l'action du controller
$this->loadComponent('OneTimer');
$time = $this->OneTimer->getTime();
```

> **Note**
>
> Gardez à l'esprit que le chargement d'un component à la volée n'appellera pas les callbacks manquants. Si vous souhaitez que les callbacks `beforeFilter` ou `startup()` soient appelées, vous devrez les appeler manuellement selon le moment où vous chargez votre component.

Utiliser les Components

Une fois que vous avez inclus quelques components dans votre controller, les utiliser est très simple. Chaque component que vous utilisez est enregistré comme propriété dans votre controller. Si vous avez chargé la *Cake\Controller\Component\FlashComponent* dans votre controller, vous pouvez y accéder comme ceci :

```
class PostsController extends AppController
{
    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('Flash');
    }

    public function delete()
    {
        if ($this->Post->delete($this->request->getData('Post.id')) {
            $this->Flash->success('Publication effacée.');

            return $this->redirect(['action' => 'index']);
        }
    }
```

> **Note**
>
> Puisque les Modèles et les Components sont tous deux ajoutés aux controllers en tant que propriétés, ils partagent le même "espace de noms". Assurez-vous de ne pas donner le même nom à un component et à un model.

Créer un Component

Supposons que notre application en ligne ait besoin de réaliser une opération mathématique complexe dans plusieurs sections différentes de l'application. Nous pourrions créer un component pour héberger cette logique partagée afin de l'utiliser dans plusieurs controllers différents.

La première étape consiste à créer un nouveau fichier et une classe pour le component. Créez le fichier dans **src/Controller/Component/MathComponent.php**. La structure de base pour le component ressemblerait à quelque chose comme cela :

```
namespace App\Controller\Component;

use Cake\Controller\Component;

class MathComponent extends Component
{
    public function doComplexOperation($amount1, $amount2)
    {
        return $amount1 + $amount2;
    }
}
```

> **Note**
>
> Tous les components doivent étendre `Cake\Controller\Component`. Ne pas le faire vous enverra une exception.

Inclure votre Component dans vos Controllers

Une fois notre component terminé, nous pouvons l'utiliser dans le controller de l'application en le chargeant durant la méthode `initialize()` du controller. Une fois chargé, le controller sera automatiquement pourvu d'un nouvel attribut nommé d'après le component, à travers lequel nous pouvons accéder à une instance de celui-ci :

```
// Dans un controller
// Rend le nouveau component disponible avec $this->Math
// ainsi que le component standard $this->Csrf
public function initialize(): void
{
    parent::initialize();
    $this->loadComponent('Math');
    $this->loadComponent('Csrf');
}
```

Quand vous incluez des Components dans un Controller, vous pouvez aussi déclarer un ensemble de paramètres qui seront passés au constructeur du Component. Ces paramètres peuvent alors être pris en charge par le Component :

```
// Dans votre controller.
public function initialize(): void
```

(suite sur la page suivante)

(suite de la page précédente)

```
{
    parent::initialize();
    $this->loadComponent('Math', [
        'precision' => 2,
        'randomGenerator' => 'srand'
    ]);
    $this->loadComponent('Csrf');
}
```

L'exemple ci-dessus passerait le tableau contenant precision et randomGenerator dans le paramètre `$config` de `MathComponent::initialize()`.

Utiliser d'autres Components dans votre Component

Parfois un de vos components a besoin d'utiliser un autre component. Dans ce cas, vous pouvez inclure d'autres components dans votre component exactement de la même manière que dans vos controllers - en utilisant la variable `$components` :

```
// src/Controller/Component/CustomComponent.php
namespace App\Controller\Component;

use Cake\Controller\Component;

class CustomComponent extends Component
{
    // L'autre component que votre component utilise
    public $components = ['Existing'];

    // Exécute une autre configuration additionnelle pour votre component.
    public function initialize(array $config): void
    {
        $this->Existing->foo();
    }

    public function bar()
    {
        // ...
   }
}

// src/Controller/Component/ExistingComponent.php
namespace App\Controller\Component;

use Cake\Controller\Component;

class ExistingComponent extends Component
{

    public function foo()
    {
        // ...
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
}
```

> **Note**
>
> Au contraire d'un component inclus dans un controller, aucun callback ne sera attrapé pour un component inclus dans un component.

Accéder au Controller du Component

À partir d'un component, vous pouvez accéder au controller courant via le registre :

```
$controller = $this->getController();
```

Callbacks des Components

Les components vous offrent aussi quelques callbacks durant leur cycle de vie qui vous permettent d'augmenter le cycle de la requête.

beforeFilter(*EventInterface $event*)

Est appelée avant la méthode du controller beforeFilter, mais *après* la méthode initialize() du controller.

startup(*EventInterface $event*)

Est appelée après la méthode du controller beforeFilter mais avant que le controller n'exécute l'action prévue.

beforeRender(*EventInterface $event*)

Est appelée après que le controller exécute la logique de l'action requêtée, mais avant le rendu de la vue et le layout du controller.

shutdown(*EventInterface $event*)

Est appelée avant que la sortie soit envoyée au navigateur.

beforeRedirect(*EventInterface $event*, *$url*, *Response $response*)

Est invoquée quand la méthode de redirection du controller est appelée, mais avant toute action qui suit. Si cette méthode retourne `false`, le controller ne continuera pas à rediriger la requête. Les paramètres $url et $response vous permettent d'inspecter et de modifier la localisation ou toutes autres entêtes dans la réponse.

Utiliser des Redirections dans les Events des Components

Pour rediriger à partir d'une méthode de rappel de composant, vous pouvez utiliser ce qui suit :

```
public function beforeFilter(EventInterface $event)
{
    $event->stopPropagation();

    return $this->getController()->redirect('/');
}
```

En arrêtant l'événement, vous faites savoir à CakePHP que vous ne voulez pas que les autres méthodes de callback de d'autres composants ne soient appelées ni que le controller ne gère l'action plus avant. À partir de la version 4.1.0, vous pouvez déclencher une `RedirectException` pour signaler une redirection :

```
use Cake\Http\Exception\RedirectException;
use Cake\Routing\Router;

public function beforeFilter(EventInterface $event)
{
    throw new RedirectException(Router::url('/'))
}
```

La levée d'une exception arrêtera tous les autres écouteurs d'événements et créera une nouvelle réponse qui ne conserve ou n'hérite d'aucun des en-têtes de la réponse actuelle. Lors de la levée d'une `RedirectException`, vous pouvez inclure des en-têtes supplémentaires :

```
throw new RedirectException(Router::url('/'), 302, [
    'Header-Key' => 'value',
]);
```

CHAPITRE 11

Views (Vues)

class Cake\View**View**

Les Views (Vues) sont le **V** dans MVC. Les vues sont chargées de générer la sortie spécifique requise par la requête. Souvent, cela est fait sous forme HTML, XML ou JSON, mais le streaming de fichiers et la création de PDFs que les utilisateurs peuvent télécharger sont aussi de la responsabilité de la couche View.

CakePHP a quelques classes de vue déjà construites pour gérer les scénarios de rendu les plus communs :

- Pour créer des services web XML ou JSON, vous pouvez utiliser *Vues JSON et XML*.
- Pour servir des fichiers protégés, ou générer des fichiers dynamiquement, vous pouvez utiliser *Envoyer des fichiers*.
- Pour créer plusieurs vues pour un thème, vous pouvez utiliser *Themes*.

La View App

`AppView` est la classe View par défaut de votre application. `AppView` étend elle-même la classe `Cake\View\View` de CakePHP et est définie dans **src/View/AppView.php** comme suit :

```
<?php
namespace App\View;

use Cake\View\View;

class AppView extends View
{
}
```

Vous pouvez utiliser `AppView` pour charger des helpers qui seront utilisés dans toutes les vues rendues de votre application. CakePHP fournit une méthode `initialize()` qui est invoquée à la fin du constructeur de la View pour ce type d'utilisation :

```php
<?php
namespace App\View;

use Cake\View\View;

class AppView extends View
{

    public function initialize(): void
    {
        // Toujours activer le helper MyUtils
        $this->loadHelper('MyUtils');
    }

}
```

Templates de Vues

La couche view de CakePHP c'est la façon dont vous parlez à vos utilisateurs. La plupart du temps, vos vues afficheront des documents HTML/XHTML pour les navigateurs, mais vous pourriez aussi avoir besoin de fournir des données AMF à un objet Flash, répondre à une application distante via SOAP ou produire un fichier CSV pour un utilisateur.

Les fichiers de template de CakePHP possèdent une extension **.php** (CakePHP Template) et utilisent la syntaxe alternative de PHP [118] pour les structures de contrôle et les sorties. Ces fichiers contiennent la logique nécessaire pour servir les données reçues d'un controller dans un format de présentation qui est lisible par votre public.

Sorties Alternatives

Utilisez `echo` ou `print` sur une variable dans votre template :

```php
<?php echo $variable; ?>
```

Avec le support de « Short Tag » :

```php
<?= $variable ?>
```

Structures de Contrôle Alternatives

Les structures de contrôle tel que `if`, `for`, `foreach`, `switch`, et `while` peuvent être écrites dans un format simplifié. Remarquez l'absence d'accolades. À la place, l'accolade de fin du `foreach` est remplacée par `endforeach`. Chacune des structures de contrôle listées ci-dessous a une syntaxe de fermeture similaire : `endif`, `endfor`, `endforeach`, et `endwhile`. Vous remarquerez aussi qu'à la place du `point-virgule` après chaque structure (à l'exception de la dernière), il y a un `double-point`.

Voici un example de `foreach` :

```php
<ul>
<?php foreach ($todo as $item): ?>
<li><?= $item ?></li>
<?php endforeach; ?>
</ul>
```

118. https://php.net/manual/fr/control-structures.alternative-syntax.php

Un autre exemple utilisant if/elseif/else. Remarquez les doubles points :

```
<?php if ($username === 'sally'): ?>
   <h3>Hi Sally</h3>
<?php elseif ($username === 'joe'): ?>
   <h3>Hi Joe</h3>
<?php else: ?>
   <h3>Hi unknown user</h3>
<?php endif; ?>
```

Si vous préférez utiliser un langage de template comme Twig [119], une sous-classe de View va faire le pont entre le langage du template et CakePHP.

Un fichier de template est stocké dans **templates/**, dans un sous-dossier portant le nom du controller qui utilise ce fichier. Il a un nom de fichier correspondant à son action. Par exemple, le fichier de vue pour l'action « view() » du controller Products devra normalement se trouver dans **templates/Products/view.php**.

La couche vue de CakePHP peut être constituée d'un certain nombre de parties différentes. Chaque partie a différents usages qui seront présentés dans ce chapitre :

- **templates** : Les templates sont la partie de la page qui est unique pour l'action lancée. Elles sont la substance de la réponse de votre application.
- **elements** : morceaux de code de view plus petits, réutilisables. Les elements sont habituellement rendus dans les vues.
- **layouts** : fichiers de template contenant le code de présentation qui se retrouve dans plusieurs interfaces de votre application. La plupart des vues sont rendues à l'intérieur d'un layout.
- **helpers** : ces classes encapsulent la logique de vue qui est requise à de nombreux endroits de la couche view. Parmi d'autres choses, les helpers de CakePHP peuvent vous aider à créer des formulaires, des fonctionnalités AJAX, à paginer les données du model ou à délivrer des flux RSS.
- **cells** : Ces classes fournissent des fonctionnalités de type controller en miniature pour créer des components avec une UI indépendante. Regardez la documentation *View Cells* pour plus d'informations.

Variables de Vue

Toute variable que vous définissez dans votre controller avec `set()` sera disponible à la fois dans la vue et dans le layout que votre action utilise. En plus, toute variable définie sera aussi disponible dans tout element. Si vous avez besoin de passer des variables supplémentaires de la vue vers le layout, vous pouvez soit appeler `set()` dans le template de vue, soit utiliser un *Utiliser les Blocks de Vues*.

Vous devriez vous rappeler de **toujours** échapper les données d'utilisateur avant de les afficher puisque CakePHP n'échappe automatiquement la sortie. Vous pouvez échapper le contenu d'utilisateur avec la fonction `h()` :

```
<?= h($user->bio); ?>
```

Définir les Variables de Vue

`Cake\View\View::set`(*string $var*, *mixed $value*)

Les vues ont une méthode `set()` qui fonctionne de la même façon que `set()` qui se trouve dans les objets Controller. Utiliser set() à partir de la vue va ajouter les variables au layout et aux elements qui seront rendus plus tard. Regardez *Définir les Variables de View* pour plus d'informations sur l'utilisation de `set()`.

Dans votre fichier de vue, vous pouvez faire :

```
$this->set('activeMenuButton', 'posts');
```

119. https://twig.symfony.com

Ensuite, dans votre layout, la variable `$activeMenuButton` sera disponible et contiendra la valeur "posts".

Vues étendues

Une vue étendue vous permet d'encapsuler une vue dans une autre. En combinant cela avec *view blocks*, cela vous donne une façon puissante pour garder vos vues *DRY*. Par exemple, votre application a une sidebar qui a besoin de changer selon la vue spécifique en train d'être rendue. En étendant un fichier de vue commun, vous pouvez éviter de répéter la balise commune pour votre sidebar, et seulement définir les parties qui changent :

```
<!-- templates/Common/view.php -->
<h1><?= $this->fetch('title') ?></h1>
<?= $this->fetch('content') ?>

<div class="actions">
    <h3>Related actions</h3>
    <ul>
    <?= $this->fetch('sidebar') ?>
    </ul>
</div>
```

Le fichier de vue ci-dessus peut être utilisé comme une vue parente. Il s'attend à ce que la vue l'étendant définisse des blocks `sidebar` et `title`. Le block `content` est un block spécial que CakePHP crée. Il contiendra tous les contenus non capturés de la vue étendue. En admettant que notre fichier de vue a une variable `$post` avec les données sur notre post. Notre vue pourrait ressembler à ceci :

```
<!-- templates/Posts/view.php -->
<?php
$this->extend('/Common/view');

$this->assign('title', $post->title);

$this->start('sidebar');
?>
<li>
<?php
echo $this->Html->link('edit', [
    'action' => 'edit',
    $post->id
]); ?>
</li>
<?php $this->end(); ?>

// The remaining content will be available as the 'content' block
// In the parent view.
<?= h($post->body) ?>
```

L'exemple ci-dessus vous montre comment vous pouvez étendre une vue, et remplir un ensemble de blocks. Tout contenu qui ne serait pas déjà dans un block défini, sera capturé et placé dans un block spécial appelé `content`. Quand une vue contient un appel vers un `extend()`, l'exécution continue jusqu'à la fin de la vue actuelle. Une fois terminé, la vue étendue va être générée. En appelant `extend()` plus d'une fois dans un fichier de vue, le dernier appel va outrepasser les précédents :

```
$this->extend('/Common/view');
$this->extend('/Common/index');
```

Le code précédent va définir **/Common/index.php** comme étant la vue parente de la vue actuelle.

Vous pouvez imbriquer les vues autant que vous le voulez et que cela vous est nécessaire. Chaque vue peut étendre une autre vue si vous le souhaitez. Chaque vue parente va récupérer le contenu de la vue précédente en tant que block `content`.

> **Note**
>
> Vous devriez éviter d'utiliser `content` comme nom de block dans votre application. CakePHP l'utilise pour définir le contenu non-capturé pour les vues étendues.

Vous pouvez récupérer la liste de tous blocks existants en utilisant la méthode `blocks()` :

```
$list = $this->blocks();
```

Utiliser les Blocks de Vues

Les blocks de vue fournissent une API flexible qui vous permet de définir des slots (emplacements), ou blocks, dans vos vues / layouts qui peuvent être définies ailleurs. Par exemple, les blocks pour implémenter des choses telles que les sidebars, ou des régions pour charger des ressources dans l'en-tête / pied de page du layout. Un block peut être défini de deux manières. Soit en tant que block capturant, soit en le déclarant explicitement. Les méthodes `start()`, `append()`, `prepend()`, `assign()`, `fetch()` et `end()` vous permettent de travailler avec les blocks capturant :

```
// Créer le block sidebar.
$this->start('sidebar');
echo $this->element('sidebar/recent_topics');
echo $this->element('sidebar/recent_comments');
$this->end();

// Le rattacher à la sidebar plus tard.
$this->start('sidebar');
echo $this->fetch('sidebar');
echo $this->element('sidebar/popular_topics');
$this->end();
```

Vous pouvez aussi ajouter dans un block en utilisant `append()` :

```
$this->append('sidebar');
echo $this->element('sidebar/popular_topics');
$this->end();

// Le même que ci-dessus.
$this->append('sidebar', $this->element('sidebar/popular_topics'));
```

Si vous devez nettoyer ou écraser un block, vous avez plusieurs alternatives. La méthode `reset()` va nettoyer ou écraser un block à n'importe quel moment. La méthode `assign()` avec une chaîne de caractères vide peut également être utilisée. :

```
// Nettoyer le contenu précédent du block de sidebar
$this->reset('sidebar');

// Assigner une chaine vide aura le même effet.
$this->assign('sidebar', '');
```

Assigner le contenu d'un block est souvent utile lorsque vous voulez convertir une variable de vue en un block. Par exemple, vous pourriez vouloir utiliser un block pour le titre de la page et parfois le définir depuis le controller :

```
// Dans une view ou un layout avant $this->fetch('title')
$this->assign('title', $title);
```

La méthode `prepend()` a été ajoutée pour ajouter du contenu avant un block existant :

```
// Ajoutez avant la sidebar
$this->prepend('sidebar', 'ce contenu va au-dessus de la sidebar');
```

Note

Vous devriez éviter d'utiliser `content` comme nom de bloc. Celui-ci est utilisé par CakePHP en interne pour étendre les vues, et le contenu des vues dans le layout.

Afficher les Blocks

Vous pouvez afficher les blocks en utilisant la méthode `fetch()`. Cette dernière va, de manière sécurisée, générer un block, en retournant "" si le block n'existe pas :

```
<?= $this->fetch('sidebar') ?>
```

Vous pouvez également utiliser fetch pour afficher du contenu, sous conditions, qui va entourer un block existant. Ceci est très utile dans les layouts, ou dans les vues étendues lorsque vous voulez, sous conditions, afficher des en-têtes ou autres balises :

```
// dans templates/layout/default.php
<?php if ($this->fetch('menu')): ?>
<div class="menu">
    <h3>Menu options</h3>
    <?= $this->fetch('menu') ?>
</div>
<?php endif; ?>
```

Vous pouvez aussi fournir une valeur par défaut pour un block qui ne devrait pas avoir de contenu. Cela vous permet d'ajouter du contenu placeholder, pour des déclarations vides. Vous pouvez fournir une valeur par défaut en utilisant le 2ème argument :

```
<div class="shopping-cart">
    <h3>Your Cart</h3>
    <?= $this->fetch('cart', 'Votre Caddie est vide') ?>
</div>
```

Utiliser des Blocks pour les Fichiers de Script et les CSS

`HtmlHelper` est lié aux blocks de vue, et ses méthodes `script()`, `css()`, et `meta()` mettent à jour chacun un block avec le même nom quand il est utilisé avec l'option `block = true` :

```
<?php
// Dans votre fichier de vue
$this->Html->script('carousel', ['block' => true]);
$this->Html->css('carousel', ['block' => true]);
```

(suite sur la page suivante)

(suite de la page précédente)

```
?>

// Dans votre fichier de layout.
<!DOCTYPE html>
<html lang="en">
    <head>
    <title><?= $this->fetch('title') ?></title>
    <?= $this->fetch('script') ?>
    <?= $this->fetch('css') ?>
    </head>
    // reste du layout à la suite
```

Le `HtmlHelper` vous permet aussi de contrôler vers quels blocks vont les scripts :

```
// dans votre vue
$this->Html->script('carousel', ['block' => 'scriptBottom']);

// dans votre layout
<?= $this->fetch('scriptBottom') ?>
```

Layouts

Un layout contient le code de présentation qui entoure une vue. Tout ce que vous voulez voir dans toutes vos vues devra être placé dans un layout.

Le fichier de layout par défaut de CakePHP est placé dans **templates/layout/default.php**. Si vous voulez changer entièrement le look de votre application, alors c'est le bon endroit pour commencer, parce que le code de vue de rendu du controller est placé à l'intérieur du layout par défaut quand la page est rendue.

Les autres fichiers de layout devront être placés dans **templates/layout**. Quand vous créez un layout, vous devez dire à CakePHP où placer la sortie pour vos vues. Pour ce faire, assurez-vous que votre layout contienne `$this->fetch('content')`. Voici un exemple de ce à quoi un layout pourrait ressembler :

```
<!DOCTYPE html>
<html lang="en">
<head>
<title><?= $this->fetch('title'); ?></title>
<link rel="shortcut icon" href="favicon.ico" type="image/x-icon">
<!-- Include external files and scripts here (See HTML helper for more info.) -->
<?php
echo $this->fetch('meta');
echo $this->fetch('css');
echo $this->fetch('script');
?>
</head>
<body>

<!-- Si vous voulez qu'un menu soit rendu pour toutes vos vues,
incluez le ici -->
<div id="header">
    <div id="menu">...</div>
</div>
```

(suite sur la page suivante)

(suite de la page précédente)

```
<!-- C'est ici que je veux voir mes vues être rendues -->
<?= $this->fetch('content') ?>

<!-- Ajoute un footer pour chaque page rendue -->
<div id="footer">...</div>

</body>
</html>
```

Les blocks `script`, `css` et `meta` contiennent tout contenu défini dans les vues en utilisant le helper HTML intégré. Il est utile pour inclure les fichiers JavaScript et les CSS à partir des vues.

> **Note**
>
> Quand vous utilisez `HtmlHelper::css()` ou `HtmlHelper::script()` dans les fichiers de template, spécifiez `'block' => true` pour placer la source html dans un block avec le même nom. (Regardez l'API pour plus de détails sur leur utilisation).

Le block `content` contient les contenus de la vue rendue.

Vous pouvez aussi définir le block `title` depuis l'intérieur d'un fichier de vue :

```
$this->assign('title', $titleContent);
```

Vous pouvez créer autant de layouts que vous souhaitez : placez les juste dans le répertoire **templates/layout**, et passez de l'un à l'autre depuis les actions de votre controller en utilisant la propriété `$layout` de votre controller ou de votre vue :

```
// À partir d'un controller
public function view()
{
    // Définir le layout
    $this->viewBuilder()->setLayout('admin');
}

// À partir d'un fichier de vue
$this->layout = 'loggedin';
```

Par exemple, si une section de mon site incorpore un plus petit espace pour une bannière publicitaire, je peux créer un nouveau layout avec le plus petit espace de publicité et le spécifier comme un layout pour toutes les actions du controller en utilisant quelque chose comme :

```
namespace App\Controller;

class UsersController extends AppController
{
    public function viewActive()
    {
        $this->set('title', 'View Active Users');
        $this->viewBuilder()->setLayout('default_small_ad');
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    public function viewImage()
    {
        $this->viewBuilder()->setLayout('image');

        // Output user image
    }
}
```

Outre le layout par défaut, le squelette officiel d'application CakePHP dispose également d'un layout "ajax". Le layout AJAX est pratique pour élaborer des réponses AJAX - c'est un layout vide (la plupart des appels ajax ne nécessitent qu'un peu de balise en retour, et pas une interface de rendu complète).

Le squelette d'application dispose également d'un layout par défaut pour aider à générer du RSS.

Utiliser les layouts à partir de plugins

Si vous souhaitez utiliser un layout qui existe dans un plugin, vous pouvez utiliser la *syntaxe de plugin*. Par exemple pour utiliser le layout de contact à partir du plugin Contacts :

```
namespace App\Controller;

class UsersController extends AppController
{
    public function view_active()
    {
        $this->viewBuilder()->layout('Contacts.contact');
    }
}
```

Elements

`Cake\View\View::element`(*string $elementPath*, *array $data*, *array $options = []*)

Beaucoup d'applications ont des petits blocks de code de présentation qui doivent être répliqués d'une page à une autre, parfois à des endroits différents dans le layout. CakePHP peut vous aider à répéter des parties de votre site web qui doivent être réutilisées. Ces parties réutilisables sont appelées des Elements. Les publicités, les boites d'aides, les contrôles de navigation, les menus supplémentaires, les formulaires de connexion et de sortie sont souvent intégrés dans CakePHP en elements. Un element est tout bêtement une mini-vue qui peut être inclue dans d'autres vues, dans les layouts, et même dans d'autres elements. Les elements peuvent être utilisés pour rendre une vue plus lisible, en plaçant le rendu d'éléments répétitifs dans ses propres fichiers. Ils peuvent aussi vous aider à réutiliser des fragments de contenu dans votre application.

Les elements se trouvent dans le dossier **templates/element/**, et ont une extension .php. Ils sont rendus en utilisant la méthode element de la vue :

```
echo $this->element('helpbox');
```

Passer des Variables à l'intérieur d'un Element

Vous pouvez passer des données dans un element grâce au deuxième argument :

```
echo $this->element('helpbox', [
    'helptext' => 'Oh, ce texte est très utile.'
]);
```

Dans le fichier element, toutes les variables passées sont disponibles comme des membres du paramètre du tableau (de la même manière que `Controller::set()` fonctionne dans le controller avec les fichiers de template). Dans l'exemple ci-dessus, le fichier **templates/element/helpbox.php** peut utiliser la variable `$helptext` :

```
// A l'intérieur de templates/element/helpbox.php
echo $helptext; //affiche 'Oh, ce texte est très utile.'
```

La méthode *`View::element()`* supporte aussi les options pour l'element. Les options supportées sont "cache" et "callbacks". Un exemple :

```
echo $this->element('helpbox', [
        'helptext' => "Ceci est passé à l'element comme $helptext",
        'foobar' => "Ceci est passé à l'element via $foobar",
    ],
    [
        // utilise la configuration de cache `long_view`
        'cache' => 'long_view"',
        // défini à true pour avoir before/afterRender appelé pour l'element
        'callbacks' => true
    ]
);
```

La mise en cache d'element est facilitée par la classe `Cache`. Vous pouvez configurer les elements devant être stockés dans toute configuration de Cache que vous avez défini. Cela vous donne une grande flexibilité pour choisir où et combien de temps les elements sont stockés. Pour mettre en cache les différentes versions du même element dans une application, fournissez une valeur unique de la clé cache en utilisant le format suivant :

```
$this->element('helpbox', [], [
        'cache' => ['config' => 'short', 'key' => 'unique value']
    ]
);
```

Si vous avez besoin de plus de logique dans votre element, comme des données dynamiques à partir d'une source de données, pensez à utiliser une View Cell plutôt qu'un element. Vous pouvez en savoir plus en consultant *les View Cells*.

Mise en cache des Elements

Vous pouvez tirer profit de la mise en cache de vue de CakePHP si vous fournissez un paramètre cache. Si défini à `true`, cela va mettre en cache l'element dans la configuration "default" de Cache. Sinon, vous pouvez définir la configuration de cache devant être utilisée. Regardez *La mise en cache* pour plus d'informations sur la façon de configurer `Cache`. Un exemple simple de mise en cache d'un element serait par exemple :

```
echo $this->element('helpbox', [], ['cache' => true]);
```

Si vous rendez le même element plus d'une fois dans une vue et que vous avez activé la mise en cache, assurez-vous de définir le paramètre "key" avec un nom différent à chaque fois. Cela évitera que chaque appel successif n'écrase le résultat de la mise en cache du précédent appel de `element()`. Par exemple :

```
echo $this->element(
    'helpbox',
    ['var' => $var],
    ['cache' => ['key' => 'first_use', 'config' => 'view_long']]
);

echo $this->element(
    'helpbox',
    ['var' => $differenVar],
    ['cache' => ['key' => 'second_use', 'config' => 'view_long']]
);
```

Ce qui est au-dessus va s'enquérir que les deux résultats d'element sont mis en cache séparément. Si vous voulez que tous les elements mis en cache utilisent la même configuration du cache, vous pouvez sauvegarder quelques répétitions, en configurant `View::$elementCache` dans la configuration de Cache que vous souhaitez utiliser. CakePHP va utiliser cette configuration, quand aucune n'est donnée.

Requêter les Elements à partir d'un Plugin

Si vous utilisez un plugin et souhaitez utiliser les elements à partir de l'intérieur d'un plugin, utilisez juste la *syntaxe de plugin* habituelle. Si la vue est rendue pour un controller/action d'un plugin, le nom du plugin va automatiquement être préfixé pour tous les elements utilisés, à moins qu'un autre nom de plugin ne soit présent. Si l'element n'existe pas dans le plugin, il ira voir dans le dossier principal APP :

```
echo $this->element('Contacts.helpbox');
```

Si votre vue fait partie d'un plugin, vous pouvez ne pas mettre le nom du plugin. Par exemple, si vous êtes dans le `ContactsController` du plugin Contacts :

```
echo $this->element('helpbox');
// et
echo $this->element('Contacts.helpbox');
```

Sont équivalents et résulteront à l'affichage du même element.

Pour les elements dans le sous-dossier d'un plugin (e.g., **plugins/Contacts/sidebar/helpbox.php**), utilisez ce qui suit :

```
echo $this->element('Contacts.sidebar/helpbox');
```

Préfix de Routing et Elements

Si vous avez configuré un préfix de routage, la résolution des chemins d'accès aux Elements peut chercher dans un chemin préfixé, comme les Layouts et les vues d'Action le font déjà. En partant du postulat que vous avez configuré le préfix « Admin » et que vous appelez :

```
echo $this->element('my_element');
```

L'element va d'abord être cherché dans **templates/Admin/Element/**. Si un tel fichier n'existe pas, il sera ensuite cherché dans le chemin par défaut.

Mettre en Cache des Sections de votre View

`Cake\View\View::cache`(*callable $block*, *array $options = []*)

Parfois, générer une section de l'affichage de votre view peut être coûteux à cause du rendu des *View Cells* ou du fait d'opérations de helper coûteuses. Pour que votre application s'exécute plus rapidement, CakePHP fournit un moyen de mettre en cache des sections de view :

```
// En supposant l'existence des variables locales
echo $this->cache(function () use ($user, $article) {
    echo $this->cell('UserProfile', [$user]);
    echo $this->cell('ArticleFull', [$article]);
}, ['key' => 'my_view_key']);
```

Par défaut, le contenu de la view ira dans la config de cache `View::$elementCache`, mais vous pouvez utiliser l'option `config` pour changer ceci.

Events de View

Tout comme le Controller, la View lance plusieurs events/callbacks (méthodes de rappel) que vous pouvez utiliser pour insérer de la logique durant tout le cycle de vie du processus de rendu :

Liste des Events

- `View.beforeRender`
- `View.beforeRenderFile`
- `View.afterRenderFile`
- `View.afterRender`
- `View.beforeLayout`
- `View.afterLayout`

Vous pouvez attacher les *listeners d'events* de votre application à ces events ou utiliser les *Callbacks de Helper*.

Créer vos propres Classes de View

Vous avez peut-être besoin de créer vos propres classes de vue pour activer des nouveaux types de données de vue, ou ajouter de la logique supplémentaire pour le rendu de vue personnalisée. Comme la plupart des components de CakePHP, les classes de vue ont quelques conventions :

- Les fichiers de classe de View doivent être mis dans **src/View**. Par exemple **src/View/PdfView.php**.
- Les classes de View doivent être suffixées avec `View`. Par exemple `PdfView`.
- Quand vous référencez les noms de classe de vue, vous devez omettre le suffixe `View`. Par exemple `$this->viewBuilder()->className('Pdf');`.

Vous voudrez aussi étendre `View` pour vous assurer que les choses fonctionnent correctement :

```
// Dans src/View/PdfView.php
namespace App\View;

use Cake\View\View;

class PdfView extends View
{
    public function render($view = null, $layout = null)
    {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        // logique personnalisée ici.
    }
}
```

Remplacer la méthode render vous laisse le contrôle total sur la façon dont votre contenu est rendu.

En savoir plus sur les vues

View Cells

View cells sont des mini-controllers qui peuvent invoquer de la logique de vue et afficher les templates. L'idée des cells est empruntée aux cells dans ruby [120], où elles remplissent un rôle et un sujet similaire.

Quand utiliser les Cells

Les Cells sont idéales pour la construction de components de page réutilisables qui nécessitent une interaction avec les models, la logique de view, et la logique de rendu. Un exemple simple serait un caddie dans un magasin en ligne, ou un menu de navigation selon des données dans un CMS.

Créer une Cell

Pour créer une cell, vous définissez une classe dans **src/View/Cell**, et un template dans **templates/cell/**. Dans cet exemple, nous ferons une cell pour afficher le nombre de messages dans la boite de messages de notification de l'utilisateur. D'abord, créons le fichier de classe. Son contenu devrait ressembler à ceci :

```
namespace App\View\Cell;

use Cake\View\Cell;

class InboxCell extends Cell
{

    public function display()
    {
    }

}
```

Sauvegardez ce fichier dans **src/View/Cell/InboxCell.php**. Comme vous pouvez le voir, comme pour les autres classes dans CakePHP, les Cells ont quelques conventions :

- Les Cells se trouvent dans le namespace `App\View\Cell`. Si vous faîtes une cell dans un plugin, le namespace sera `PluginName\View\Cell`.
- Les noms de classe doivent finir en Cell.
- Les classes doivent hériter de `Cake\View\Cell`.

Nous avons ajouté une méthode vide `display()` à notre cell, c'est la méthode conventionnelle par défaut pour le rendu de cell. Nous couvrirons la façon d'utiliser les autres méthodes plus tard dans la doc. Maintenant, créons le fichier **templates/cell/Inbox/display.php**. Ce sera le template pour notre nouvelle cell.

Vous pouvez générer ce bout de code rapidement en utilisant `bake` :

```
bin/cake bake cell Inbox
```

Générera le code que nous avons tapé.

120. https://github.com/trailblazer/cells

Implémenter la Cell

Supposons que nous travaillions sur une application qui permette aux utilisateurs d'envoyer des messages aux autres. Nous avons un model `Messages`, et nous voulons montrer le nombre de messages non lus sans avoir à polluer AppController. C'est un cas d'utilisation parfait pour une cell. Dans la classe, nous avons juste ajouté ce qui suit :

```
namespace App\View\Cell;

use Cake\View\Cell;

class InboxCell extends Cell
{

    public function display()
    {
        $unread = $this->fetchTable('Messages')->find('unread');
        $this->set('unread_count', $unread->count());
    }

}
```

Puisque les cells utilisent `LocatorAwareTrait` et `ViewVarsTrait`, elles se comportent un peu comme un controller. Nous pouvons utiliser les méthodes `fetchTable()` et `set()` un peu comme nous le ferions dans un controller. Dans notre fichier de template, ajoutons ce qui suit :

```
<!-- templates/cell/Inbox/display.php -->
<div class="notification-icon">
    Vous avez <?= $unread_count ?> messages non lus.
</div>
```

> **Note**
>
> Les templates des cells ont une portée isolée et ne partage pas la même instance de View que celle utilisée pour rendre le template et le layout de l'action du controller courant ou d'autres cells. Ils ne sont donc pas au courant de tous les appels aux helpers ou aux blocs définis dans template / layout de l'action et vice versa.

Charger les Cells

Les cells peuvent être chargées à partir des views en utilisant la méthode `cell()` et fonctionne de la même manière dans les deux contextes :

```
// Charge une cell d'une application
$cell = $this->cell('Inbox');

// Charge une cell d'un plugin
$cell = $this->cell('Messaging.Inbox');
```

Ce qui est au-dessus va charger la classe de cell nommée et exécuter la méthode `display()`. Vous pouvez exécuter d'autres méthodes en utilisant ce qui suit :

```
// Lance la méthode expanded() dans la cell Inbox
$cell = $this->cell('Inbox::expanded');
```

Si vous avez besoin que votre controller décide quelles cells doivent être chargées dans une requête, vous pouvez utiliser le `CellTrait` dans votre controller pour y activer la méthode `cell()` :

```
namespace App\Controller;

use App\Controller\AppController;
use Cake\View\CellTrait;

class DashboardsController extends AppController
{
    use CellTrait;

    // More code.
}
```

Passer des Arguments à une Cell

Vous voudrez souvent paramétrer les méthodes cell pour rendre les cells plus flexibles. En utilisant les deuxième et troisième arguments de `cell()`, vous pouvez passer des paramètres d'action, et des options supplémentaires à vos classes de cell, en tableau indexé :

```
$cell = $this->cell('Inbox::recent', ['-3 days']);
```

Ce qui est au-dessus correspondra à la signature de la fonction suivante :

```
public function recent($since)
{
}
```

Afficher une Cell

Une fois qu'une cell a été chargée et exécutée, vous voudrez probablement l'afficher. La façon la plus simple pour rendre une cell est de faire une echo :

```
<?= $cell ?>
```

Ceci va afficher le template correspondant à la version en minuscule et avec des underscores de notre nom d'action, par exemple **display.php**.

Puisque les cells utilisent `View` pour afficher les templates, vous pouvez charger les cells supplémentaires dans un template de cell si nécessaire.

> **Note**
>
> L'affichage d'une cell utilise la méthode magique PHP `__toString()` qui empêche PHP de montrer le nom du fichier et le numéro de la ligne pour toutes les erreurs fatales levées. Pour obtenir un message d'erreur qui a du sens, il est recommandé d'utiliser la méthode `Cell::render()`, par exemple `<?= $cell->render() ?>`.

Afficher un Template alternatif

Par convention, les cells affichent les templates qui correspondent à l'action qu'ils exécutent. Si vous avez besoin d'afficher un template de vue différent, vous pouvez spécifier le template à utiliser lors de l'affichage de la cell :

```
// Appel de render() explicitement
echo $this->cell('Inbox::recent', ['-3 days'])->render('messages');

// Définit le template avant de faire un echo de la cell.
$cell = $this->cell('Inbox'); ?>
$cell->viewBuilder()->setTemplate('messages');

echo $cell;
```

Mettre en Cache la Sortie de Cell

Quand vous affichez une cell, vous pouvez mettre en cache la sortie rendue si les contenus ne changent pas souvent ou pour aider à améliorer la performance de votre application. Vous pouvez définir l'option `cache` lors de la création d'une cell pour activer & configurer la mise en cache :

```
// Le Cache utilisant la config par défaut et une clé générée
$cell = $this->cell('Inbox', [], ['cache' => true]);

// Mise en cache avec une config de cache spécifique et une clé générée
$cell = $this->cell('Inbox', [], ['cache' => ['config' => 'cell_cache']]);

// Spécifie la clé et la config à utiliser.
$cell = $this->cell('Inbox', [], [
    'cache' => ['config' => 'cell_cache', 'key' => 'inbox_' . $user->id]
]);
```

Si une clé est générée, la version en underscore de la classe cell et le nom du template seront utilisés.

> **Note**
>
> Une nouvelle instance de `View` est utilisée pour retourner chaque cell et ces nouveaux objets ne partagent pas de contexte avec le template /layout principal. Chaque cell est auto-contenu et a seulement accès aux variables passés en arguments par l'appel de `View::cell()`.

Paginer des Données dans une Cell

Créer une cell qui qui rend des résultats paginés peut être fait en utilisant la classe `Paginator` de l'ORM. Voici un exemple de pagination des messages favoris d'un utilisateur :

```
namespace App\View\Cell;

use Cake\View\Cell;
use Cake\Datasource\Paginator;

class FavoritesCell extends Cell
{
    public function display($user)
    {
        // Création du paginator
        $paginator = new Paginator();

        // Pagination du model
```

(suite sur la page suivante)

(suite de la page précédente)

```
        $results = $paginator->paginate(
            $this->fetchTable('Messages'),
            $this->request->getQueryParams(),
            [
                // Utilisation d'un finder personnalisé avec paramètre
                'finder' => ['favorites' => [$user]],

                // Utilisation de paramètre de query 'scoped'.
                'scope' => 'favorites',
            ]
        );
        $this->set('favorites', $results);
    }
}
```

La cell ci-dessus va paginer le model `Messages` en utilisant les *paramètres de pagination "scopés"*.

Utiliser des Helpers dans une Cell

Les cells ont leur propre contexte et leur propre instance View, mais les helpers chargés dans `AppView::initialize()` restent chargés comme d'habitude.

Pour charger un Helper spécifique uniquement pour une Cell spécifique, procédez de la façon suivante :

```
namespace App\View\Cell;

use Cake\View\Cell;

class FavoritesCell extends Cell
{
    public function initialize(): void {
        $this->viewBuilder()->addHelper('MonHelperPersonnalise');
    }
}
```

Themes

Les themes dans CakePHP sont simplement des plugins qui se ne fournissent que des fichiers de template. Consultez la section *Créer Vos Propres Plugins*. Vous pouvez profiter des themes, ce qui permet le changement rapide du visuel et du ressenti de votre page. En plus des fichiers de template, ils peuvent fournir des helpers et des cells si votre theme le nécessite. Quand vous utilisez des cells et des helpers à partir de votre theme, vous devrez continuer à utiliser la *syntaxe de plugin*.

Pour utiliser les themes, définissez le nom du theme dans votre controller ou dans votre callback `beforeRender()` :

```
class ExamplesController extends AppController
{
    // Pour CakePHP avant 3.1
    public $theme = 'Modern';

    public function beforeRender(\Cake\Event\Event $event)
    {
        $this->viewBuilder()->setTheme('Modern');
```

(suite sur la page suivante)

(suite de la page précédente)

```
        // Pour les versions antérieures à CakePHP 3.5
        $this->viewBuilder()->theme('Modern');
    }
}
```

Les fichiers de template du theme doivent être dans un plugin avec le même nom. Par exemple, le theme ci-dessus se trouvera dans **plugins/Modern/src/Template**. Il est important de se rappeler que CakePHP s'attend à trouver des noms de plugin/theme en CamelCase. En plus de cela, la structure de dossier dans le dossier **plugins/Modern/src/Template** est exactement la même que **templates/**.

Par exemple, le fichier de vue pour une action edit d'un controller Posts se trouvera dans **plugins/Modern/templates/Posts/edit.php**. Les fichiers de layout se trouveront dans **plugins/Modern/templates/layout/**. Vous pouvez aussi fournir des templates personnalisés pour les plugins avec un theme. Si vous aviez un plugin s'appelant "Cms", qui contient un TagsController, le theme Modern pourrait fournir **plugins/Modern/templates/plugin/Cms/Tags/edit.php** pour remplacer le template edit dans le plugin.

Si un fichier de template ne peut pas être trouvé dans le theme, CakePHP va essayer de le trouver dans le dossier **templates/**. De cette façon, vous pouvez créer les fichiers de template principaux et simplement les surcharger au cas par cas dans votre dossier theme.

Assets du theme

Puisque les themes sont des plugins CakePHP standards, ils peuvent inclure tout asset nécessaire dans leur répertoire webroot. Cela permet de packager et distribuer les themes. En développement, les requêtes pour les assets de theme seront gérées par `Cake\Routing\Dispatcher`. Pour améliorer les performances pour les environnements de production, il est recommandé d'"*améliorer les performances de votre application.*.

Tous les helpers intégrés à CakePHP connaissent les themes et seront créés avec les bons chemins automatiquement. Comme les fichiers de template, si un fichier n'est pas dans le dossier du theme, il va chercher par défaut dans le dossier webroot principal :

```
//Quand un theme avec le nom de 'purple_cupcake'
$this->Html->css('main.css');

//Crée un chemin comme
/purple_cupcake/css/main.css

// et les liens vers
plugins/PurpleCupcake/webroot/css/main.css
```

Vues JSON et XML

Les views `JsonView` et `XmlView` vous permettent de créer des réponses JSON et XML, et sont intégrées à `Cake\Controller\Component\RequestHandlerComponent`.

En activant `RequestHandlerComponent` dans votre application, et en activant le support pour les extensions `json` et/ou `xml`, vous pouvez automatiquement vous appuyer sur les nouvelles classes de vue. `JsonView` et `XmlView` feront référence aux vues de données pour le reste de cette page.

Il y a deux façons de générer des vues de données. La première est en utilisant l'option `serialize`, et la seconde en créant des fichiers de template normaux.

Activation des Vues de Données dans votre Application

Avant que vous ne puissiez utiliser les classes de vue de données, vous devrez charger `Cake\Controller\Component\RequestHandlerComponent` dans votre controller :

```
public function initialize(): void
{
    ...
    $this->loadComponent('RequestHandler');
}
```

Ceci peut être fait dans votre *AppController* et va activer automatiquement la classe de vue en s'adaptant selon les types de contenu. Vous pouvez aussi configurer le component avec le paramètre `viewClassMap`, pour faire correspondre les types à vos classes personnalisées et/ou les faire correspondre à d'autres types de données.

Vous pouvez en option activer les extensions json et ou xml avec les *Routing des Extensions de Fichier*. Ceci va vous permettre d'accéder à `JSON`, `XML` ou tout autre format spécial de vue en utilisant une URL personnalisée finissant avec le nom du type de réponse en tant qu'extension de fichier comme par exemple `http://example.com/articles.json`.

Par défaut, quand vous n'activez pas les *Routing des Extensions de Fichier*, l'en-tête `Accept` de la requête est utilisé pour sélectionner le type de format qui doit être rendu à l'utilisateur. Un exemple de format `Accept` utilisé pour rendre les réponses `JSON` est `application/json`.

Utilisation des Vues de Données avec l'option Serialize

L'option `serialize` indique quelle(s) autre(s) variable(s) de vue devrai(en)t être sérialisée(s) quand on utilise la vue de données. Cela vous permet de sauter la définition des fichiers de template pour vos actions de controller si vous n'avez pas besoin de faire un formatage avant que vos données ne soient converties en json/xml.

Si vous avez besoin de faire tout type de formatage ou de manipulation de vos variables de vue avant la génération de la réponse, vous devrez utiliser les fichiers de template. La valeur de `serialize` peut être soit une chaîne de caractère, soit un tableau de variables de vue à sérialiser :

```
class ArticlesController extends AppController
{
    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('RequestHandler');
    }

    public function index()
    {
        // Défini les variables de vues qui doivent être sérialisées.
        $this->set('articles', $this->paginate());

        // Spécifie quelles variables de vues JsonView doit sérialiser.
        $this->viewBuilder()->setOption('serialize', 'articles');
    }
}
```

Vous pouvez aussi définir `serialize` en tableau de variables de vue à combiner :

```
class ArticlesController extends AppController
{
    public function initialize(): void
```

(suite sur la page suivante)

(suite de la page précédente)

```
    {
        parent::initialize();
        $this->loadComponent('RequestHandler');
    }

    public function index()
    {
        // Du code qui crée created $articles et $comments

        // Définit les variables de vues qui doivent être sérialisées.
        $this->set(compact('articles', 'comments'));

        // Spécifie les variables de vues JsonView à sérialiser.
        $this->viewBuilder()->setOption('serialize', ['articles', 'comments']);
    }
}
```

Définir `serialize` en tableau comporte le bénéfice supplémentaire d'ajouter automatiquement un élément de top-niveau `<response>` en utilisant *XmlView*. Si vous utilisez une valeur de chaîne de caractère pour `serialize` et XmlView, assurez-vous que vos variables de vue aient un élément unique de top-niveau. Sans un élément de top-niveau, le Xml ne pourra être généré.

Utilisation d'une Vue de Données avec les Fichiers de Template

Vous devrez utiliser les fichiers de template si vous avez besoin de faire des manipulations du contenu de votre vue avant de créer la sortie finale. Par exemple, si vous avez des articles, qui ont un champ contenant du HTML généré, vous aurez probablement envie d'omettre ceci à partir d'une réponse JSON. C'est une situation où un fichier de vue est utile :

```
// Code du controller
class ArticlesController extends AppController
{
    public function index()
    {
        $articles = $this->paginate('Articles');
        $this->set(compact('articles'));
    }
}

// Code de la vue - templates/Articles/json/index.php
foreach ($articles as &$article) {
    unset($article->generated_html);
}
echo json_encode(compact('articles'));
```

Vous pouvez faire des manipulations encore beaucoup plus complexes, comme utiliser les helpers pour formater. Les classes de vue ded onnées ne supportent pas les layouts. Elles supposent que le fichier de vue va afficher le contenu sérialisé.

Créer des Views XML

class XmlView

Par défaut quand on utilise `serialize`, XmlView va envelopper vos variables de vue sérialisées avec un nœud `<response>`. Vous pouvez définir un nom personnalisé pour ce nœud en utilisant l'option `rootNode`.

La classe XmlView intègre l'option `xmlOptions` qui vous permet de personnaliser les options utilisées pour générer le XML, par exemple `tags` au lieu d'"`attributes`.

Comme exemple d'utilisation de `XmlView`, on peut imaginer la génération d'un sitemap.xml [121]. Ce tye de document nécessite de changer `_rootNode` et de définir des attributs. Les attributs sont définis en utilisant le préfixe `@` :

```
use Cake\View\XmlView;

public function viewClasses(): array
{
    return [XmlView::class];
}

public function sitemap()
{
    $pages = $this->Pages->find()->all();
    $urls = [];
    foreach ($pages as $page) {
        $urls[] = [
            'loc' => Router::url(['controller' => 'Pages', 'action' => 'view', $page->
↪slug, '_full' => true]),
            'lastmod' => $page->modified->format('Y-m-d'),
            'changefreq' => 'daily',
            'priority' => '0.5'
        ];
    }

    // Définir un root node personnalisé dans le document généré.
    $this->viewBuilder()
        ->setOption('rootNode', 'urlset')
        ->setOption('serialize', ['@xmlns', 'url']);
    $this->set([
        // Définir un attribut sur le root node.
        '@xmlns' => 'http://www.sitemaps.org/schemas/sitemap/0.9',
        'url' => $urls
    ]);
}
```

Créer des Views JSON

class JsonView

La classe JsonView intègre la variable `_jsonOptions` qui vous permet de personnaliser le masque utilisé pour générer le JSON. Regardez la documentation json_encode [122] sur les valeurs valides de cette option.

Par exemple, pour serializer le rendu des erreurs de validation des entités de CakePHP de manière cohérente, vous pouvez le faire de la manière suivante :

121. https://www.sitemaps.org/protocol.html
122. https://php.net/json_encode

```
// Dans l'action de votre controller, quand une sauvegarde échoue
$this->set('errors', $articles->errors());
$this->viewBuilder()
    ->setOption('serialize', ['errors'])
    ->setOption('jsonOptions', JSON_FORCE_OBJECT);
```

Réponse JSONP

Quand vous utilisez `JsonView`, vous pouvez utiliser la variable de vue spéciale `_jsonp` pour retourner une réponse JSONP. La définir à `true` fait que la classe de vue vérifie si le paramètre de chaine de la requête nommée « callback » est défini et si c'est le cas, permet d'envelopper la réponse json dans le nom de la fonction fournie. Si vous voulez utiliser un nom personnalisé de paramètre de requête à la place de « callback », définissez `_jsonp` avec le nom requis à la place de `true`.

Exemple d'Utilisation

Alors que *RequestHandlerComponent* peut automatiquement définir la vue en fonction du content-type ou de l'extension de la requête, vous pouvez aussi gérer les mappings de vue dans votre controller :

```
// src/Controller/VideosController.php
namespace App\Controller;

use App\Controller\AppController;
use Cake\Http\Exception\NotFoundException;

class VideosController extends AppController
{
    public function export($format = '')
    {
        $format = strtolower($format);

        // Format pour le view mapping
        $formats = [
          'xml' => 'Xml',
          'json' => 'Json',
        ];

        // Erreur sur un type inconnu
        if (!isset($formats[$format])) {
            throw new NotFoundException(__('Unknown format.'));
        }

        // Définit le format de la Vue
        $this->viewBuilder()->className($formats[$format]);

        // Récupérer les données
        $videos = $this->Videos->find('latest');

        // Définir les Données de la Vue
        $this->set(compact('videos'));
        $this->viewBuilder()->setOption('serialize', ['videos']);

        // Définit le téléchargement forcé
```

(suite sur la page suivante)

(suite de la page précédente)

```
        return $this->response->withDownload('report-' . date('YmdHis') . '.' . $format);
    }
}
```

Helpers (Assistants)

Les Helpers (Assistants) sont des classes comme les components, pour la couche de présentation de votre application. Ils contiennent la logique de présentation qui est partagée entre plusieurs vues, elements ou layouts. Ce chapitre vous montrera comment créer vos propres helpers et soulignera les tâches basiques que les helpers du cœur de CakePHP peuvent vous aider à accomplir.

CakePHP dispose d'un certain nombre de helpers qui aident à la création des vues. Ils aident à la création de balises bien-formatées (y compris les formulaires), aident à la mise en forme du texte, les durées et les nombres, et peuvent même accélérer la fonctionnalité AJAX. Pour plus d'informations sur les helpers inclus dans CakePHP, regardez le chapitre pour chaque helper :

Breadcrumbs

class `Cake\View\Helper\`**`BreadcrumbsHelper`**(*View $view*, *array $config = []*)

BreadcrumbsHelper vous offre la possibilité de gérer la création et le rendu de vos *breadcrumbs* (fil d'Ariane) pour vos applications.

Créer un fil d'Ariane

Vous pouvez ajouter un élément à la liste en utilisant la méthode `add()`. Elle accepte trois arguments :

- `title` La chaîne affichée comme titre de l'élément.
- `url` Une chaîne ou un tableau de paramètres qui sera passé au *Url*
- `options` Un tableau d'attribut pour les templates `item` et `itemWithoutLink`. Référez-vous à la section sur *la définition d'attributs pour un élément* pour plus d'informations.

En plus de pouvoir ajouter un élément à la fin de la liste, vous pouvez effectuer diverses opérations :

```
// Ajoute à la fin de la liste
$this->Breadcrumbs->add(
    'Produits',
    ['controller' => 'products', 'action' => 'index']
);

// Ajoute plusieurs éléments à la fin de la liste
$this->Breadcrumbs->add([
    ['title' => 'Produits', 'url' => ['controller' => 'products', 'action' => 'index']],
    ['title' => 'Nom du produit', 'url' => ['controller' => 'products', 'action' => 'view
↪', 1234]]
]);

// Ajoute l'élément en premier dans la liste
$this->Breadcrumbs->prepend(
    'Produits',
    ['controller' => 'products', 'action' => 'index']
);

// Ajoute plusieurs éléments en premier dans la liste
$this->Breadcrumbs->prepend([
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ['title' => 'Produits', 'url' => ['controller' => 'products', 'action' => 'index']],
    ['title' => 'Nom du produit', 'url' => ['controller' => 'products', 'action' => 'view
↪', 1234]]
]);

// Insert l'élément à un index spécifique. Si l'index n'existe pas
// une exception sera levée.
$this->Breadcrumbs->insertAt(
    2,
    'Produits',
    ['controller' => 'products', 'action' => 'index']
);

// Insert l'élément avant un autre, basé sur le titre.
// Si l'élément ne peut pas être trouvé, une exception sera levée
$this->Breadcrumbs->insertBefore(
    'Un nom de produit', // le titre de l'élément devant lequel on veut faire l'insertion
    'Produits',
    ['controller' => 'products', 'action' => 'index']
);

// Insert l'élément après un autre, basé sur le titre.
// Si l'élément ne peut pas être trouvé, une exception sera levée
$this->Breadcrumbs->insertAfter(
    'Un nom de produit', // le titre de l'élément derrière lequel on veut faire l
↪'insertion
    'Produits',
    ['controller' => 'products', 'action' => 'index']
);
```

Utilisez ces méthodes vous donne la possibilité de contourner la façon dont CakePHP rend les vues. Puisque les templates et les layouts sont rendu de l'intérieur vers l'extérieur (entendez par là que les éléments inclus sont rendus avant les éléments qui les inclusent), cela vous permet de définir précisemment où vous voulez ajouter un élément.

Afficher le fil d'Ariane

Affichage simple

Après avoir ajouté un élément à la liste, vous pouvez facilement l'afficher avec la méthode `render()`. Cette méthode accepte deux tableaux comme arguments :

- `$attributes` : Un tableau d'attributs qui seront appliqués au template `wrapper`. Cela vous donne lapossibilité d'ajouter des attributs au tag HTML utilisé. Il accepte également la clé `templateVars` ce qui vous permet d'insérer des variables de template personnalisées dans le template.
- `$separator` : Un tableau d'attributs pour le template `separator`. Voici les propriétés disponibles :
  - `separator` La chaîne qui sera utilisée comme séparateur
  - `innerAttrs` Pour fournir des attributs dans le cas où votre séparateur est en deux éléments
  - `templateVars` Vous permet de définir des variables de templates personnalisées dans le template

  Toutes les autres propriétés seront converties en attributs HTML et remplaceront la clé `attrs` dans le template. Si vous fournissez un tableau vide (le défaut) pour cet argument, aucun séparateur ne sera affiché.

Voici un exemple d'affichage d'un fil d'Ariane :

```
echo $this->Breadcrumbs->render(
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ['class' => 'breadcrumbs-trail'],
    ['separator' => '<i class="fa fa-angle-right"></i>']
);
```

Personnaliser l'affichage

Personnaliser les templates

Le BreadcrumbsHelper utiliser le trait `StringTemplateTrait` en interne, ce qui vous permet de facilement personnaliser le rendu des différentes chaînes HTML qui composent votre fil d'Ariane. Quatre templates sont inclus. Voici leur déclaration par défaut :

```
[
    'wrapper' => '<ul{{attrs}}>{{content}}</ul>',
    'item' => '<li{{attrs}}><a href="{{url}}"{{innerAttrs}}>{{title}}</a></li>{
↪{separator}}',
    'itemWithoutLink' => '<li{{attrs}}><span{{innerAttrs}}>{{title}}</span></li>{
↪{separator}}',
    'separator' => '<li{{attrs}}><span{{innerAttrs}}>{{separator}}</span></li>'
]
```

Vous pouvez facilement personnaliser ces templates via la méthode `setTemplates()` du `StringTemplateTrait` :

```
$this->Breadcrumbs->setTemplates([
    'wrapper' => '<nav class="breadcrumbs"><ul{{attrs}}>{{content}}</ul></nav>',
]);
```

Puisque les templates supportent l'option `templateVars`, vous pouvez ajouter vos propres variables de templates :

```
$this->Breadcrumbs->setTemplates([
    'item' => '<li{{attrs}}>{{icon}}<a href="{{url}}"{{innerAttrs}}>{{title}}</a></li>{
↪{separator}}'
]);
```

Et pour définir le paramètre `{{icon}}`, vous n'avez qu'à la spécifier lorsque vous ajouter l'élément à la liste :

```
$this->Breadcrumbs->add(
    'Produits',
    ['controller' => 'products', 'action' => 'index'],
    [
        'templateVars' => [
            'icon' => '<i class="fa fa-money"></i>'
        ]
    ]
);
```

Defining Attributes for the Item

Si vous voulez déclarez des attributs HTML à l'élément et ses sous-éléments, vous pouvez utiliser la clé `innerAttrs` supportée par l'argument `$options`. Toutes les clés exceptées `innerAttrs` et `templateVars` seront affichés comme attributs HTML :

```
$this->Breadcrumbs->add(
    'Produits',
    ['controller' => 'products', 'action' => 'index'],
    [
        'class' => 'products-crumb',
        'data-foo' => 'bar',
        'innerAttrs' => [
            'class' => 'inner-products-crumb',
            'id' => 'the-products-crumb'
        ]
    ]
);

// En se basant sur le template par défaut, la chaîne suivante sera affichée:
<li class="products-crumb" data-foo="bar">
    <a href="/products/index" class="inner-products-crumb" id="the-products-crumb">
↪Produits</a>
</li>
```

Réinitialiser la Liste d'éléments

Vous pouvez réinitialiser la liste d'éléments à l'aide de la méthode `reset()`. Ceci est particulièrement utile quand vous souhaitez modifier les éléments et complètement réinitialiser la liste :

```
$crumbs = $this->Breadcrumbs->getCrumbs();
$crumbs = collection($crumbs)->map(function ($crumb) {
    $crumb['options']['class'] = 'breadcrumb-item';

    return $crumb;
})->toArray();

$this->Breadcrumbs->reset()->add($crumbs);
```

Flash

class Cake\View\Helper**FlashHelper**(*View $view*, *array $config = []*)

FlashHelper fournit une façon de rendre les messages flash qui sont définis dans `$_SESSION` par *FlashComponent*. *FlashComponent* et FlashHelper utilisent principalement des elements pour rendre les messages flash. Les elements flash se trouvent dans le répertoire **templates/element/flash**. Vous remarquerez que le template de l'App de CakePHP est livré avec trois éléments flash : **success.php**, **default.php** et **error.php**.

Rendre les Messages Flash

Pour rendre un message flash, vous pouvez simplement utiliser la méthode `render()` du FlashHelper :

```
<?= $this->Flash->render() ?>
```

Par défaut, CakePHP utilise une clé « flash » pour les messages flash dans une session. Mais si vous spécifiez une clé lors de la définition du message flash dans *FlashComponent*, vous pouvez spécifier la clé flash à rendre :

```
<?= $this->Flash->render('other') ?>
```

Vous pouvez aussi surcharger toutes les options qui sont définies dans FlashComponent :

```
// Dans votre Controller
$this->Flash->set('The user has been saved.', [
    'element' => 'success'
]);

// Dans votre View: Va utiliser great_success.php au lieu de succcess.php
<?= $this->Flash->render('flash', [
    'element' => 'great_success'
]);
```

Note

Quand vous construisez vos propres templates de messages flash, assurez- vous de correctement encoder les données utilisateurs. CakePHP n'échappera pas les paramètres passés aux templates des messages flash pour vous.

Pour plus d'informations sur le tableau d'options disponibles, consultez la section *FlashComponent*.

Préfixe de Routage et Messages Flash

Si vous avez configuré un préfixe de Routage, vous pouvez maintenant stocker vos elements de messages Flash dans **templates/{Prefix}/element/flash**. De cette manière, vous pouvez avoir des layouts de messages spécifiques en fonction des différentes parties de votre application : par exemple, avoir des layouts différents pour votre front-end et votre administration.

Les Messages Flash et les Themes

FlashHelper utilise des elements normaux pour afficher les messages et va donc correspondre à n'importe quel thème que vous avez éventuellement spécifié. Donc quand votre thème a un fichier **templates/element/flash/error.php**, il sera utilisé, comme avec tout Element et View.

Form

`class Cake\View\Helper\FormHelper(View $view, array $config = [])`

Le FormHelper prend en charge la plupart des opérations lourdes de la création de formulaire. Le FormHelper se concentre sur la possibilité de créer des formulaires rapidement, d'une manière qui permettra de rationaliser la validation, la re-population et la mise en page (layout). Le FormHelper est aussi flexible - il va faire à peu près tout pour vous en utilisant les conventions, ou vous pouvez utiliser des méthodes spécifiques pour ne prendre uniquement que ce dont vous avez besoin.

Création de Formulaire

`Cake\View\Helper\FormHelper::create(mixed $context = null, array $options = [])`

- `$context` - Le contexte pour lequel le formulaire est créé. Cela peut être une Entity de l'ORM, un retour (ResultSet) de l'ORM, un tableau de meta-données ou `false/null` (dans le cas où vous créez un formulaire qui ne serait lié à aucun Model).
- `$options` - Un tableau d'options et / ou d'attributs HTML.

La première méthode que vous aurez besoin d'utiliser pour tirer pleinement profit du FormHelper est `create()`. Cette méthode affichera une balise d'ouverture de formulaire.

Tous les paramètres sont optionnels. Si `create()` est appelée sans paramètre, CakePHP supposera que vous voulez créer un formulaire en rapport avec le controller courant, via l'URL actuelle. Par défaut, la méthode de soumission par

des formulaires est POST. Si vous appelez `create()` dans une vue pour `UsersController::add()`, vous verrez la sortie suivante dans la vue :

```
<form method="post" action="/users/add">
```

L'argument `$context` est utilisé comme "context" du formulaire. Il y a plusieurs contextes de formulaires intégrés et vous pouvez ajouter les vôtres, ce que nous allons voir dans la prochaine section. Ceux intégrés correspondent aux valeurs suivantes de `$context` :

- Une instance `Entity` ou un iterateur qui mappe vers EntityContext [123] ; ce contexte permet au FormHelper de fonctionner avec les retours de l'ORM intégré.
- Un tableau contenant la clé `schema`, qui mappe vers ArrayContext [124] ce qui vous permet de créer des structures simples de données pour construire des formulaires.
- `null` et `false` mappe vers NullContext [125] ; cette classe de contexte satisfait simplement l'interface requise par FormHelper. Ce contexte est utile si vous voulez construire un formulaire court qui ne nécessite pas de persistance via l'ORM.

Une fois qu'un formulaire a été créé avec un contexte, tous les inputs que vous créez vont utiliser le contexte actif. Dans le cas d'un formulaire basé sur l'ORM, FormHelper peut accéder aux données associées, aux erreurs de validation et aux metadata du schema. Vous pouvez fermer le contexte actif en utilisant la méthode `end()`, ou en appelant `create()` à nouveau. Pour créer un formulaire pour une entity, faites ce qui suit :

```
// Si vous êtes sur /articles/add
// $article devra être une entity Article vide.
echo $this->Form->create($article);
```

Affichera :

```
<form method="post" action="/articles/add">
```

Celui-ci va POSTer les données de formulaire à l'action `add()` de `ArticlesController`. Cependant, vous pouvez utiliser la même logique pour créer un formulaire d'édition. Le FormHelper utilise l'objet `Entity` pour détecter automatiquement s'il faut créer un formulaire d'ajout (*add*) ou un d'édition (*edit*). Si l'entity fournie n'est pas "nouvelle", le form va être créé comme un formulaire d'édition.

Par exemple, si nous naviguons vers **http ://example.org/articles/edit/5**, nous pourrions faire ce qui suit :

```
// src/Controller/ArticlesController.php:
public function edit($id = null)
{
    if (empty($id)) {
        throw new NotFoundException;
    }
    $article = $this->Articles->get($id);
    // La logique d'enregistrement ici
    $this->set('article', $article);
}

// View/Articles/edit.php:
// Puisque $article->isNew() est false, nous aurons un formulaire d'édition
<?= $this->Form->create($article) ?>
```

Affichera :

123. https://api.cakephp.org/4.x/class-Cake.View.Form.EntityContext.html
124. https://api.cakephp.org/4.x/class-Cake.View.Form.ArrayContext.html
125. https://api.cakephp.org/4.x/class-Cake.View.Form.NullContext.html

```
<form method="post" action="/articles/edit/5">
<input type="hidden" name="_method" value="PUT">
```

Note

Puisque c'est un formulaire d'édition, un champ input caché est généré pour surcharger la méthode HTTP par défaut.

Dans certains cas, l'ID de l'entité est automatiquement ajoutée à la fin de l'URL `action` du formulaire. Si vous voulez *éviter* qu'un ID soit ajouté à l'URL, vous pouvez passer une chaîne dans `$options['url']`, telle que `'/my-account'` ou `\Cake\Routing\Router::url(['controller' => 'Users', 'action' => 'myAccount'])`.

Options pour la Création de Formulaire

Le tableau `$options` est l'endroit où se passe l'essentiel de la configuration du formulaire. Ce tableau spécial peut contenir un certain nombre de paires clé-valeur différentes qui affectent la façon dont la balise form est générée. Voici les valeurs autorisées :

- `'type'` - Vous permet de choisir le type de formulaire à créer. Si vous ne fournissez pas de type, il sera automatiquement détecté en fonction du "context" du formulaire. Cette option peut prendre une des valeurs suivantes :
  - `'get'` - Définira la `method` du formulaire à GET.
  - `'file'` - Définira la `method` du formulaire à POST et le `'enctype'` à « multipart/form-data ».
  - `'post'` - Définira la `method` à POST.
  - `'put', 'delete', 'patch'` - Écrasera la méthode HTTP avec PUT, DELETE ou PATCH, respectivement, quand le formulaire sera soumis.
- `'method'` - Vous permet de définir explicitement la `method` du formulaire. Les valeurs autorisés sont les même que pour le paramètre ci-dessus.
- `'url'` - Permet de spécifier l'URL à laquelle le formulaire postera les données. Peut être une chaîne ou un tableau de paramètre d'URL.
- `'encoding'` - Permet de définir l'attribut `accept-charset` du formulaire. Par défaut, la valeur de `Configure::read('App.encoding')` sera utilisée.
- `'enctype'` - Vous permet de définir l'encodage du formulaire de manière explicite.
- `'templates'` - Les templates pour les éléments à utiliser pour ce formulaire. Tous les templates fournis écraseront les templates déjà chargés. Ce paramètre peut soit être un nom de fichier (sans extension) du dossier `/config` ou un tableau de templates.
- `'context'` - Options supplémentaires qui seront fournies à la classe de "context" liée au formulaire. (Par exemple, le "context" `EntityContext` accepte une option `table` qui permet de définir la classe Table sur laquelle le formulaire devra se baser).
- `'idPrefix'` - Préfixe à utiliser pour les attributs `id` des éléments du formulaire.
- `'templateVars'` - Vous permet de définir des variables de template pour le template `formStart`.
- `autoSetCustomValidity` - Défini à `true` pour utiliser des messages de validation personnalisés pour required et notBlank dans le message de validité HTML5 du contrôle. Par défaut `true`.

Astuce

Vous pouvez, en plus des options définies ci-dessus, définir dans l'argument `$options`, tous les attributs HTML que vous pourriez vouloir passer à l'élément `form` (des classes, des attributs `data`, etc.).

Récupérer les valeurs du formulaire depuis d'autres sources

Les sources de valeurs du FormHelper définissent d'où les éléments du formulaire reçoivent leurs valeurs.

Les sources supportées sont `context`, `data` et `query`. Vous pouvez utiliser une ou plusieurs de ces sources en définissant l'option `valueSources` ou en appelant `setValuesSource()`. Tous les éléments générés par `FormHelper` vont collecter leurs valeurs à partir de ces sources, dans l'ordre que vous aurez défini.

Par défaut, Formhelper récupère ses valeurs depuis les `data` ou le « context », c'est-à-dire qu'il va récupérer les données avec `$request->getData()` ou, si elles sont absentes, à partir des données du contexte actif, qui sont les données de l'entity dans le cas de `EntityContext`.

Cependant, si vous construisez un formulaire qui a besoin d'aller récupérer ses valeurs dans la query string, vous pouvez utiliser `valueSource()` pour définir où le `FormHelper` doit aller récupérer les valeurs de ses champs :

```
// Donner la priorité à la query string plutôt qu'au contexte
echo $this->Form->create($article, [
    'type' => 'get',
    'valueSources' => ['query', 'context']
]);

// Même effet:
echo $this->Form
    ->setValueSources(['query', 'context'])
    ->create($articles, ['type' => 'get']);
```

Lorsque les données reçues ont besoin d'être traitées par l'entity (c'est-à-dire les convertir, traiter une table ou computer des entités) et affichées après une ou plusieurs soumissions de formulaire pendant lesquelles les données de la requête sont conservées, vous aurez besoin de placer `context` en premier :

```
// Donner la priorité au contexte par rapport aux données de la requête:
echo $this->Form->create($article,
    'valueSources' => ['context', 'data']
]);
```

Les sources définies seront réinitialisées à leur valeur par défaut `['data', 'context']` quand `end()` sera appelée.

Changer la méthode HTTP pour un Formulaire

En utilisant l'option `type`, vous pouvez changer la méthode HTTP qu'un formulaire va utiliser :

```
echo $this->Form->create($article, ['type' => 'get']);
```

Affichera :

```
<form method="get" action="/articles/edit/5">
```

En spécifiant `file` à l'option `type`, cela changera la méthode de soumission à "post", et ajoutera un `enctype` « multipart/form-data » dans le tag du formulaire. Vous devez l'utiliser si vous avez des demandes de fichiers dans votre formulaire. L'absence de cet attribut `enctype` empêchera le fonctionnement de l'envoi de fichiers :

```
echo $this->Form->create($article, ['type' => 'file']);
```

Affichera :

```
<form enctype="multipart/form-data" method="post" action="/articles/add">
```

Quand vous utilisez `put`, `patch` ou `delete` dans l'option `type`, votre formulaire aura un fonctionnement équivalent à un formulaire de type "post", mais quand il sera envoyé, la méthode de requête HTTP sera respectivement réécrite avec "PUT", "PATCH" ou "DELETE". Cela permet à CakePHP d'émuler un support REST dans les navigateurs web.

Définir l'URL pour le Formulaire

Utiliser l'option `url` vous permet de diriger le formulaire vers une action spécifique dans votre controller courant ou dans toute votre application. Par exemple, si vous voulez diriger le formulaire vers une action `publish()` du controller courant, vous pouvez fournir le tableau `$options` comme suit :

```
echo $this->Form->create($article, ['url' => ['action' => 'publish']]);
```

Affichera :

```
<form method="post" action="/articles/publish">
```

Si l'action que vous désirez appeler avec le formulaire n'est pas dans le controller courant, vous pouvez spécifier une URL dans le formulaire. L'URL fournie peut être relative à votre application CakePHP :

```
echo $this->Form->create(null, [
    'url' => [
        'controller' => 'Articles',
        'action' => 'publish'
    ]
]);
```

Affichera :

```
<form method="post" action="/articles/publish">
```

ou pointer vers un domaine extérieur :

```
echo $this->Form->create(null, [
    'url' => 'https://www.google.com/search',
    'type' => 'get'
]);
```

Affichera :

```
<form method="get" action="https://www.google.com/search">
```

Utilisez `'url' => false` si vous ne souhaitez pas d'URL en tant qu'action de formulaire.

Utiliser des Validateurs Personnalisés

Les models vont souvent avoir des ensembles de validation multiples et vous voudrez que FormHelper marque les champs nécessaires basés sur les règles de validation spécifiques que l'action de votre controller est en train d'appliquer. Par exemple, votre table Users a des règles de validation spécifiques qui s'appliquent uniquement quand un compte est enregistré :

```
echo $this->Form->create($user, [
    'context' => ['validator' => 'register']
]);
```

L'exemple précédent va utiliser les règles de validation définies dans le validateur `register`, définies par `UsersTable::validationRegister()`, pour le `$user` et toutes les associations liées. Si vous créez un formulaire pour les entities associées, vous pouvez définir les règles de validation pour chaque association en utilisant un tableau :

```
echo $this->Form->create($user, [
    'context' => [
        'validator' => [
            'Users' => 'register',
            'Comments' => 'default'
        ]
    ]
]);
```

Ce qui est au-dessus va utiliser `register` pour l'utilisateur, et `default` pour les commentaires de l'utilisateur. FormHelper utilise les validateurs pour générer les attributs HTML5 *required*, les attributs ARIA appropriés, et définir les messages d'erreur avec la browser validator API [126] . Si vous voulez désactiver les messages de validation HTML5, utilisez :

```
$this->Form->setConfig('autoSetCustomValidity', false);
```

Cela ne désactivera pas les attributs``required``/`aria-required`.

Créer des Classes de Contexte

Alors que les classes de contexte intégrées essaient de couvrir les cas habituels que vous pouvez rencontrer, vous pouvez avoir besoin de construire une nouvelle classe de contexte si vous utilisez un ORM différent. Dans ces situations, vous devrez implémenter Cake\View\Form\ContextInterface [127] . Une fois que vous avez implémenté cette interface, vous pouvez connecter votre nouveau contexte dans le FormHelper. Le mieux est souvent de le faire dans un event listener `View.beforeRender`, ou dans une classe de vue de l'application :

```
$this->Form->addContextProvider('myprovider', function ($request, $data) {
    if ($data['entity'] instanceof MyOrmClass) {
        return new MyProvider($data);
    }
});
```

Les fonctions de fabrique de contexte sont l'endroit où vous pouvez ajouter la logique pour vérifier les options du formulaire pour le type d'entity approprié. Si une donnée d'entrée correspondante est trouvée, vous pouvez retourner un objet. Si n'y a pas de correspondance, retournez null.

Création d'éléments de Formulaire

`Cake\View\Helper\FormHelper::control(string $fieldName, array $options = [])`

- `$fieldName` - Nom du champ (attribut `name`) de l'élément sous la forme `'Modelname.fieldname'`.
- `$options` - Un tableau d'option qui peut inclure à la fois des *Options pour la méthode control()* et des options d'autres méthodes (que la méthode `control()` utilise en interne pour générer les différents éléments HTML) ainsi que attribut HTML valide.

La méthode `control()` vous permet de générer des inputs de formulaire complets. Ces inputs inclueront une div enveloppante, un label, un widget d'input, et une erreur de validation si besoin. En utilisant les metadonnées dans le contexte du formulaire, cette méthode va choisir un type d'input approprié pour chaque champ. En interne, `control()` utilise les autres méthodes de FormHelper.

126. https://developer.mozilla.org/en-US/docs/Learn/HTML/Forms/Form_validation#Customized_error_messages
127. https://api.cakephp.org/4.x/interface-Cake.View.Form.ContextInterface.html

Astuce

Veuillez notez que, même si les éléments générés par la méthode `control()` sont appelés des « inputs » sur cette page, techniquement parlant, la méthode `control()` peut générer non seulement n'importe quel type de balise `input` mais aussi tous les autres types d'éléments HTML de formulaire (`select`, `button`, `textarea`).

Par défaut, la méthode `control()` utilisera les templates de widget suivant :

```
'inputContainer' => '<div class="input {{type}}{{required}}">{{content}}</div>'
'input' => '<input type="{{type}}" name="{{name}}"{{attrs}}>'
```

En cas d'erreurs de validation, elle utilisera également :

```
'inputContainerError' => '<div class="input {{type}}{{required}} error">{{content}}{
↪{error}}</div>'
```

Le type d'élément créé, dans le cas où aucune autre option n'est fournie pour générer le type d'élément, est induit par l'introspection du Model et dépendra du datatype de la colonne en question :

Column Type
: Champ de formulaire résultant

string, uuid (char, varchar, etc.)
: text

boolean, tinyint(1)
: checkbox

decimal
: number

float
: number

integer
: number

text
: textarea

text, avec le nom de password, passwd, ou psword
: password

text, avec le nom de email
: email

text, avec le nom de tel, telephone, ou phone
: tel

date
: date

datetime, timestamp
: datetime-local

datetimefractional, timestampfractional
: datetime-local

time
: time

month
: month

year
: select avec des années

binary
file

Le paramètre `$options` vous permet de choisir un type d'input spécifique si vous avez besoin :

```
echo $this->Form->control('published', ['type' => 'checkbox']);
```

> **Astuce**
>
> Veuillez notez que, par défaut, générer un élément via la méthode `control()` générera systématiquement un `div` autour de l'élément généré. Cependant, générer le même élément mais avec la méthode spécifique du `FormHelper` (par exemple `$this->Form->checkbox('published');`) ne générera pas, dans la majorité des cas, un `div` autour de l'élément. En fonction de votre cas d'usage, utilisez l'une ou l'autre méthode.

Un nom de classe `required` sera ajouté à la `div` enveloppante si les règles de validation pour le champ du model indiquent qu'il est requis et ne peut pas être vide. Vous pouvez désactiver les `required` automatiques en utilisant l'option `required` :

```
echo $this->Form->control('title', ['required' => false]);
```

Pour empêcher la validation faite par le navigateur pour l'ensemble du formulaire, vous pouvez définir l'option `'formnovalidate' => true` pour le bouton input que vous générez en utilisant *`submit()`* ou définir `'novalidate' => true` dans les options pour *`create()`*.

Par exemple, supposons que votre model User intègre les champs pour un *username* (varchar), *password* (varchar), *approved* (datetime) et *quote* (text). Vous pouvez utiliser la méthode `control()` du FormHelper pour créer les bons inputs pour tous ces champs de formulaire :

```
echo $this->Form->create($user);
// Va générer un input type="text"
echo $this->Form->control('username');
// Va générer un input type="password"
echo $this->Form->control('password');
// En partant du principe que 'approved' est un "datetime" ou un "timestamp",
// va générer un input de type "datetime-local"
echo $this->Form->control('approved');
// Va générer un textarea
echo $this->Form->control('quote');

echo $this->Form->button('Ajouter');
echo $this->Form->end();
```

Un exemple plus complet montrant quelques options pour le champ de date :

```
echo $this->Form->control('birth_date', [
    'label' => 'Date de naissance',
    'min' => date('Y') - 70,
    'max' => date('Y') - 18,
 ]);
```

Outre les *Options pour la méthode control()* vues ci-dessus, vous pouvez spécifier n'importe quelle option acceptée par la méthode spécifique au widget choisi (ou déduit par CakePHP) et n'importe quel attribut HTML (par exemple `onfocus`).

Si vous voulez un `select` utilisant une relation *belongsTo* ou *hasOne*, vous pouvez ajouter ceci dans votre controller Users (en supposant que l'User *belongsTo* Group) :

```
$this->set('groups', $this->Users->Groups->find('list')->all());
```

Après cela, ajoutez les lignes suivantes à votre template de vue du formulaire :

```
echo $this->Form->control('group_id', ['options' => $groups]);
```

Pour créer un `select` pour l'association *belongsToMany* Groups, vous pouvez ajouter ce qui suit dans votre UsersController :

```
$this->set('groups', $this->Users->Groups->find('list')->all());
```

Ensuite, ajouter les lignes suivantes à votre template de vue :

```
echo $this->Form->control('groups._ids', ['options' => $groups]);
```

Si votre nom de model est composé de deux mots ou plus (ex. « UserGroup »), quand vous passez les données en utilisant `set()` vous devrez nommer vos données dans un format CamelCase [128] (les Majuscules séparent les mots) et au pluriel comme ceci :

```
$this->set('userGroups', $this->UserGroups->find('list'));
```

> **Note**
>
> N'utilisez pas `FormHelper::control()` pour générer les boutons submit. Utilisez plutôt *`submit()`*.

Conventions de Nommage des Champs

Lors de la création de widgets, vous devez nommer vos champs d'après leur attribut correspondant dans l'entity du formulaire. Par exemple, si vous créez un formulaire pour un `$article`, vous créez des champs nommés d'après les propriétés. Par exemple `title`, `body` et `published`.

Vous pouvez créer des inputs pour les models associés, ou pour des models arbitraires en le passant dans `association.fieldname` en premier paramètre :

```
echo $this->Form->control('association.fieldname');
```

Tout point dans vos noms de champs sera converti en données de requête imbriquées. Par exemple, si vous créez un champ avec un nom `0.comments.body` vous aurez un nom d'attribut qui sera `0[comments][body]`. Cette convention coorespond à celle de l'ORM. Plus de détails pour tous les types d'associations se trouvent dans la section *Créer des Inputs pour les Données Associées*.

Lors de la création d'inputs de type datetime, FormHelper va ajouter un suffixe au champ. Vous pouvez remarquer des champs supplémentaires nommés `year`, `month`, `day`, `hour`, `minute`, ou `meridian` qui ont été ajoutés. Ces champs seront automatiquement convertis en objets `DateTime` quand les entities seront traitées.

Options pour la méthode control()

`FormHelper::control()` supporte un nombre important d'options via son paramètre `$options`. En plus de ses propres options, `control()` accepte des options pour les champs input générés (devinés ou choisis, comme les `checkbox` ou les `textarea`), ou encore les attributs HTML. Ce qui suit va couvrir les options spécifiques de `FormHelper::control()`.

128. https://fr.wikipedia.org/wiki/Camel_case#Variations_et_synonymes

— `$options['type']` - Une chaîne qui précise le type de widget à générer. En plus des types de champs vus dans *Création d'éléments de Formulaire*, vous pouvez aussi créer input de type `file`, `password` et tous les types supportés par HTML5. En spécifiant vous-même le type de l'élément à générer, vous écraserez le type automatique deviné par l'introspection du Model. Le défaut est `null` :

```
echo $this->Form->control('field', ['type' => 'file']);
echo $this->Form->control('email', ['type' => 'email']);
```

Affichera :

```
<div class="input file">
    <label for="field">Field</label>
    <input type="file" name="field" value="" id="field">
</div>
<div class="input email">
    <label for="email">Email</label>
    <input type="email" name="email" value="" id="email">
</div>
```

— `$options['label']` Soit une chaîne qui sera utilisée comme valeur pour l'élément HTML `<label>`, soit un tableau *d'options pour le label*. Vous pouvez placer sous cette clé le texte que vous voudriez voir affiché dans le label qui accompagne habituellement les éléments HTML `input`. Le défaut est `null`.

Par exemple :

```
echo $this->Form->control('name', [
    'label' => 'The User Alias'
]);
```

Affiche :

```
<div class="input">
    <label for="name">The User Alias</label>
    <input name="name" type="text" value="" id="name">
</div>
```

Vous pouvez définir cette clé à `false` pour désactiver l'affichage de l'élément `<label>` Par exemple :

```
echo $this->Form->control('name', ['label' => false]);
```

Affiche :

```
<div class="input">
    <input name="name" type="text" value="" id="name">
</div>
```

Si le label est désactivé et qu'un attribut `placeholder` est fourni, l'input généré aura un `aria-label` défini.

Définissez l'option `label` comme un tableau pour fournir des options supplémentaires pour l'élément `label`. Si vous faites ainsi, vous pouvez utiliser une clé `text` dans le tableau pour personnaliser le texte du label : : Par exemple :

```
echo $this->Form->control('name', [
    'label' => [
        'class' => 'thingy',
        'text' => 'The User Alias'
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ]
]);
```

Affiche :

```
<div class="input">
    <label for="name" class="thingy">The User Alias</label>
    <input name="name" type="text" value="" id="name">
</div>
```

— `$options['options']` - Vous pouvez passer à cette option un tableau contenant les choix pour les éléments comme les `radio` et les `select`, qui ont besoin d'un tableau d'items en argument. Reportez-vous à *Créer des Boutons Radio* et *Créer des Select* pour plus de détails. Le défaut est `null`.

— `$options['error']` Utiliser cette clé vous permettra de transformer les messages par défaut du model et de les utiliser, par exemple, pour définir des messages i18n. Pour désactiver le rendu des messages d'erreurs définissez la clé `error` à `false` :

```
echo $this->Form->control('name', ['error' => false]);
```

Pour surcharger les messages d'erreurs du model utilisez un tableau avec les clés respectant les messages d'erreurs de validation originaux :

```
$this->Form->control('name', [
    'error' => ['Not long enough' => __('This is not long enough')]
]);
```

Comme vu précédemment, vous pouvez définir le message d'erreur pour chaque règle de validation présente dans vos models. De plus, vous pouvez fournir des messages i18n pour vos formulaires.

— `$options['nestedInput']` - À utiliser avec les inputs `checkbox` et `radio`. Cette option permet de contrôler si les éléments `input` doivent être générés à l'intérieur ou à l'extérieur de l'élément `label`. Quand `control()` génère une checkbox ou un bouton radio, vous pouvez définir l'option à `false` pour forcer la génération de l'élément `input` en dehors du `label`.

D'autre part, vous pouvez également la définir à `true` pour n'importe quel type d'élément pour forcer la génération de l'élément `input` dans le `label`. Si vous changez l'option pour les boutons radio, vous aurez également besoin de modifier le template par défaut *"radioWrapper"*. Selon le `type` d'élément à générer, la valeur par défaut sera `true` ou `false`.

— `$options['templates']` - Les templates à utiliser pour cet `input`. N'importe quel template spécifié via cette option surchargera les templates déjà chargés. Cette option accepte soit un nom de fichier (sans extension) provenant de `/config` qui contient les templates à charger, soit un tableau définissant les templates à utiliser.

— `$options['labelOptions']` - Définissez l'option à `false` pour désactiver les `label` autour des `nestedWidgets` ou bien définissez un tableau d'attributs à appliquer à l'élément `label`.

Générer des Types d'Inputs Spécifiques

En plus de la méthode générique `control()`, le `FormHelper` a des méthodes spécifiques pour générer différents types d'inputs. Ceci peut être utilisé pour générer juste un extrait de code input, et combiné avec d'autres méthodes comme *`label()`* et *`error()`* pour générer des layouts (mise en page) complètement personnalisés.

Options Communes à Tous les Inputs

Parmi les différentes méthodes d'input, beaucoup supportent un jeu d'options communes qui, selon la méthode de formulaire utilisée, doivent être insérées soit sous la clé `$options` soit sous la clé `$attributes` du tableau en argument. Toutes ces options sont aussi supportées par `control()`. Pour réduire les répétitions, les options communes partagées par toutes les méthodes input sont :

— `id` Définir cette clé pour forcer la valeur du DOM id pour cet input. Cela remplacera l'"`idPrefix` qui pourrait être fixé.
— `default` Utilisé pour définir une valeur par défaut au champ input. La valeur est utilisée si les données passées au formulaire ne contiennent pas de valeur pour le champ (ou si aucune donnée n'est transmise). Si aucune valeur par défaut n'est définie, c'est la valeur par défaut de la colonne qui sera utilisée.

Exemple d'utilisation :

```
echo $this->Form->text('ingredient', ['default' => 'Sucre']);
```

Exemple avec un champ `select` (la taille « Medium » sera sélectionnée par défaut) :

```
$sizes = ['s' => 'Small', 'm' => 'Medium', 'l' => 'Large'];
echo $this->Form->select('size', $sizes, ['default' => 'm']);
```

> **Note**
>
> Vous ne pouvez pas utiliser `default` pour sélectionner une checkbox - vous devez plutôt définir cette valeur dans `$this->request->getData()` dans votre controller, ou définir l'option `checked` de l'input à `true`.
>
> Attention à l'utilisation de `false` pour assigner une valeur par défaut. Une valeur `false` est utilisée pour désactiver/exclure les options d'un champ, ainsi `'default' => false` ne définirait aucune valeur. À la place, utilisez `'default' => 0`.

— `value` Utilisée pour définir une valeur spécifique pour le champ d'input. Ceci va surcharger toute valeur qui aurait pu être injectée à partir du contexte, comme Form, Entity or `request->getData()` etc.

> **Note**
>
> Si vous souhaitez définir un champ pour qu'il ne rende pas sa valeur récupérée à partir du contexte ou de valuesSource, vous devrez définir `value` à `''` (au lieu de le définir à `null`).

En plus des options ci-dessus, vous pouvez y mélanger n'importe quel attribut HTML que vous souhaitez utiliser. Tout nom d'option non spécifiquement prévu par CakePHP sera traité comme un attribut HTML, et appliqué à l'élément HTML input généré.

Créer des Éléments Input

Les autres méthodes disponibles dans le FormHelper permettent la création d'éléments spécifiques de formulaire. La plupart de ces méthodes utilisent également un paramètre spécial `$options` ou `$attributes`. Toutefois, dans ce cas, ce paramètre est utilisé en priorité pour spécifier les attributs des balises HTML (comme la valeur ou le DOM id d'un élément du formulaire).

Créer des Inputs Text

`Cake\View\Helper\FormHelper::text`(*string $name*, *array $options*)

— `$name` - Le nom du champ (attribut `name`) sous la forme `'Modelname.fieldname'`.
— `$options` - Un tableau optionnel pouvant contenir n'importe quelles *options générales* ainsi que n'importe quels attributs HTML valides.

Va créer un `input` de type `text` :

```
echo $this->Form->text('username', ['class' => 'users']);
```

Affichera :

```
<input name="username" type="text" class="users">
```

Créer des Inputs Password

`Cake\View\Helper\FormHelper::password(string $fieldName, array $options)`

- `$name` - Le nom du champ (attribut `name`) sous la forme `'Modelname.fieldname'`.
- `$options` - Un tableau optionnel pouvant contenir n'importe quelles *options générales* ainsi que n'importe quels attributs HTML valides.

Création d'un input simple de type `password` :

```
echo $this->Form->password('password');
```

Affichera :

```
<input name="password" value="" type="password">
```

Créer des Inputs Cachés

`Cake\View\Helper\FormHelper::hidden(string $fieldName, array $options)`

- `$name` - Le nom du champ (attribut `name`) sous la forme `'Modelname.fieldname'`.
- `$options` - Un tableau optionnel pouvant contenir n'importe quelles *options générales* ainsi que n'importe quels attributs HTML valides.

Créera un input de type `hidden`. Exemple :

```
echo $this->Form->hidden('id');
```

Affichera :

```
<input name="id" type="hidden">
```

Créer des Textareas

`Cake\View\Helper\FormHelper::textarea(string $fieldName, array $options)`

- `$name` - Le nom du champ (attribut `name`) sous la forme `'Modelname.fieldname'`.
- `$options` - Un tableau optionnel pouvant contenir n'importe quelles *options générales*, des options spécifiques aux textareas (cf. ci-dessous) ou encore n'importe quels attributs HTML valides.

Crée un champ `textarea` (zone de texte). Le template utilisé par défaut est :

```
'textarea' => '<textarea name="{{name}}"{{attrs}}>{{value}}</textarea>'
```

Par exemple :

```
echo $this->Form->textarea('notes');
```

Affichera :

```
<textarea name="notes"></textarea>
```

Si le formulaire est un formulaire d'édition (c'est-à-dire si le tableau `$this->request->getData()` contient des informations précédemment sauvegardées pour l'entity `User`), la valeur correspondant au champ `notes` sera automatiquement ajoutée au HTML généré. Exemple :

```
<textarea name="notes" id="notes">
    Ce Texte est fait pour être édité.
</textarea>
```

Options pour Textarea

En plus des *options générales*, `textarea()` supporte quelques autres options spécifiques :

- `'escape'` - Permet de définir si le contenu du `textarea` doit être échappé ou non. Le défaut est `true`.

  Par exemple :

  ```
  echo $this->Form->textarea('notes', ['escape' => false]);
  // OU....
  echo $this->Form->control('notes', ['type' => 'textarea', 'escape' => false]);
  ```

- `'rows', 'cols'` - Ces deux clés permettent de définir les attributs HTML du même nom et qui désignent respectivement le nombre de lignes et de colonnes : : ;

  echo $this->Form->textarea("comment", ["rows" => "5", "cols" => "5"]) ;

  Affichera :

  ```
  <textarea name="textarea" cols="5" rows="5">
  </textarea>
  ```

Créer des Select, des Checkbox et des Boutons Radio

Ces éléments ont certains points communs et des options communes, c'est pourquoi ils sont regroupés dans cette section.

Les Options pour Select, Checkbox et Boutons Radio

Vous trouverez ci-dessous les options partagées par `select()`, `checkbox()` et `radio()` (les options spécifiques à une seule méthode sont décrites dans les sections dédiées à ces méthodes).

- `value` - Permet de définir ou sélectionner la valeur de l'élément ciblé :
  - Pour les checkboxes, cela définit l'attribut HTML `value` assigné à l'input à la valeur que vous définissez.
  - Pour les boutons radio ou les select, cela définit quel élément sera sélectionné quand le formulaire sera rendu (dans ce cas, `'value'` doit avoir une valeur valide, correspondant à un élément qui existe). Elle peut aussi être utilisée avec n'importe quel élément basé sur un select comme `date()`, `time()`, `dateTime()` :

    ```
    echo $this->Form->time('close_time', [
        'value' => '13:30:00'
    ]);
    ```

    > **Note**
    >
    > La clé `value` pour les `date()` et `dateTime()` peut aussi être un timestamp UNIX ou un objet DateTime.

    Pour un input `select` où vous définissez l'attribut `multiple` à true, vous pouvez utiliser un tableau des valeurs que vous voulez sélectionner par défaut :

```
// Les tags <options> avec valeurs 1 et 3 seront sélectionnés par défaut
echo $this->Form->select(
    'rooms',
    [1, 2, 3, 4, 5],
    [
        'multiple' => true,
        'value' => [1, 3]
    ]
);
```

— `empty` - S'applique à `radio()` et `select()`. Le défaut est `false`.
  — Quand elle est passée à `radio()` et définie à `true`, cela crée un élément `input` supplémentaire qui sera affiché avant le premier bouton radio, avec une valeur de `''` et un `label` qui vaudra `'empty'`. Si vous voulez un autre texte pour le label définissez la chaîne que vous voulez plutôt que `true`.
  — Quand elle est passée à la méthode `select`, cela crée un élément `option` vide avec une valeur vide dans la liste des choix. Si à la place d'une valeur vide vous souhaitez afficher un texte, passez une chaîne dans l'option :

```
echo $this->Form->select(
    'field',
    [1, 2, 3, 4, 5],
    ['empty' => '(choisissez)']
);
```

Affiche :

```
<select name="field">
    <option value="">(choisissez)</option>
    <option value="0">1</option>
    <option value="1">2</option>
    <option value="2">3</option>
    <option value="3">4</option>
    <option value="4">5</option>
</select>
```

— `hiddenField` Pour les checkboxes et les boutons radios, par défaut, un input caché est créé près de l'élément. Ainsi, la clé dans `$this->request->getData()` existera même sans valeur spécifiée. Pour les checkboxes, sa valeur vaudra `0`; pour les boutons radio, elle sera `''`.

Exemple d'un rendu par défaut :

```
<input type="hidden" name="published" value="0">
<input type="checkbox" name="published" value="1">
```

Ceci peut être désactivé en définissant l'option `hiddenField` à `false` :

```
echo $this->Form->checkbox('published', ['hiddenField' => false]);
```

Retournera :

```
<input type="checkbox" name="published" value="1">
```

Si vous voulez créer de multiples blocs d'entrées regroupés ensemble dans un formulaire, vous devriez définir ce paramètre à `false` sur tous les inputs, excepté le premier. Si l'input caché est à plusieurs endroits dans la page, c'est seulement le dernier groupe d'inputs qui sera sauvegardé.

Dans cet exemple , seules les couleurs tertiaires seront passées, et les couleurs primaires seront écrasées :

```
<h2>Couleurs primaires</h2>
<input type="hidden" name="color" value="0">
<label for="color-red">
    <input type="checkbox" name="color[]" value="5" id="color-red">
    Rouge
</label>

<label for="color-blue">
    <input type="checkbox" name="color[]" value="5" id="color-blue">
    Bleu
</label>

<label for="color-yellow">
    <input type="checkbox" name="color[]" value="5" id="color-yellow">
    Jaune
</label>

<h2>Couleurs tertiaires</h2>
<input type="hidden" name="color" value="0">
<label for="color-green">
    <input type="checkbox" name="color[]" value="5" id="color-green">
    Vert
</label>
<label for="color-purple">
    <input type="checkbox" name="color[]" value="5" id="color-purple">
    Magenta
</label>
<label for="color-orange">
    <input type="checkbox" name="color[]" value="5" id="color-orange">
    Orange
</label>
```

Désactiver l'option `'hiddenField'` dans le second groupe d'input empêcherait au contraire ce comportement.

Vous pouvez définir une autre valeur pour le champ caché, autre que 0, comme "N" :

```
echo $this->Form->checkbox('published', [
    'value' => 'Y',
    'hiddenField' => 'N',
]);
```

Utiliser des Collections pour construire des options

Il est possible d'utiliser la classe Collection pour construire votre tableau d'options. Cette approche est idéale si vous avez déjà une collection d'entities et que vous voulez vous en servir pour construire un élément select.

Vous pouvez utiliser la méthode `combine` pour construire un tableau d'options basique :

```
$options = $examples->combine('id', 'name');
```

Il est aussi possible d'ajouter d'autres attributs en étendant le tableau. Ce qui suit va créer un attribut data sur l'élément option, en utilisant la méthode de collections `map`

```
$options = $examples->map(function ($value, $key) {
    return [
        'value' => $value->id,
        'text' => $value->name,
        'data-created' => $value->created
    ];
});
```

Créer des Checkboxes

`Cake\View\Helper\FormHelper::checkbox(string $fieldName, array $options)`

- `$name` - Le nom du champ (attribut `name`) sous la forme `'Modelname.fieldname'`.
- `$options` - Un tableau optionnel pouvant contenir n'importe quelles *des options générales*, des options de la section *Les Options pour Select, Checkbox et Boutons Radio*, des options spécifiques aux checkboxes (ci-dessous) ou encore n'importe quels attributs HTML valides.

Créer un élément `checkbox`. Le template de widget utilisé est le suivant :

```
'checkbox' => '<input type="checkbox" name="{{name}}" value="{{value}}"{{attrs}}>'
```

Options spécifiques pour les Checkboxes

- `'checked'` - Booléen utilisé pour indiquer si cette checkbox est cochée ou non. Par défaut à `false`.
- `'disabled'` - Crée une checkbox désactivée (non éditable).

Cette méthode génère également un input de type `hidden` pour forcer l'existence de la donnée dans le tableau de POST.

Exemple

```
echo $this->Form->checkbox('done');
```

Affichera :

```
<input type="hidden" name="done" value="0">
<input type="checkbox" name="done" value="1">
```

Il est possible de modifier la valeur du checkbox en utilisant le tableau `$options` :

```
echo $this->Form->checkbox('done', ['value' => 555]);
```

Affichera :

```
<input type="hidden" name="done" value="0">
<input type="checkbox" name="done" value="555">
```

Si vous ne voulez pas que le FormHelper génère un input `hidden`, vous pouvez passer l'option `hiddenField` à `false` :

```
echo $this->Form->checkbox('done', ['hiddenField' => false]);
```

Affichera :

```
<input type="checkbox" name="done" value="1">
```

Créer des Boutons Radio

`Cake\View\Helper\FormHelper::radio(string $fieldName, array $options, array $attributes)`

- `$name` - Le nom du champ (attribut `name`) sous la forme `'Modelname.fieldname'`.
- `$options` - Un tableau optionnel contenant au minimum les labels pour les boutons radio. Ce tableau peut également contenir les `value` et des attributs HTML. Si ce tableau n'est pas fourni, la méthode générera seulement l'input `hidden` (si `'hiddenField'` vaut `true`) ou pas d'élément du tout (si `'hiddenField'` vaut `false`).
- `$attributes` - Un tableau optionnel pouvant contenir n'importe quelles *des options générales*, des options de la section *Les Options pour Select, Checkbox et Boutons Radio*, des options spécifiques aux boutons radio (ci-dessous) ou encore n'importe quels attributs HTML valides.

Crée un jeu de boutons radios. Les templates de widget utilisés par défaut seront :

```
'radio' => '<input type="radio" name="{{name}}" value="{{value}}"{{attrs}}>'
'radioWrapper' => '{{label}}'
```

Attributs spécifiques aux boutons radio

- `label` - booléen pour indiquer si oui ou non les labels pour les widgets doivent être affichés, ou un tableau d'attributs à appliquer aux labels. Dans le cas où un attribut `class` est défini, `selected` sera ajouté à l'attribut `class` du bouton sélectionné. Défaut à `true`.
- `hiddenField` - booléen pour indiquer si vous voulez que les résultats de radio() incluent un input caché avec une valeur de `''`. C'est utile pour créer des ensembles de boutons radio qui ne sont pas continus. Défaut à `true`.
- `disabled` - Définir à `true` ou `disabled` pour désactiver tous les boutons radio. Défaut à `false`.

Vous devez fournir le texte des label pour les boutons radio via l'argument `$options`.

Par exemple :

```
$this->Form->radio('genre', ['Masculin','Féminin','Neutre']);
```

Affichera :

```
<input name="gender" value="" type="hidden">
<label for="gender-0">
    <input name="gender" value="0" id="gender-0" type="radio">
    Masculine
</label>
<label for="gender-1">
    <input name="gender" value="1" id="gender-1" type="radio">
    Feminine
</label>
<label for="gender-2">
    <input name="gender" value="2" id="gender-2" type="radio">
    Neuter
</label>
```

Généralement, `$options` contient de simples paires clé => valeur. Cependant, si vous avez besoin de mettre des attributs personnalisés sur vos boutons radio, vous pouvez utiliser le format étendu.

Par exemple :

```
echo $this->Form->radio(
    'favorite_color',
    [
        ['value' => 'r', 'text' => 'Red', 'style' => 'color:red;'],
        ['value' => 'u', 'text' => 'Blue', 'style' => 'color:blue;'],
```

(suite sur la page suivante)

(suite de la page précédente)

```
        ['value' => 'g', 'text' => 'Green', 'style' => 'color:green;'],
    ]
);
```

Affichera :

```
<input type="hidden" name="favorite_color" value="">
<label for="favorite-color-r">
    <input type="radio" name="favorite_color" value="r" style="color:red;" id="favorite-
↪color-r">
    Red
</label>
<label for="favorite-color-u">
    <input type="radio" name="favorite_color" value="u" style="color:blue;" id="favorite-
↪color-u">
    Blue
</label>
<label for="favorite-color-g">
    <input type="radio" name="favorite_color" value="g" style="color:green;" id=
↪"favorite-color-g">
    Green
</label>
```

Vous pouvez tout aussi bien définir des attributs supplémentaires pour un label particulier :

```
echo $this->Form->radio(
    'couleur_preferee',
    [
        ['value' => 'r', 'text' => 'Rouge', 'label' => ['class' => 'rouge']],
        ['value' => 'u', 'text' => 'Bleu', 'label' => ['class' => 'bleu']],
    ]
);
```

Afichera :

```
<input type="hidden" name="favorite_color" value="">
<label for="couleur-preferee-r" class="rouge">
    <input type="radio" name="couleur_preferee" value="r" id="couleur-preferee-r">
    Rouge
</label>
<label for="couleur-preferee-u" class="bleu">
    <input type="radio" name="couleur_preferee" value="u" id="couleur-preferee-u">
    Bleu
</label>
```

Si la clé `label` est utilisée sur une option, les attributs dans `$attributes['label']` seront ignorés.

Créer des Select

`Cake\View\Helper\FormHelper::select`(*string $fieldName*, *array $options*, *array $attributes*)

- `$name` - Le nom du champ (attribut `name`) sous la forme `'Modelname.fieldname'`.
- `$options` - Un tableau optionnel contenant la liste des éléments pour le select. Si ce tableau n'est pas fourni, la méthode généra seulement un élément `select` vide, sans élément `option`.

— `$attributes` - Un tableau optionnel pouvant contenir n'importe quelles *des options générales*, des options de la section *Les Options pour Select, Checkbox et Boutons Radio*, des options spécifiques aux select (ci-dessous) ou encore n'importe quels attributs HTML valides.

Crée un élément `select`, rempli avec les éléments contenus dans `$options`. Si l'option `$attributes['value']` est fournie, alors les éléments `option` ayant cette(ces) valeur(s) seront affichés comme sélectionné(s) quand le select sera rendu.

Par défaut, `select` utilise ces templates de widget :

```
'select' => '<select name="{{name}}"{{attrs}}>{{content}}</select>'
'option' => '<option value="{{value}}"{{attrs}}>{{text}}</option>'
```

Il pourra également utiliser les templates suivants :

```
'optgroup' => '<optgroup label="{{label}}"{{attrs}}>{{content}}</optgroup>'
'selectMultiple' => '<select name="{{name}}[]" multiple="multiple"{{attrs}}>{{content}}</
↪select>'
```

Attributs pour les Select

— `'multiple'` - Si cette option est définie à `true`, le select sera multiple (plusieurs valeurs pourront être sélectionnées). Si elle est définie à `checkbox`, à la place d'un select multiple, vous aurez des checkbox. Défaut à `null`.
— `'escape'` - Booleén. Si `true`, le contenu des éléments `option` sera échappé (les caractères spéciaux seront convertis en entités HTML). Défaut à `true`.
— `'val'` - Permet de pré-sélectionner la valeur du select.
— `'disabled'` - Contrôle l'attribut `disabled`. Si l'option est définie à `true`, l'ensemble du select sera `disabled`. Si définie sous forme de tableau, seuls les éléments `option` dont la valeur est dans le tableau seront désactivés.

L'argument `$options` vous permet de définir manuellement le contenu des éléments `option` du `select`.

Par exemple

```
echo $this->Form->select('field', [1, 2, 3, 4, 5]);
```

Affichera :

```
<select name="field">
    <option value="0">1</option>
    <option value="1">2</option>
    <option value="2">3</option>
    <option value="3">4</option>
    <option value="4">5</option>
</select>
```

La tableau `$options` peut aussi être fourni sous forme de paires clé => valeur.

Par exemple

```
echo $this->Form->select('field', [
    'Value 1' => 'Label 1',
    'Value 2' => 'Label 2',
    'Value 3' => 'Label 3'
]);
```

Affichera :

```
<select name="field">
    <option value="Value 1">Label 1</option>
    <option value="Value 2">Label 2</option>
    <option value="Value 3">Label 3</option>
</select>
```

Si vous souhaitez générer un `select` avec des `optgroups`, passez les données sous forme de tableau multidimensionnel. Cela marche également avec les checkbox et les boutons radio, mais à la place d'éléments `optgroup`, vos éléments seront entourés d'un `fieldset`.

Par exemple :

```
$options = [
    'Group 1' => [
        'Value 1' => 'Label 1',
        'Value 2' => 'Label 2'
    ],
    'Group 2' => [
        'Value 3' => 'Label 3'
    ]
];
echo $this->Form->select('field', $options);
```

Affichera :

```
<select name="field">
    <optgroup label="Group 1">
        <option value="Value 1">Label 1</option>
        <option value="Value 2">Label 2</option>
    </optgroup>
    <optgroup label="Group 2">
        <option value="Value 3">Label 3</option>
    </optgroup>
</select>
```

Pour ajouter des attributs HTML aux éléments `option` :

```
$options = [
    ['text' => 'Description 1', 'value' => 'value 1', 'attr_name' => 'attr_value 1'],
    ['text' => 'Description 2', 'value' => 'value 2', 'attr_name' => 'attr_value 2'],
    ['text' => 'Description 3', 'value' => 'value 3', 'other_attr_name' => 'other_attr_
↪value'],
];
echo $this->Form->select('field', $options);
```

Affichera :

```
<select name="field">
    <option value="value 1" attr_name="attr_value 1">Description 1</option>
    <option value="value 2" attr_name="attr_value 2">Description 2</option>
    <option value="value 3" other_attr_name="other_attr_value">Description 3</option>
</select>
```

Contrôle des Select via Attributes

En utilisant des options spéciales dans l'argument `$attributes`, vous pouvez contrôler certains comportement de la méthode `select()`.

— `'empty'` - Définissez cette option à `true` pour ajouter une option vide en première position de la liste de vos `option`. Défaut à `false`.

Par exemple :

```
$options = ['M' => 'Masculin', 'F' => 'Féminin'];
echo $this->Form->select('sexe', $options, ['empty' => true]);
```

Affichera :

```
<select name="sexe">
    <option value=""></option>
    <option value="M">Masculin</option>
    <option value="F">Féminin</option>
</select>
```

— `'escape'` - La méthode `select()` examine cet attribut qui contient un booléen et détermine si le contenu des `option` sera échappé (les caractères spéciaux seront convertis en entités HTML).

Par exemple

```
// Ceci empêchera l'échappement du contenu de chaque élément option
$options = ['M' => 'Masculin', 'F' => 'Féminin'];
echo $this->Form->select('sexe', $options, ['escape' => false]);
```

— `'multiple'` - Si définie à `true`, cette option autorisera les sélections multiples dans le``select``.

Par exemple

```
echo $this->Form->select('field', $options, ['multiple' => true]);
```

Vous pouvez également définir `'multiple'` à `'checkbox'` pour afficher une liste de checkbox à la place :

```
$options = [
    'Value 1' => 'Label 1',
    'Value 2' => 'Label 2'
];
echo $this->Form->select('field', $options, [
    'multiple' => 'checkbox'
]);
```

Affichera :

```
<input name="field" value="" type="hidden">
<div class="checkbox">
    <label for="field-1">
        <input name="field[]" value="Value 1" id="field-1" type="checkbox">
        Label 1
    </label>
</div>
<div class="checkbox">
    <label for="field-2">
        <input name="field[]" value="Value 2" id="field-2" type="checkbox">
        Label 2
    </label>
```

(suite sur la page suivante)

(suite de la page précédente)

```
</div>
```

— `'disabled'` - Cette option sert à désactiver une partie ou tous les éléments `option`. Pour désactiver tous les éléments, passez `'disabled'` à `true`. Pour désactiver seulement certains éléments, définissez un tableau avec les clés des éléments que vous voulez désactiver.

Par exemple

```
$options = [
    'M' => 'Masculin',
    'F' => 'Féminin',
    'N' => 'Neutre'
];
echo $this->Form->select('genre', $options, [
    'disabled' => ['M', 'N']
]);
```

Affichera :

```
<select name="genre">
    <option value="M" disabled="disabled">Masculin</option>
    <option value="F">Féminin</option>
    <option value="N" disabled="disabled">Neutre</option>
</select>
```

Cette option fonctionne également quand `'multiple'` est définie à `'checkbox'` :

```
$options = [
    'Value 1' => 'Label 1',
    'Value 2' => 'Label 2'
];
echo $this->Form->select('field', $options, [
    'multiple' => 'checkbox',
    'disabled' => ['Value 1']
]);
```

Affichera :

```
<input name="field" value="" type="hidden">
<div class="checkbox">
    <label for="field-1">
        <input name="field[]" disabled="disabled" value="Value 1" type="checkbox">
        Label 1
    </label>
</div>
<div class="checkbox">
    <label for="field-2">
        <input name="field[]" value="Value 2" id="field-2" type="checkbox">
        Label 2
    </label>
</div>
```

Créer des Inputs File

`Cake\View\Helper\FormHelper::file(string $fieldName, array $options)`

- `$name` - Le nom du champ (attribut `name`) sous la forme `'Modelname.fieldname'`.
- `$options` - Un tableau optionnel pouvant contenir n'importe quelles *options générales* ainsi que n'importe quels attributs HTML valides.

Permet de créer un input de type `file` dans votre formulaire, pour faire de l'upload de fichier. Le template de widget utilisé sera :

```
'file' => '<input type="file" name="{{name}}"{{attrs}}>'
```

Vous devez vous assurer que le `enctype` du formulaire est défini a `multipart/form-data`. Pour cela, commencez par appeler la méthode `create` de votre formulaire via une des deux méthodes ci-dessous :

```
echo $this->Form->create($document, ['enctype' => 'multipart/form-data']);
// OU
echo $this->Form->create($document, ['type' => 'file']);
```

Ensuite ajoutez l'une des deux lignes dans votre formulaire :

```
echo $this->Form->control('submittedfile', [
    'type' => 'file'
]);

// OU
echo $this->Form->file('submittedfile');
```

> **Note**
>
> En raison des limitations du code HTML lui même, il n'est pas possible de définir des valeurs par défaut dans les champs input de type "file". À chaque affichage du formulaire, la valeur sera vide.

Pour empêcher le `submittedfile` d'être écrasé par un contenu vide, enlevez-le de `$_accessible`. Au choix, vous pouvez aussi retirer sa clé depuis la méthode `beforeMarshal` :

```
public function beforeMarshal(\Cake\Event\EventInterface $event, \ArrayObject $data, \
↪ArrayObject $options)
{
   if ($data['submittedfile'] === '') {
      unset($data['submittedfile']);
   }
}
```

Lors de la soumission du formulaire, vous pouvez accéder aux inputs de type file par le biais des objets `UploadedFileInterface` présents dans la requête. Pour déplacer les fichiers uploadés vers un emplacement permanent, vous pouvez utiliser :

```
$fileobject = $this->request->getData('submittedfile');
$destination = UPLOAD_DIRECTORY . $fileobject->getClientFilename();

// S'il existe un fichier du même nom, il sera écrasé.
$fileobject->moveTo($destination);
```

> **Note**
>
> Quand vous utilisez `$this->Form->file()`, pensez à bien définir le type d'envodage du formulaire en définissant l'option type à "file" dans `$this->Form->create()`.

-.. _create-datetime-controls :

Créer des éléments de formulaire pour les dates et heures

`Cake\View\Helper\FormHelper::dateTime`(*string $fieldName*, *array $options = []*)

- `$fieldName` - Une chaîne qui sera utilisée comme préfixe pour l'attribut `name` des `select`.
- `$options` - Un tableau optionnel pouvant contenir n'importe quelles *options générales* ainsi que n'importe quels attributs HTML valides.

Générera une balise input de type « datetime-local ».

Par exemple

```
<?= $this->form->dateTime('inscription') ?>
```

Affichera :

```
<input type="datetime-local" name="inscription" />
```

Le valeur de l'input peut être n'importe quel datetime valide ou une instance `DateTime`.

Par exemple

```
<?= $this->form->dateTime('inscription', ['value' => new DateTime()]) ?>
```

Affichera :

```
<input type="datetime-local" name="inscription" value="2019-02-08T18:20:10" />
```

Créer des Éléments Date

`Cake\View\Helper\FormHelper::date`(*string $fieldName*, *array $options = []*)

- `$fieldName` - Une chaîne qui sera utilisée comme préfixe pour l'attribut `name` des `select`.
- `$options` - Un tableau optionnel pouvant contenir n'importe quelles *options générales* ainsi que n'importe quels attributs HTML valides.

Générera une balise input de type « date ».

Par exemple

```
<?= $this->form->date('inscription') ?>
```

Affichera :

```
<input type="date" name="inscription" />
```

Créer des Éléments Time

`Cake\View\Helper\FormHelper::time(string $fieldName, array $options = [])`

- `$fieldName` - Une chaîne qui sera utilisée comme préfixe pour l'attribut `name` des `select`.
- `$options` - Un tableau optionnel pouvant contenir n'importe quelles *options générales* ainsi que n'importe quels attributs HTML valides.

Générera une balise input de type « time ».

Par exemple

```
echo $this->Form->time('redemarrage');
```

Affichera :

```
<input type="time" name="redemarrage" />
```

Créer des Éléments Mois

`Cake\View\Helper\FormHelper::month(string $fieldName, array $attributes)`

- `$fieldName` - Une chaîne qui sera utilisée comme préfixe pour l'attribut `name` des `select`.
- `$options` - Un tableau optionnel pouvant contenir n'importe quelles *options générales* ainsi que n'importe quels attributs HTML valides.

Générera une balise input de type « month ».

Par exemple

```
echo $this->Form->month('mob');
```

Affichera :

```
<input type="month" name="mob">
```

Créer des Éléments Année

`Cake\View\Helper\FormHelper::year(string $fieldName, array $options = [])`

- `$fieldName` - Une chaîne qui sera utilisée comme préfixe pour l'attribut `name` des `select`.
- `$options` - Un tableau optionnel pouvant contenir n'importe quelles *options générales* ainsi que n'importe quels attributs HTML valides. Les autres options valides sont :
  - `min` : La plus petite valeur pouvant être utilisée dans l'élément de saisie de l'année.
  - `max` : La plus grande valeur pouvant être utilisée dans l'élément de saisie de l'année.
  - `order` : L'ordre des années dans l'élément de saisie. Les valeurs possibles sont `'asc'` et `'desc'`. Par défaut `'desc'`.

Crée un élément `select` qui contiendra une option par année pour les années situées entre `min` et `max` si ces options sont fournies, ou pour les années entre -5 et +5 par rapport à l'année en cours. En complément, des attributs HTML peuvent être passés dans `$options`. Si `$options['empty']` est passé à `false`, le `select` n'aura pas d'élément vide en début de liste.

Par exemple pour créer un élément qui propose les années entre 2000 et l'année en cours, vous utiliserez le code suivant :

```
echo $this->Form->year('achat', [
    'min' => 2000,
    'max' => date('Y')
]);
```

Si nous sommes en 2009, nous obtiendrons :

```
<select name="achat">
    <option value=""></option>
    <option value="2009">2009</option>
    <option value="2008">2008</option>
    <option value="2007">2007</option>
    <option value="2006">2006</option>
    <option value="2005">2005</option>
    <option value="2004">2004</option>
    <option value="2003">2003</option>
    <option value="2002">2002</option>
    <option value="2001">2001</option>
    <option value="2000">2000</option>
</select>
```

Créer les Labels

`Cake\View\Helper\FormHelper::label(string $fieldName, string $text, array $options)`

- `$fieldName` - Le nom du champ (attribut `name`) sous la forme `'Modelname.fieldname'`.
- `$text` - Chaîne optionnelle pour définir le texte du label.
- `$options` - Optionel. Tableau qui peut contenir n'importe quelles *des options générales* ainsi que n'importe quels attributs HTML valides.

Crée un élément label. `$fieldName` est utilisé pour générer l'attribut `for`. Si `$text` n'est pas défini, `$fieldName` sera utilisé pour définir le texte du label :

```
echo $this->Form->label('name');
echo $this->Form->label('name', 'Votre nom');
```

Affichera :

```
<label for="name">Name</label>
<label for="name">Votre nom</label>
```

Avec le troisième paramètre, `$options`, vous pouvez fixer le nom de la classe ou d'autres attributs :

> echo $this->Form->label("name", null, ["id" => "user-label"]) ; echo $this->Form->label("name", "Votre nom", ["class" => "highlight"]) ;

Affichera :

```
<label for="name" id="user-label">Name</label>
<label for="name" class="highlight">Votre nom</label>
```

Afficher et vérifier les erreurs

FormHelper dispose de quelques méthodes qui vous permettent de vérifier facilement si vos champs contiennent des erreurs et d'afficher des messages d'erreur personnalisés.

Afficher les Erreurs

`Cake\View\Helper\FormHelper::error(string $fieldName, mixed $text, array $options)`

- `$fieldName` - Le nom du champ (attribut `name`) sous la forme `'Modelname.fieldname'`.
- `$text` - Optionnel. Une chaîne ou un tableau fournissant le(s) message(s) d'erreur. Si c'est un tableau, cela devra être un tableau de paires clé / valeur où la clé est le nom du champ en erreur et la valeur le message associé. Défaut à `null`.
- `$options` - Tableau optionnel qui ne peut contenir qu'une clé `escape` qui attend un booléen et qui permet de définir si le contenu HTML du message d'erreur doit être échappé ou non. Défaut à `true`.

Affiche un message d'erreur de validation, spécifié par `$text`, pour le champ donné, dans le cas où une erreur de validation s'est produite. Si `$text` n'est pas fourni alors le message de validation par défaut pour le type de champ sera utilisé.

Cette méthode utilise les templates de widgets suivant :

```
'error' => '<div class="error-message">{{content}}</div>'
'errorList' => '<ul>{{content}}</ul>'
'errorItem' => '<li>{{text}}</li>'
```

Les templates `'errorList'` et `'errorItem'` sont utilisés pour formater plusieurs messages d'erreur pour un seul champ.

Exemple :

```
// Si vous avez une règle de validation 'notEmpty' dans TicketsTable:
public function validationDefault(Validator $validator): Validator
{
    $validator
        ->requirePresence('ticket', 'create')
        ->notEmpty('ticket');
}

// Et dans templates/Tickets/add.php vous avez:
echo $this->Form->text('ticket');

if ($this->Form->isFieldError('ticket')) {
    echo $this->Form->error('ticket', 'Message d\'erreur 100% personnalisé !');
}
```

Si vous soumettez le formulaire sans fournir de valeur pour le champ *Ticket*, votre formulaire affichera :

```
<input name="ticket" class="form-error" required="required" value="" type="text">
<div class="error-message">Message d'erreur 100% personnalisé !</div>
```

> **Note**
>
> En utilisant *control()*, les erreurs sont rendues par défaut, donc vous n'aurez pas besoin d'utiliser `isFieldError()` ou d'appeler `error()` manuellement.

Si vous utilisez un champ particulier du modèle pour générer des champs de formulaire multiples *via* `control()`, et si vous voulez que le même message d'erreur soit utilisé pour chacun d'eux, il sera probablement préférable de définir votre message d'erreur personnalisé dans les *validator rules*.

Vérifier la Présence d'Erreurs

`Cake\View\Helper\FormHelper::isFieldError(string $fieldName)`

— `$fieldName` - Un nom de champ sous la forme `'Modelname.fieldname'`.

Renvoie `true` si le champ `$fieldName` fourni a une erreur de validation en cours. Sinon, retournera `false` :

```
if ($this->Form->isFieldError('gender')) {
    echo $this->Form->error('gender');
}
```

Afficher des messages dans le système de validation HTML5

Si l'option `autoSetCustomValidity` du FormHelper est définie à `true`, les messages du navigateur HTML5 pour signifier qu'un champ est obligatoire seront remplacés par les messages d'erreur des règles de validation *required* et *notBlank* . L'activation de cette option ajoutera les attributs `onvalid` et `oninvalid` à vos champs, par exemple :

```
<input type="text" name="field" required onvalid="this.setCustomValidity('')"
 oninvalid="this.setCustomValidity('Message personnalisé pour la règle notBlank')" />
```

Si vous voulez définir manuellement ces events avec votre propre JavaScript, vous pouvez définir l'option `autoSetCustomValidity` à `false` et utiliser à la place la variable de template spéciale `customValidityMessage`. Cette variable de template est ajoutée quand un champ est obligatoire :

```
// template exemple
[
    'input' => '<input type="{{type}}" name="{{name}}" data-error-message="{
↪{customValidityMessage}}" {{attrs}}/>',
]

// créerait un input ressemblant à cela
<input type="text" name="field" required
 data-error-message="Message personnalisé pour la règle notBlank" />
```

Vous pourriez alors utiliser le JavaScript pour définir les events `onvalid` et `oninvalid` de la façon dont vous le souhaitez.

Création des boutons et des éléments submit

Créer des éléments Submit

`Cake\View\Helper\FormHelper::submit(string $caption, array $options)`

— `$caption` - Chaîne optionnelle qui permet de fournir le texte à afficher ou le chemin vers une image pour le bouton. Défaut à `'Submit'`.
— `$options` - Optionel. Tableau qui peut contenir n'importe quelles *des options générales*, ou des options spécifiques aux boutons submit (cf. ci-dessous), ainsi que n'importe quels attributs HTML valides.

Crée un input `submit` avec la valeur `$caption`. Si la `$caption` fournie est l'URL d'une image (c'est-à-dire si la valeur fournie contient " ://" ou une des extensions ".jpg, .jpe, .jpeg, .gif"), une image cliquable sera générée comme bouton

sibmit (si l'image existe). Si le premier caractère est "/" alors le chemin de l'image sera relatif à *webroot*, sinon, il sera relatif à *webroot/img*.

Par défaut, les templates de widgets utilisés sont :

```
'inputSubmit' => '<input type="{{type}}"{{attrs}}>'
'submitContainer' => '<div class="submit">{{content}}</div>'
```

Options pour les Submit

- `'type'` - Définissez cette option à `'reset'` pour générer un bouton « reset » (de remise à zéro du formulaire). Défaut à `'submit'`.
- `'templateVars'` - Utilisez ce tableau pour fournir des variables de template supplémentaires pour l'élément et ses conteneurs.
- Tout autre paramètre sera ajouté comme un attribut à l'élément HTML `input`.

Le code suivant :

```
echo $this->Form->submit('Cliquez ici');
```

Affichera :

```
<div class="submit"><input value="Cliquez ici" type="submit"></div>
```

Vous pouvez aussi passer une URL relative ou absolue vers une image au paramètre caption au lieu d'un texte :

```
echo $this->Form->submit('ok.png');
```

Affichera :

```
<div class="submit"><input type="image" src="/img/ok.png"></div>
```

Les inputs submit sont utiles quand vous avez seulement besoin de textes basiques ou d'images. Si vous avez besoin d'un contenu de bouton plus complexe, vous devrez plutôt utiliser `button()`.

Créer des Éléments Button

`Cake\View\Helper\FormHelper::button`(*string $title*, *array $options = []*)

- `$title` - Chaîne obligatoire qui correspond au texte du bouton.
- `$options` - Optionel. Tableau qui peut contenir n'importe quelles *des options générales*, ou des options spécifiques aux boutons (cf. ci-dessous), ainsi que n'importe quels attributs HTML valides.

Crée un bouton HTML avec le titre spécifié et un type par défaut `button`.

Options pour les Button

- `'type'` - Vous pouvez définir cette variable à l'une des trois valeurs suivantes :
  1. `'submit'` - Comme pour la méthode `$this->Form->submit()`, cela créera un bouton de type `submit`. Notez cependant que cela ne générera pas de `div` autour comme pour `submit()`. C'est le type par défaut.
  2. `'reset'` - Crée un bouton « reset » (remise à zéro) pour le formulaire.
  3. `'button'` - Crée un bouton standard.
- `'escapeTitle'` - Booléen. Si cette option est définie à `true`, le contenu HTML de la valeur fournie pour `$title` sera échappé. Défaut à `true`.
- `'escape'` - Booléen. S'il est défini à `true`, tous les attributs HTML générés pour le bouton seront échappés. Défaut à `true`.
- `'confirm'` - Le message de confirmation à afficher lors du clic. Défaut à `null`.

Par exemple :

```
echo $this->Form->button('Un bouton');
echo $this->Form->button('Un autre bouton', ['type' => 'button']);
echo $this->Form->button('Réinitialiser le formulaire', ['type' => 'reset']);
echo $this->Form->button('Valider', ['type' => 'submit']);
```

Affichera :

```
<button type="submit">Un bouton</button>
<button type="button">Un autre bouton</button>
<button type="reset">Réinitialiser le formulaire</button>
<button type="submit">Valider</button>
```

Exemple en utilisant l'option `escapeTitle` :

```
// Rendra le code HTML sans échappement.
echo $this->Form->button('<em>Valider</em>', [
    'type' => 'submit',
    'escapeTitle' => false,
]);
```

Fermer le Formulaire

`Cake\View\Helper\FormHelper::end`(*$secureAttributes = []*)

— `$secureAttributes` - Optionnel. Vous permet de fournir des attributs qui seront utilisés comme attributs HTML aux inputs `hidden` générés par le `FormProtectionComponent`.

La méthode `end()` ferme et complète le marquage du formulaire. Souvent, `end()` se contente d'afficher la balise fermante du formulaire, mais l'utilisation de `end()` est une bonne pratique puisqu'elle permet également au FormHelper d'ajouter les champs cachées dont le `Cake\Controller\Component\FormProtectionComponent` a besoin :

```
<?= $this->Form->create(); ?>

<!-- Éléments de formulaire ici -->

<?= $this->Form->end(); ?>
```

Si vous avez besoin d'appliquer des attributs supplémentaires aux inputs `hidden`, vous pouvez utiliser l'argument `$secureAttributes`.

Ainsi :

```
echo $this->Form->end(['data-type' => 'hidden']);
```

Affichera :

```
<div style="display:none;">
    <input type="hidden" name="_Token[fields]" data-type="hidden"
        value="2981c38990f3f6ba935e6561dc77277966fabd6d%3AAddresses.id">
    <input type="hidden" name="_Token[unlocked]" data-type="hidden"
        value="address%7Cfirst_name">
</div>
```

> **Note**
>
> Si vous utilisez `Cake\Controller\Component\FormProtectionComponent` dans votre application, vous devrez obligatoirement terminer vos formulaires avec `end()`.

Créer des Boutons Indépendants et des liens POST

Créer des Boutons POST

`Cake\View\Helper\FormHelper::postButton`(*string $title*, *mixed $url*, *array $options = []*)

- `$title` - Chaîne obligatoire qui sera utilisée comme texte du bouton. Notez que, par défaut, cette valeur ne sera pas échappée.
- `$url` - URL cible du formulaire, sous forme de chaîne ou de tableau.
- `$options` - Optionel. Tableau qui peut contenir n'importe quelles *des options générales*, ou des options spécifiques (cf. ci-dessous), ainsi que n'importe quels attributs HTML valides.

Crée une balise `<button>` avec un `<form>` l'entourant qui soumet par défaut une requête POST. De plus, par défaut, cela générera des inputs `hidden` pour le `FormProtectionComponent`.

Options for POST Button

- `'data'` - Tableau clé / valeur à passer aux inputs `hidden`.
- `'method'` - La méthode de requête à utiliser. Par exemple si vous voulez émettre une requête HTTP/1.1 DELETE, passez `delete`. La valeur par défaut est `post`.
- `'form'` - Tableau dans lequel vous pouvez passer n'importe quelle valeur supportée par `FormHelper::create()`.
- De plus, la méthode `postButton()` acceptera n'importe quelle option également valide pour la méthode `button()`.

Par exemple

```
// Dans templates/Tickets/index.php
<?= $this->Form->postButton('Supprimer', ['controller' => 'Tickets', 'action' => 'delete
↪', 5]) ?>
```

Affichera un HTML similaire à :

```
<form method="post" accept-charset="utf-8" action="/Rtools/tickets/delete/5">
    <div style="display:none;">
        <input name="_method" value="POST" type="hidden">
    </div>
    <button type="submit">Supprimer</button>
    <div style="display:none;">
        <input name="_Token[fields]" value="186cfbfc6f519622e19d1e688633c4028229081f%3A"␣
↪type="hidden">
        <input name="_Token[unlocked]" value="" type="hidden">
        <input name="_Token[debug]" value="%5B%22%5C%2FRtools%5C%2Ftickets%5C%2Fdelete%5C
↪%2F1%22%2C%5B%5D%2C%5B%5D%5D" type="hidden">
    </div>
</form>
```

Dans la mesure où cette méthode crée un élément `<form>`, ne l'utilisez pas à l'intérieur d'un formulaire ouvert. Utilisez plutôt *`Cake\View\Helper\FormHelper::submit()`* ou *`Cake\View\Helper\FormHelper::button()`* pour créer des boutons à l'intérieur de formulaires ouvert.

Créer des liens POST

`Cake\View\Helper\FormHelper::postLink(string $title, mixed $url = null, array $options = [])`

- `$title` - Chaîne obligatoire qui sera utilisée comme texte du lien.
- `$url` - URL cible du formulaire, sous forme de chaîne ou de tableau. L'URL est soit relative à CakePHP, soit externe si elle commence par `http://`.
- `$options` - Optionel. Tableau qui peut contenir n'importe quelles *des options générales*, ou des options spécifiques (cf. ci-dessous), ainsi que n'importe quels attributs HTML valides.

Crée un lien HTML, mais accède à l'Url en utilisant la méthode spécifiée (par défaut POST). Requiert que JavaScript soit activé dans le navigateur :

```
// Dans votre template, à l'emplacement pour supprimer un article
<?= $this->Form->postLink(
   'Supprimer',
   ['action' => 'delete', $article->id],
   ['confirm' => 'Êtes-vous sûr ?'])
?>
```

Options pour les liens POST

- `'data'` - Tableau clé / valeur à passer aux inputs `hidden`.
- `'method'` - La méthode de requête à utiliser. Par exemple si vous voulez émettre une requête HTTP/1.1 DELETE, passez `delete`. La valeur par défaut est `post`.
- `'confirm'` - Le message de confirmation à afficher lors du clic sur le lien. Défaut à `null`.
- `'block'` - Définissez cette option à `true` pour ajouter le lien au « view block » `'postLink'` ou pour fournir un nom de bloc personnalisé. Défaut à `null`.
- De plus, la méthode `postLink` acceptera n'importe quelle option également valide pour la méthode `link()`.

Cette méthode crée un élément `<form>`. Si vous souhaitez utiliser cette méthode à l'intérieur d'un formulaire existant, vous devez utiliser l'option `block` pour que le nouveau formulaire soit défini dans un *bloc de vue* qui peut être affiché en dehors du formulaire principal.

Si vous souhaitez seulement créer un bouton pour soumettre votre formulaire, alors vous devriez plutôt utiliser *`Cake\View\Helper\FormHelper::button()`* ou *`Cake\View\Helper\FormHelper::submit()`*.

> **Note**
>
> Attention à ne pas mettre un postLink à l'intérieur d'un formulaire ouvert. À la place, utilisez l'option `block` pour mettre en mémoire tampon le formulaire dans un *Utiliser les Blocks de Vues*.

Personnaliser les Templates Utilisés par FormHelper

Comme beaucoup de helpers dans CakePHP, FormHelper utilise les chaînes de template pour mettre en forme le HTML qu'il crée. Bien que les templates par défaut soient destinés à fournir un ensemble raisonnable de canevas pour les usages courants, vous aurez peut-être besoin de personnaliser des templates qui correspondront davantage à votre application.

Pour changer les templates au moment du chargement du helper, vous pouvez définir l'option `templates` lors de l'inclusion du helper dans votre controller :

```
// Dans une classe de View
$this->loadHelper('Form', [
    'templates' => 'app_form',
]);
```

Ceci chargera les balises contenues dans **config/app_form.php**. Ce fichier devra contenir un tableau des templates *indexés par leur nom* :

```
// dans config/app_form.php
return [
    'inputContainer' => '<div class="form-control">{{content}}</div>',
];
```

Tous les templates que vous définirez vont remplacer ceux inclus par défaut dans le helper. Les Templates qui ne sont pas remplacés vont continuer à être utilisés avec les valeurs par défaut.

Vous pouvez aussi changer les templates à la volée en utilisant la méthode `setTemplates()` :

```
$myTemplates = [
    'inputContainer' => '<div class="form-control">{{content}}</div>',
];
$this->Form->setTemplates($myTemplates);
```

Avertissement

Les chaînes de template contenant un signe pourcentage (%) nécessitent une attention spéciale. Vous devriez préfixer ce caractère avec un autre pourcentage pour le faire ressembler à %%. La raison est que les templates sont compilés en interne pour être utilisés avec `sprintf()`. Exemple : `<div style="width:{{size}}%%">{{content}}</div>`

Liste des Templates

La liste des templates par défaut, leur format par défaut et les variables qu'ils attendent se trouvent dans la documentation API du FormHelper [129].

Utiliser des conteneurs personnalisés distincts pour les éléments

En plus de ces templates, la méthode `control()` va essayer d'utiliser des templates différents pour chaque conteneur d'input. Par exemple, lors de la création d'un input datetime, `datetimeContainer` va être utilisé s'il est présent. Si le conteneur n'est pas présent, le template `inputContainer` sera utilisé. Par exemple :

```
// Ajoute du HTML personnalisé autour d'un input radio
$this->Form->setTemplates([
    'radioContainer' => '<div class="form-radio">{{content}}</div>'
]);

// Crée un ensemble d'inputs radio avec notre div personnalisé autour
echo $this->Form->control('email_notifications', [
    'options' => ['y', 'n'],
    'type' => 'radio'
]);
```

129. https://api.cakephp.org/4.x/class-Cake.View.Helper.FormHelper.html#%24_defaultConfig

Utiliser des groupes de formulaire personnalisés distincts

De la même manière qu'avec les conteneurs d'input, la méthode `control()` essayera d'utiliser différents templates pour chaque groupe de formulaire. Un groupe de formulaire est un ensemble composé d'un label et d'un input. Par exemple, lorsque vous créez des inputs de type radio, le template `radioFormGroup` sera utilisé s'il est présent. Si ce template est absent, par défaut chaque ensemble `label` & `input` sera généré en utilisant le template `formGroup` :

```
// Ajoute un groupe de formulaire personnalisé pour les boutons radio
$this->Form->setTemplates([
    'radioFormGroup' => '<div class="radio">{{label}}{{input}}</div>'
]);
```

Ajouter des Variables de Template Supplémentaires aux Templates

Vous pouvez aussi ajouter des espaces réservés supplémentaires dans des templates personnalisés et les remplir lors de la génération des inputs.

Par exemple :

```
// Ajoute un template avec un espace réservé pour un message d'aide
$this->Form->setTemplates([
    'inputContainer' => '<div class="input {{type}}{{required}}">
        {{content}} <span class="help">{{help}}</span></div>'
]);

// Génère un input et remplit la variable help
echo $this->Form->control('password', [
    'templateVars' => ['help' => 'Au moins 8 caractères.']
]);
```

Affichera :

```
<div class="input password">
    <label for="password">
        Password
    </label>
    <input name="password" id="password" type="password">
    <span class="help">Au moins 8 caractères.</span>
</div>
```

Déplacer les Checkboxes & Boutons Radios à l'Extérieur du Label

Par défaut, CakePHP incorpore les cases à cocher créées via `control()` et les boutons radios créés par `control()` et `radio()` dans des éléments label. Cela contribue à faciliter l'intégration des frameworks CSS populaires. Si vous avez besoin de placer ces éléments à l'extérieur de la balise label, vous pouvez le faire en modifiant les templates :

```
$this->Form->setTemplates([
    'nestingLabel' => '{{hidden}}{{input}}<label{{attrs}}>{{text}}</label>',
    'formGroup' => '{{input}}{{label}}',
]);
```

Cela générera les checkbox et les boutons radio à l'extérieur de leurs labels.

Générer des Formulaires Entiers

Créer plusieurs éléments (controls)

`Cake\View\Helper\FormHelper::controls(array $fields = [], array $options = [])`

- `$fields` - Un tableau des champs à générer. Permet de définir des types personnalisés, des labels et toutes autres options pour chaque champ.
- `$options` - Optionnel. Un tableau d'options. Les clés supportées sont :
  1. `'fieldset'` - Définir à `false` pour désactiver l'ajout d'un `fieldset`. Si vide, le `fieldset` sera ajouté. Peut aussi être un tableau de paramètres à appliquer comme attributs HTML au `fieldset` généré.
  2. `legend` - Chaîne utilisée pour personnaliser le texte de l'élément `legend`. Définir à `false` pour désactiver l'ajout de l'élément `legend`.

Génère un ensemble d'inputs pour un contexte donné, entouré d'un `fieldset`. Vous pouvez spécifier les champs générés en les incluant :

```
echo $this->Form->controls([
    'name',
    'email'
]);
```

Vous pouvez personnaliser le texte de la légende en utilisant une option :

```
echo $this->Form->controls($fields, ['legend' => 'Mettre à jour le post']);
```

Vous pouvez personnaliser les inputs générés en définissant des options supplémentaires dans le paramètre `$fields` :

```
echo $this->Form->controls([
    'name' => ['label' => 'Label personnalisé']
]);
```

Quand vous personnalisez `fields`, vous pouvez utiliser le paramètre `$options` pour contrôler les éléments `legend` et `fieldset` générés.

Par exemple :

```
echo $this->Form->controls(
    [
        'name' => ['label' => 'Label personnalisé']
    ],
    ['legend' => 'Mettre à jour votre post']
);
```

Si vous désactiver le `fieldset`, la `legend` ne s'affichera pas.

Créer les éléments pour une Entity complète

`Cake\View\Helper\FormHelper::allControls(array $fields, $options = [])`

- `$fields` - Optionnel. Un tableau de paramétrages pour les champs qui seront générés. Permet de définir des types personnalisés, des labels et toutes autres options.
- `$options` - Optionnel. Un tableau d'options. Les clés supportées sont :
  1. `'fieldset'` - Définir à `false` pour désactiver l'ajout d'un `fieldset`. Si vide, le `fieldset` sera ajouté. Peut aussi être un tableau de paramètres à appliquer comme attributs HTML au `fieldset` généré.

2. `legend` - Chaîne utilisée pour personnaliser le texte de l'élément `legend`. Définir à `false` pour désactiver l'ajout de l'élément `legend`.

Cette méthode est étroitement liée à `controls()`, cependant l'argument `$fields` est égal par défaut à *tous* les champs de l'entity de niveau supérieur actuelle. Pour exclure certains champs de la liste d'inputs générée, définissez-les à `false` dans le paramètre `$fields` :

```
echo $this->Form->allControls(['password' => false]);
```

Créer des Inputs pour les Données Associées

Créer des formulaires pour les données associées est assez simple et est étroitement lié aux chemins des données de votre entity. Imaginons les relations suivantes :

- Authors HasOne Profiles
- Authors HasMany Articles
- Articles HasMany Comments
- Articles BelongsTo Authors
- Articles BelongsToMany Tags

Si nous éditions un article avec ces associations chargées, nous pourrions créer les inputs suivantes :

```
$this->Form->create($article);

// Inputs article
echo $this->Form->control('title');

// Inputs auteur (belongsTo)
echo $this->Form->control('author.id');
echo $this->Form->control('author.first_name');
echo $this->Form->control('author.last_name');

// Profile de l'auteur (belongsTo + hasOne)
echo $this->Form->control('author.profile.id');
echo $this->Form->control('author.profile.username');

// Inputs Tags (belongsToMany)
// en tant qu'inputs séparés
echo $this->Form->control('tags.0.id');
echo $this->Form->control('tags.0.name');
echo $this->Form->control('tags.1.id');
echo $this->Form->control('tags.1.name');

// Inputs pour la table de jointure (articles_tags)
echo $this->Form->control('tags.0._joinData.starred');
echo $this->Form->control('tags.1._joinData.starred');

// Inputs commentaires (hasMany)
echo $this->Form->control('comments.0.id');
echo $this->Form->control('comments.0.comment');
echo $this->Form->control('comments.1.id');
echo $this->Form->control('comments.1.comment');
```

Le code ci-dessus pourrait ensuite être converti en un graph d'entity en utilisant le code suivant dans votre controller :

```
$article = $this->Articles->patchEntity($article, $this->request->getData(), [
    'associated' => [
        'Authors',
        'Authors.Profiles',
        'Tags',
        'Comments'
    ]
]);
```

Ajouter des Widgets Personnalisés

Vous pouvez ajouter des widgets personnalisés dans CakePHP, et les utiliser comme n'importe quel input. Tous les types d'input que contient le cœur de cake sont implémentés comme des widgets. Ainsi vous pouvez remplacer n'importe quel widget de base par votre propre implémentation.

Construire une Classe Widget

L'interface que les classes Widget doivent respecter est vraiment simple. Il s'agit de `Cake\View\Widget\WidgetInterface`. Cette interface nécessite d'implémenter les méthodes `render(array $data)` et `secureFields(array $data)`. La méthode `render()` attend un tableau de données pour construire le widget et doit renvoyer une chaine HTML pour le widget. La méthode `secureFields()` attend également un tableau de données et doit retourner un tableau contenant la liste des champs à sécuriser pour ce widget. Si CakePHP construit votre widget, vous pouvez vous attendre à recevoir une instance de `Cake\View\StringTemplate` en premier argument, suivi de toutes les dépendances que vous aurez définies. Si vous vouliez construire un widget Autocomplete, vous pourriez le faire comme ceci :

```
namespace App\View\Widget;

use Cake\View\Form\ContextInterface;
use Cake\View\StringTemplate;
use Cake\View\Widget\WidgetInterface;

class AutocompleteWidget implements WidgetInterface
{
    /**
     * StringTemplate instance.
     *
     * @var \Cake\View\StringTemplate
     */
    protected $_templates;

    /**
     * Constructor.
     *
     * @param \Cake\View\StringTemplate $templates Liste des templates.
     */
    public function __construct(StringTemplate $templates)
    {
        $this->_templates = $templates;
    }

    /**
```

(suite sur la page suivante)

(suite de la page précédente)

```
     * Méthode qui fait le rendu du widget.
     *
     * @param array $data Les données à utiliser pour construire l'input.
     * @param \Cake\View\Form\ContextInterface $context Le contexte courant du␣
↪formulaire.
     *
     * @return string
     */
    public function render(array $data, ContextInterface $context): string
    {
        $data += [
            'name' => '',
        ];

        return $this->_templates->format('autocomplete', [
            'name' => $data['name'],
            'attrs' => $this->_templates->formatAttributes($data, ['name'])
        ]);
    }

    public function secureFields(array $data): array
    {
        return [$data['name']];
    }
}
```

Évidemment, c'est un exemple très simple, mais il montre comment développer un widget personnalisé. Ce widget ferait un rendu de la chaîne de template « autocomplete », si définie comme ceci par exemple :

```
$this->Form->setTemplates([
    'autocomplete' => '<input type="autocomplete" name="{{name}}" {{attrs}} />'
]);
```

Pour plus d'informations sur les templates, référez-vous à la section *Personnaliser les Templates Utilisés par FormHelper*.

Utiliser les Widgets

Vous pouvez charger des widgets personnalisés lors du chargement du FormHelper ou en utilisant la méthode `addWidget()`. Lors du chargement du FormHelper, les widgets sont définis comme des paramètres :

```
// Dans une classe de View
$this->loadHelper('Form', [
    'widgets' => [
        'autocomplete' => ['Autocomplete']
    ]
]);
```

Si votre widget nécessite d'autres widgets, le FormHelper peut remplir ces dépendances lorsqu'elles sont déclarées :

```
$this->loadHelper('Form', [
    'widgets' => [
        'autocomplete' => [
```

(suite sur la page suivante)

(suite de la page précédente)

```
            'App\View\Widget\AutocompleteWidget',
            'text',
            'label'
        ]
    ]
]);
```

Dans l'exemple ci-dessus, le widget `autocomplete` dépendrait des widgets `text` et `label`. Si votre widget doit accéder à la View, vous devrez utiliser le "widget" `_view`. Lorsque le widget `autocomplete` est créé, les objets widget liés aux noms `text` et `label` lui sont passés. Ajouter des widgets en utilisant la méthode `addWidget` ressemblerait à ceci :

```
// En utilisant un nom de classe.
$this->Form->addWidget(
    'autocomplete',
    ['Autocomplete', 'text', 'label']
);

// En utilisant une instance - vous oblige à résoudre les dépendances.
$autocomplete = new AutocompleteWidget(
    $this->Form->getTemplater(),
    $this->Form->widgetRegistry()->get('text'),
    $this->Form->widgetRegistry()->get('label'),
);
$this->Form->addWidget('autocomplete', $autocomplete);
```

Une fois ajoutés/remplacés, les widgets peuvent être utilisés en tant que "type" de l'input :

```
echo $this->Form->control('search', ['type' => 'autocomplete']);
```

Cela créera un widget personnalisé avec un `label` et une `div` enveloppante tout comme le fait toujours `control()`. Sinon vous pouvez juste créer un widget en utilisant la méthode magique :

```
echo $this->Form->autocomplete('search', $options);
```

Travailler avec FormProtectionComponent

`Cake\Controller\Component\FormProtectionComponent` offre plusieurs fonctionnalités qui rendent vos formulaires plus sûrs et plus sécurisés. En incluant simplement le `FormProtectionComponent` dans votre controller, vous bénéficierez automatiquement des fonctionnalités de prévention contre la falsification de formulaires.

Comme mentionné précédemment, lorsque vous utilisez le FormProtectionComponent, vous devez toujours fermer vos formulaires en utilisant *end()*. Cela assurera que les inputs spéciales `_Token` soient générées.

`Cake\View\Helper\FormHelper::`**`unlockField`**(*$name*)

— `$name` - Optionnel. Le nom du champ en notation avec point (sous la forme `'Modelname.fieldname'`).

Déverrouille un champ en l'exemptant du hachage par `FormProtectionComponent`. Cela autorise également le client à manipuler le champ via JavaScript. Le paramètre `$name` doit correspondre au nom de la propriété de l'entity pour l'input :

```
$this->Form->unlockField('id');
```

`Cake\View\Helper\FormHelper::`**`secure`**(*array $fields = [], array $secureAttributes = []*)

— `$fields` - Optionnel. Un tableau contenant la liste des champs à utiliser lors de la génération du hash. S'il n'est pas fourni, alors `$this->fields` sera utilisé.
— `$secureAttributes` - Optionnel. Un tableau d'attributs HTML à passer aux élément `input` de type `hidden` qui seront générés.

Génère un `input` de type `hidden` avec un hash de sécurité basé sur les champs utilisés dans le formulaire, ou une chaîne vide si la sécurisation des formulaires n'est pas utilisée. Si l'option `$secureAttributes` est définie, ces attributs HTML seront fusionnés avec ceux générés par le FormProtectionComponent. C'est particulièrement utile pour définir des attributs HTML5 tels que `'form'`.

Html

class Cake\View\Helper**HtmlHelper**(*View $view*, *array $config = []*)

Le rôle du Helper Html dans CakePHP est de fabriquer les options du HTML plus facilement, plus rapidement. L'utilisation de ce Helper permettra à votre application d'être plus légère, bien ancrée et plus flexible de l'endroit où il est placé en relation avec la racine de votre domaine.

De nombreuses méthodes du Helper Html contiennent un paramètre `$attributes`, qui vous permet d'insérer un attribut supplémentaire sur vos tags. Voici quelques exemples sur la façon d'utiliser les paramètres `$attributes` :

```
Attributs souhaités: <tag class="someClass" />
Paramètre du tableau: ['class' => 'someClass']

Attributs souhaités: <tag name="foo" value="bar" />
Paramètre du tableau:  ['name' => 'foo', 'value' => 'bar']
```

Insertion d'éléments correctement formatés

La tâche la plus importante que le Helper Html accomplit est la création d'un balisage bien formé. Cette section couvrira quelques méthodes du Helper Html et leur utilisation.

Créer un Tag Charset

Cake\View\Helper\HtmlHelper::**charset**(*$charset=null*)

Utilisé pour créer une balise meta spécifiant le jeu de caractères du document. UTF-8 par défaut. Exemple d'utilisation :

```
echo $this->Html->charset();
```

Affichera :

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
```

Sinon :

```
echo $this->Html->charset('ISO-8859-1');
```

Affichera :

```
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
```

Lier des Fichiers CSS

`Cake\View\Helper\HtmlHelper::css(mixed $path, array $options = [])`

Crée un ou plusieurs lien(s) vers une feuille de style CSS. Si l'option `block` est définie à `true`, les balises de liens sont ajoutées au bloc `css` qui peut être dans la balise head du document.

Vous pouvez utiliser l'option `block` pour contrôler sur lequel des blocs l'élément lié sera ajouté. Par défaut il sera ajouté au bloc `css`.

Si la clé "rel" dans le tableau `$options` est définie à "import", la feuille de style sera importée.

Cette méthode d'inclusion CSS présume que le CSS spécifié se trouve dans le répertoire **webroot/css** si le chemin ne commence pas par un "/" :

```
echo $this->Html->css('forms');
```

Affichera :

```
<link rel="stylesheet" href="/css/forms.css" />
```

Le premier paramètre peut être un tableau pour inclure des fichiers multiples :

```
echo $this->Html->css(['forms', 'tables', 'menu']);
```

Affichera :

```
<link rel="stylesheet" href="/css/forms.css" />
<link rel="stylesheet" href="/css/tables.css" />
<link rel="stylesheet" href="/css/menu.css" />
```

Vous pouvez inclure un fichier CSS depuis un plugin chargé en utilisant la *syntaxe de plugin*. Pour inclure **plugins/DebugKit/webroot/css/toolbar.css**, vous pouvez utiliser ce qui suit :

```
echo $this->Html->css('DebugKit.toolbar.css');
```

Si vous voulez inclure un fichier CSS qui partage un nom avec un plugin chargé vous pouvez faire ce qui suit. Par exemple vous avez un plugin `Blog`, et souhaitez inclure également **webroot/css/Blog.common.css** vous pouvez faire ceci :

```
echo $this->Html->css('Blog.common.css', ['plugin' => false]);
```

Créer des CSS par Programmation

`Cake\View\Helper\HtmlHelper::style(array $data, boolean $oneline = true)`

Construit les définitions de style CSS en se basant sur les clés et valeurs du tableau passé à la méthode. Particulièrement pratique si votre fichier CSS est dynamique :

```
echo $this->Html->style([
    'background' => '#633',
    'border-bottom' => '1px solid #000',
    'padding' => '10px'
]);
```

Affichera

```
background:#633; border-bottom:1px solid #000; padding:10px;
```

Créer des Balises meta

`Cake\View\Helper\HtmlHelper::meta(string|array $type, string $url = null, array $options = [])`

Cette méthode est pratique pour faire des liens vers des ressources externes comme RSS/Atom feeds et les favicons. Comme avec css(), vous pouvez spécifier si vous voulez l'apparition de la balise en ligne ou l'ajouter au bloc `meta` en définissant la clé "block" à `true` dans les paramètres $attributes, ex. - `['block' => true]`.

Si vous définissez l'attribut « type » en utilisant le paramètre $attributes, CakePHP contient quelques raccourcis :

type	valeur résultante
html	text/html
rss	application/rss+xml
atom	application/atom+xml
icon	image/x-icon

```
<?= $this->Html->meta(
    'favicon.ico',
    '/favicon.ico',
    ['type' => 'icon']
);
?>
// Affiche (saut de lignes ajoutés)
<link
    href="http://example.com/favicon.ico"
    title="favicon.ico" type="image/x-icon"
    rel="alternate"
/>
<?= $this->Html->meta(
    'Comments',
    '/comments/index.rss',
    ['type' => 'rss']
);
?>
// Affiche (saut de lignes ajoutés)
<link
    href="http://example.com/comments/index.rss"
    title="Comments"
    type="application/rss+xml"
    rel="alternate"
/>
```

Cette méthode peut aussi être utilisée pour ajouter les meta keywords (mots clés) et descriptions. Exemple :

```
<?= $this->Html->meta(
    'keywords',
    'entrez vos mots clés pour la balise meta ici'
);
?>
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Affiche
<meta name="keywords" content="entrez vos mots clés pour la balise meta ici" />

<?= $this->Html->meta(
    'description',
    'entrez votre description pour la balise meta ici'
);
?>
// Affiche
<meta name="description" content="entrez votre description pour la balise meta ici" />
```

En plus de faire des balises meta prédéfinies, vous pouvez créer des éléments de lien :

```
<?= $this->Html->meta([
    'link' => 'http://example.com/manifest',
    'rel' => 'manifest'
]);
?>
// Affiche
<link href="http://example.com/manifest" rel="manifest"/>
```

Tout attribut fourni à meta() lorsqu'elle est appelée de cette façon, sera ajoutée à la balise de lien générée.

Créer le DOCTYPE

`Cake\View\Helper\HtmlHelper::docType(string $type = 'html5')`

Retourne une déclaration DOCTYPE (*document type declaration*) (X)HTML. Spécifiez le DOCTYPE souhaité selon la table suivante :

type	valeur finale
html4-strict	HTML 4.01 Strict
html4-trans	HTML 4.01 Transitional
html4-frame	HTML 4.01 Frameset
html5 (défaut)	HTML5
xhtml-strict	XHTML 1.0 Strict
xhtml-trans	XHTML 1.0 Transitional
xhtml-frame	XHTML 1.0 Frameset
xhtml11	XHTML 1.1

```
echo $this->Html->docType();
// Affiche: <!DOCTYPE html>

echo $this->Html->docType('html4-trans');
// Affiche:
// <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
//    "http://www.w3.org/TR/html4/loose.dtd">
```

Lier des Images

`Cake\View\Helper\HtmlHelper::image(string $path, array $options = [])`

Crée une balise image formatée. Le chemin fourni devra être relatif à **webroot/img/** :

```
echo $this->Html->image('cake_logo.png', ['alt' => 'CakePHP']);
```

Affichera :

```
<img src="/img/cake_logo.png" alt="CakePHP" />
```

Pour créer un lien d'image, spécifiez le lien de destination en utilisant l'option `url` dans `$attributes` :

```
echo $this->Html->image("recipes/6.jpg", [
    "alt" => "Brownies",
    'url' => ['controller' => 'Recipes', 'action' => 'view', 6]
]);
```

Affichera :

```
<a href="/recipes/view/6">
    <img src="/img/recipes/6.jpg" alt="Brownies" />
</a>
```

Si vous créez des images dans des emails, ou si vous voulez des chemins absolus pour les images, vous pouvez utiliser l'option `fullBase` :

```
echo $this->Html->image("logo.png", ['fullBase' => true]);
```

Affichera :

```
<img src="http://example.com/img/logo.jpg" alt="" />
```

Vous pouvez inclure des fichiers images depuis un plugin chargé en utilisant *syntaxe de plugin*. Pour inclure **plugins/DebugKit/webroot/img/icon.png**, vous pouvez faire cela :

```
echo $this->Html->image('DebugKit.icon.png');
```

Si vous voulez inclure un fichier image qui partage un nom avec un plugin chargé vous pouvez faire ce qui suit. Par exemple si vous avez un plugin `Blog`, et si vous voulez également inclure **webroot/img/Blog.icon.png**, vous feriez :

```
echo $this->Html->image('Blog.icon.png', ['plugin' => false]);
```

Créer des Liens

`Cake\View\Helper\HtmlHelper::link(string $title, mixed $url = null, array $options = [])`

Méthode générale pour la création de liens HTML. Utilisez les `$options` pour spécifier les attributs des éléments et si le `$title` doit ou non être échappé :

```
echo $this->Html->link(
    'Enter',
    '/pages/home',
    ['class' => 'button', 'target' => '_blank']
);
```

Affichera :

```
<a href="/pages/home" class="button" target="_blank">Enter</a>
```

Utilisez l'option `'_full'=>true` pour des URLs absolues :

```
echo $this->Html->link(
    'Dashboard',
    ['controller' => 'Dashboards', 'action' => 'index', '_full' => true]
 );
```

Affichera :

```
<a href="http://www.yourdomain.com/dashboards/index">Dashboard</a>
```

Spécifiez la clé `confirm` dans les options pour afficher une boite de dialogue de confirmation JavaScript `confirm()` :

```
echo $this->Html->link(
    'Delete',
    ['controller' => 'Recipes', 'action' => 'delete', 6],
    ['confirm' => 'Are you sure you wish to delete this recipe?']
);
```

Affichera :

```
<a href="/recipes/delete/6"
    onclick="return confirm(
        'Are you sure you wish to delete this recipe?'
    );">
    Delete
</a>
```

Les chaînes de requête peuvent aussi être créées avec `link()` :

```
echo $this->Html->link('View image', [
    'controller' => 'Images',
    'action' => 'view',
    1,
    '?' => ['height' => 400, 'width' => 500]
]);
```

Affichera :

```
<a href="/images/view/1?height=400&width=500">View image</a>
```

Les caractères spéciaux HTML de `$title` seront convertis en entités HTML. Pour désactiver cette conversion, définissez l'option escape à `false` dans le tableau `$options` :

```
echo $this->Html->link(
    $this->Html->image("recipes/6.jpg", ["alt" => "Brownies"]),
    "recipes/view/6",
    ['escape' => false]
);
```

Affichera :

```
<a href="/recipes/view/6">
    <img src="/img/recipes/6.jpg" alt="Brownies" />
</a>
```

Définir `escape` à `false` va aussi désactiver l'échappement des attributs du lien. Vous pouvez utiliser l'option `escapeTitle` pour juste désactiver l'échappement du titre et pas des attributs :

```
echo $this->Html->link(
    $this->Html->image('recipes/6.jpg', ['alt' => 'Brownies']),
    'recipes/view/6',
    ['escapeTitle' => false, 'title' => 'hi "howdy"']
);
```

Affichera :

```
<a href="/recipes/view/6" title="hi &quot;howdy&quot;">
    <img src="/img/recipes/6.jpg" alt="Brownies" />
</a>
```

Regardez aussi la méthode *`Cake\View\Helper\UrlHelper::build()`* pour plus d'exemples des différents types d'URLs.

Liens vers des Videos et Fichiers Audio

`Cake\View\Helper\HtmlHelper::`**`media`**(*string|array $path*, *array $options*)

Options :

- `type` Type d'éléments média à générer, les valeurs valides sont « audio » ou « video ». Si le type n'est pas fourni le type de média se basera sur le type mime du fichier.
- `text` Texte à inclure dans la balise vidéo.
- `pathPrefix` Préfixe du chemin à utiliser pour les URLs relatives, par défaut à "files/".
- `fullBase` S'il est fourni, l'attribut src prendra l'adresse complète incluant le nom de domaine.

Retourne une balise formatée audio/video :

```
<?= $this->Html->media('audio.mp3') ?>

// Sortie
<audio src="/files/audio.mp3"></audio>

<?= $this->Html->media('video.mp4', [
    'fullBase' => true,
    'text' => 'Fallback text'
]) ?>

// Sortie
<video src="http://www.somehost.com/files/video.mp4">Fallback text</video>

   <?= $this->Html->media(
        ['video.mp4', ['src' => 'video.ogg', 'type' => "video/ogg; codecs='theora, vorbis
↪'"]],
        ['autoplay']
    ) ?>

    // Sortie
```

(suite sur la page suivante)

(suite de la page précédente)

```
    <video autoplay="autoplay">
        <source src="/files/video.mp4" type="video/mp4"/>
        <source src="/files/video.ogg" type="video/ogg;
            codecs='theora, vorbis'"/>
    </video>
```

Lier des Fichiers Javascript

`Cake\View\Helper\HtmlHelper::script(mixed $url, mixed $options)`

Inclus un(des) fichier(s), présent soit localement soit à une URL distante.

Par défaut, les balises du script sont ajoutées au document inline. Si vous le surchargez en configurant `$options['block']` à `true`, les balises du script vont plutôt être ajoutées au block `script` que vous pouvez afficher ailleurs dans le document. Si vous souhaitez surcharger le nom du block utilisé, vous pouvez le faire en configurant `$options['block']`.

`$options['once']` contrôle si vous voulez ou non inclure le script une fois par requête. Par défaut à `true`.

Vous pouvez utiliser $options pour définir des propriétés supplémentaires pour la balise script générée. Si un tableau de balise script est utilisé, les attributs seront appliqués à toutes les balises script générées.

Cette méthode d'inclusion de fichier JavaScript suppose que les fichiers JavaScript spécifiés se trouvent dans le répertoire `webroot/js` :

```
echo $this->Html->script('scripts');
```

Affichera :

```
<script src="/js/scripts.js"></script>
```

Vous pouvez lier à des fichiers avec des chemins absolus tant qu'ils ne se trouvent pas dans `webroot/js` :

```
echo $this->Html->script('/autrerep/fichier_script');
```

Vous pouvez aussi lier à une URL d'un dépôt distant :

```
echo $this->Html->script('https://code.jquery.com/jquery.min.js');
```

Affichera :

```
<script src="https://code.jquery.com/jquery.min.js"></script>
```

Le premier paramètre peut être un tableau pour inclure des fichiers multiples :

```
echo $this->Html->script(['jquery', 'wysiwyg', 'scripts']);
```

Affichera :

```
<script src="/js/jquery.js"></script>
<script src="/js/wysiwyg.js"></script>
<script src="/js/scripts.js"></script>
```

Vous pouvez insérer dans la balise script un bloc spécifique en utilisant l'option `block` :

```
$this->Html->script('wysiwyg', ['block' => 'scriptBottom']);
```

Dans votre layout, vous pouvez afficher toutes les balises script ajoutées dans "scriptBottom" :

```
echo $this->fetch('scriptBottom');
```

Vous pouvez inclure des fichiers de script depuis un plugin en utilisant la *syntaxe de plugin*. Pour inclure **plugins/DebugKit/webroot/js/toolbar.js** vous pouvez faire cela :

```
echo $this->Html->script('DebugKit.toolbar.js');
```

Si vous voulez inclure un fichier de script qui partage un nom de fichier avec un plugin chargé vous pouvez faire cela. Par exemple si vous avez Un plugin `Blog`, et voulez inclure également **webroot/js/Blog.plugins.js**, vous feriez :

```
echo $this->Html->script('Blog.plugins.js', ['plugin' => false]);
```

Créer des Blocs Javascript Inline

`Cake\View\Helper\HtmlHelper::scriptBlock`(*string $code*, *array $options = []*)

Pour générer des blocks Javascript à partir d'un code de vue en PHP, vous pouvez utiliser une des méthodes de script de blocks. Les scripts peuvent soit être affichés à l'endroit où ils sont écrits, soit être mis en mémoire tampon dans un block :

```
// Définit un block de script en une fois, avec l'attribut defer.
$this->Html->scriptBlock('alert("hi")', ['defer' => true]);

// Mis en mémoire d'un block de script pour être affiché plus tard.
$this->Html->scriptBlock('alert("hi")', ['block' => true]);
```

`Cake\View\Helper\HtmlHelper::scriptStart`(*array $options = []*)

`Cake\View\Helper\HtmlHelper::scriptEnd`()

Vous pouvez utiliser la méthode `scriptStart()` pour créer un block capturant qui va être affiché dans une balise `<script>`. Les bouts de code de script capturés peuvent être affichés inline, ou mis en mémoire tampon dans un block :

```
// Ajoute dans le block 'script'.
$this->Html->scriptStart(['block' => true]);
echo "alert('Je suis dans le JavaScript');";
$this->Html->scriptEnd();
```

Une fois que vous avez mis en mémoire tampon le javascript, vous pouvez l'afficher comme vous le feriez pour tout autre *Block de vue* :

```
// Dans votre layout
echo $this->fetch('script');
```

Créer des Listes Imbriquées

`Cake\View\Helper\HtmlHelper::nestedList`(*array $list*, *array $options = []*, *array $itemOptions = []*)

Fabrique une liste imbriquée (UL/OL) dans un tableau associatif :

```
$list = [
    'Languages' => [
        'English' => [
            'American',
            'Canadian',
            'British',
        ],
        'Spanish',
        'German',
    ]
];
echo $this->Html->nestedList($list);
```

Affichera :

```
// Affichera (sans les espaces blancs)
<ul>
    <li>Languages
        <ul>
            <li>English
                <ul>
                    <li>American</li>
                    <li>Canadian</li>
                    <li>British</li>
                </ul>
            </li>
            <li>Spanish</li>
            <li>German</li>
        </ul>
    </li>
</ul>
```

Créer des En-Têtes de Tableaux

`Cake\View\Helper\HtmlHelper::tableHeaders`(*array $names*, *array $trOptions = null*, *array $thOptions = null*)

Crée une ligne de cellule d'en-tête à placer dans la balise <table> :

```
echo $this->Html->tableHeaders(['Date', 'Title', 'Active']);
```

// Affichera

```
<tr>
    <th>Date</th>
    <th>Title</th>
    <th>Active</th>
</tr>
```

```
echo $this->Html->tableHeaders(
    ['Date','Title','Active'],
    ['class' => 'status'],
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ['class' => 'product_table']
);
```

Affichera :

```
<tr class="status">
     <th class="product_table">Date</th>
     <th class="product_table">Title</th>
     <th class="product_table">Active</th>
</tr>
```

Vous pouvez définir des attributs par colonne, ceux-ci sont utilisés à la place de ceux par défaut dans `$thOptions` :

```
echo $this->Html->tableHeaders([
    'id',
    ['Name' => ['class' => 'highlight']],
    ['Date' => ['class' => 'sortable']]
]);
```

Sortie :

```
<tr>
    <th>id</th>
    <th class="highlight">Name</th>
    <th class="sortable">Date</th>
</tr>
```

Créer des Cellules de Tableaux

`Cake\View\Helper\HtmlHelper::tableCells`(*array $data*, *array $oddTrOptions = null*, *array $evenTrOptions = null*, *$useCount = false*, *$continueOddEven = true*)

Crée des cellules de table, en assignant aux lignes des attributs <tr> différents pour les lignes paires et les lignes impaires. Entoure une table simple de cellule dans un [] pour des attributs <td> spécifiques :

```
echo $this->Html->tableCells([
    ['Jul 7th, 2007', 'Best Brownies', 'Yes'],
    ['Jun 21st, 2007', 'Smart Cookies', 'Yes'],
    ['Aug 1st, 2006', 'Anti-Java Cake', 'No'],
]);
```

Sortie :

```
<tr><td>Jul 7th, 2007</td><td>Best Brownies</td><td>Yes</td></tr>
<tr><td>Jun 21st, 2007</td><td>Smart Cookies</td><td>Yes</td></tr>
<tr><td>Aug 1st, 2006</td><td>Anti-Java Cake</td><td>No</td></tr>
```

```
echo $this->Html->tableCells([
    ['Jul 7th, 2007', ['Best Brownies', ['class' => 'highlight']] , 'Yes'],
    ['Jun 21st, 2007', 'Smart Cookies', 'Yes'],
    ['Aug 1st, 2006', 'Anti-Java Cake', ['No', ['id' => 'special']]],
]);
```

// Sortie

```
<tr>
    <td>
        Jul 7th, 2007
    </td>
    <td class="highlight">
        Best Brownies
    </td>
    <td>
        Yes
    </td>
</tr>
<tr>
    <td>
        Jun 21st, 2007
    </td>
    <td>
        Smart Cookies
    </td>
    <td>
        Yes
    </td>
</tr>
<tr>
    <td>
        Aug 1st, 2006
    </td>
    <td>
        Anti-Java Cake
    </td>
    <td id="special">
        No
    </td>
</tr>
```

```
echo $this->Html->tableCells(
    [
        ['Red', 'Apple'],
        ['Orange', 'Orange'],
        ['Yellow', 'Banana'],
    ],
    ['class' => 'darker']
);
```

Affichera :

```
<tr class="darker"><td>Red</td><td>Apple</td></tr>
<tr><td>Orange</td><td>Orange</td></tr>
<tr class="darker"><td>Yellow</td><td>Banana</td></tr>
```

Changer l'affichage des balises avec le Helper Html

`Cake\View\Helper\HtmlHelper::setTemplates($templates)`

Le paramètre `setTemplates` peut être soit un chemin de fichier en chaîne de caractères vers le fichier PHP contenant les balises que vous souhaitez charger, soit avec un tableau des templates à ajouter/remplacer :

```
// Charger les templates à partir de config/my_html.php
$this->Html->setTemplates('my_html');

// Charger les templates spécifiques.
$this->Html->setTemplates([
    'javascriptlink' => '<script src="{{url}}" type="text/javascript"{{attrs}}></script>'
]);
```

Lors du chargement des fichiers de templates, votre fichier ressemblera à :

```
<?php
return [
    'javascriptlink' => '<script src="{{url}}" type="text/javascript"{{attrs}}></script>'
];
```

Avertissement

Les chaînes de template contenant un signe pourcentage (%) nécessitent une attention spéciale, vous devriez préfixer ce caractère avec un autre pourcentage pour qu'il ressemble à %%. La raison est que les templates sont compilés en interne pour être utilisé avec `sprintf()`. Exemple : `<div style="width:{{size}}%%">{{content}}</div>`

Création d'un chemin de navigation avec le Helper Html

`Cake\View\Helper\HtmlHelper::addCrumb(string $name, string $link = null, mixed $options = null)`

`Cake\View\Helper\HtmlHelper::getCrumbs(string $separator = '»', string $startText = false)`

`Cake\View\Helper\HtmlHelper::getCrumbList(array $options = [], $startText = false)`

Beaucoup d'applications utilisent un chemin de navigation (fil d'Ariane) pour faciliter la navigation des utilisateurs. Vous pouvez créer un chemin de navigation avec l'aide du HtmlHelper. Pour mettre cela en service, ajoutez cela dans votre template de layout :

```
echo $this->Html->getCrumbs(' > ', 'Home');
```

L'option `$startText` peut aussi accepter un tableau. Cela donne plus de contrôle à travers le premier lien généré :

```
echo $this->Html->getCrumbs(' > ', [
    'text' => $this->Html->image('home.png'),
    'url' => ['controller' => 'Pages', 'action' => 'display', 'home'],
    'escape' => false
]);
```

Toute clé qui n'est pas `text` ou `url` sera passée à *link()* comme paramètre `$options`.

Maintenant, dans votre vue vous allez devoir ajouter ce qui suit pour démarrer le fil d'Ariane sur chacune de vos pages :

```
$this->Html->addCrumb('Users', '/users');
$this->Html->addCrumb('Add User', ['controller' => 'Users', 'action' => 'add']);
```

Ceci ajoutera la sortie « **Home > Users > Add User** » dans votre layout où `getCrumbs` a été ajouté.

Vous pouvez aussi récupérer le fil d'Ariane en tant que liste Html :

```
echo $this->Html->getCrumbList();
```

Cette méthode utilise `Cake\View\Helper\HtmlHelper::tag()` pour générer la liste et ses éléments. Fonctionne de la même manière que *getCrumbs()*, il utilise toutes les options que chacun des fils a ajouté. Vous pouvez utiliser le paramètre `$startText` pour fournir le premier lien de fil. C'est utile quand vous voulez inclure un lien racine. Cette option fonctionne de la même façon que l'option `$startText` pour *getCrumbs()*.

En option vous pouvez préciser un attribut standard HTML valide pour un `<ul>` (Liste non ordonnées) comme `class` et pour des options spécifiques, vous avez : `separator` (sera entre les éléments `<li>`), `firstClass` et `lastClass` comme :

```
echo $this->Html->getCrumbList(
    [
        'firstClass' => false,
        'lastClass' => 'active',
        'class' => 'breadcrumb'
    ],
    'Home'
);
```

Cette méthode utilise `Cake\View\Helper\HtmlHelper::tag()` pour générer une liste et ses éléments. Fonctionne de la même manière que *getCrumbs()*, donc elle utilise des options pour lesquelles chaque crumb a été ajouté. Vous pouvez utiliser le paramètre `$startText` pour fournir le premier lien/texte breadcrumb. C'est utile quand vous voulez toujours inclure un lien avec la racine. Cette option fonctionne de la même manière que l'option `$startText` pour *getCrumbs()*.

Number

class `Cake\View\Helper\`**`NumberHelper`**(*View $view*, *array $config = []*)

Le NumberHelper contient des méthodes pratiques qui permettent l'affichage des nombres dans divers formats communs dans vos vues. Ces méthodes contiennent des moyens pour formater les devises, pourcentages, taille des données, le format des nombres avec précisions et aussi de vous donner davantage de souplesse en matière de formatage des nombres.

Toutes ces fonctions retournent le nombre formaté ; Elles n'affichent pas automatiquement la sortie dans la vue.

Formatage des Devises

`Cake\View\Helper\NumberHelper::`**`currency`**(*mixed $value*, *string $currency = null*, *array $options = []*)

Cette méthode est utilisée pour afficher un nombre dans des formats de monnaie courante (EUR,GBP,USD). L'utilisation dans une vue ressemble à ceci :

```
// Appelé avec NumberHelper
echo $this->Number->currency($value, $currency);

// Appelé avec Number
echo CakeNumber::currency($value, $currency);
```

Le premier paramètre `$value`, doit être un nombre décimal qui représente le montant d'argent que vous désirez. Le second paramètre est utilisé pour choisir un schéma de formatage de monnaie courante :

$currency	1234.56, formaté par le type courant
EUR	€1.234,56
GBP	£1,234.56
USD	$1,234.56

Le troisième paramètre est un tableau d'options pour définir la sortie. Les options suivantes sont disponibles :

Option	Description
before	Chaine de caractères à placer avant le nombre.
after	Chaine de caractères à placer après le nombre.
zero	Le texte à utiliser pour des valeurs à zéro, peut être une chaîne de caractères ou un nombre. ex : 0, "Free !"
places	Nombre de décimales à utiliser. ex : 2
precision	Nombre maximal de décimale à utiliser. ex :2
locale	Le nom de la locale utilisée pour formatter le nombre, ie. « fr_FR ».
fractionSymbol	Chaîne de caractères à utiliser pour les nombres en fraction. ex : " cents"
fractionPosition	Soit "before" soit "after" pour placer le symbole de fraction
pattern	Un modèle de formatage ICU à utiliser pour formatter le nombre. ex : #,###.00
useIntlCode	Mettre à `true` pour remplacer le symbole monétaire par le code monétaire international

Si la valeur de $currency est `null`, la devise par défaut est récupérée par *`Cake\I18n\Number::defaultCurrency()`*.

Paramétrage de la Devise par Défaut

`Cake\View\Helper\NumberHelper::defaultCurrency`(*string $currency*)

Setter/getter pour la monnaie par défaut. Ceci retire la nécessité de toujours passer la monnaie à *`Cake\I18n\Number::currency()`* et change toutes les sorties de monnaie en définissant les autres par défaut. Si `$currency` est `false`, cela effacera la valeur actuellement enregistrée. Par défaut, cette fonction retourne la valeur `intl.default_locale` si définie et "en_US" sinon.

Formatage Des Nombres A Virgules Flottantes

`Cake\View\Helper\NumberHelper::precision`(*float $value*, *int $precision = 3*, *array $options = []*)

Cette méthode affiche un nombre avec la précision spécifiée (place de la décimale). Elle arrondira afin de maintenir le niveau de précision défini :

```
// Appelé avec NumberHelper
echo $this->Number->precision(456.91873645, 2 );

// Sortie
456.92

// Appelé avec Number
echo Number::precision(456.91873645, 2 );
```

Formatage Des Pourcentages

`Cake\View\Helper\NumberHelper::toPercentage(mixed $value, int $precision = 2, array $options = [])`

Option	Description
multiply	Booléen pour indiquer si la valeur doit être multipliée par 100. Utile pour les pourcentages avec décimale.

Comme `Cake\I18n\Number::precision()`, cette méthode formate un nombre selon la précision fournie (où les nombres sont arrondis pour parvenir à ce degré de précision). Cette méthode exprime aussi le nombre en tant que pourcentage et ajoute un signe de pourcent à la sortie :

```
// appelé avec NumberHelper. Sortie: 45.69%
echo $this->Number->toPercentage(45.691873645);

// appelé avec Number. Sortie: 45.69%
echo Number::toPercentage(45.691873645);

// Appelé avec multiply. Sortie: 45.69%
echo Number::toPercentage(0.45691, 2, [
    'multiply' => true
]);
```

Interagir Avec Des Valeurs Lisibles Par L'Homme

`Cake\View\Helper\NumberHelper::toReadableSize(string $dataSize)`

Cette méthode formate les tailles de données dans des formes lisibles pour l'homme. Elle fournit une manière raccourcie de convertir les en KB, MB, GB, et TB. La taille est affichée avec un niveau de précision à deux chiffres, selon la taille de données fournie (ex : les tailles supérieurs sont exprimées dans des termes plus larges) :

```
// Appelé avec NumberHelper
echo $this->Number->toReadableSize(0); // 0 Byte
echo $this->Number->toReadableSize(1024); // 1 KB
echo $this->Number->toReadableSize(1321205.76); // 1.26 MB
echo $this->Number->toReadableSize(5368709120); // 5 GB

// Appelé avec Number
echo Number::toReadableSize(0); // 0 Byte
echo Number::toReadableSize(1024); // 1 KB
echo Number::toReadableSize(1321205.76); // 1.26 MB
echo Number::toReadableSize(5368709120); // 5 GB
```

Formatage Des Nombres

`Cake\View\Helper\NumberHelper::format(mixed $value, array $options=[])`

Cette méthode vous donne beaucoup plus de contrôle sur le formatage des nombres pour l'utilisation dans vos vues (et est utilisée en tant que méthode principale par la plupart des autres méthodes de NumberHelper). L'utilisation de cette méthode pourrait ressembler à cela :

```
// Appelé avec NumberHelper
$this->Number->format($value, $options);

// Appelé avec Number
Number::format($value, $options);
```

Le paramètre `$value` est le nombre que vous souhaitez formater pour la sortie. Avec aucun `$options` fourni, le nombre 1236.334 sortirait comme ceci : 1,236. Notez que la précision par défaut est d'aucun chiffre après la virgule.

Le paramètre `$options` est là où réside la réelle magie de cette méthode.

- Si vous passez un entier alors celui-ci devient le montant de précision pour la fonction.
- Si vous passez un tableau associatif, vous pouvez utiliser les clés suivantes :

Option	Description
places	Nombre de décimales à utiliser. ex : 2
precision	Nombre maximal de décimale à utiliser. ex :2
pattern	Un modèle de formatage ICU à utiliser pour formatter le nombre. ex : #,###.00
locale	Le nom de la locale utilisée pour formatter le nombre, ie. « fr_FR ».
before	Chaine de caractères à placer avant le nombre.
after	Chaine de caractères à placer après le nombre.

Exemple :

```
// Appelé avec NumberHelper
echo $this->Number->format('123456.7890', [
    'places' => 2,
    'before' => '¥ ',
    'after' => ' !'
]);
// Sortie '¥ 123,456.79 !'

echo $this->Number->format('123456.7890', [
    'locale' => 'fr_FR'
]);
// Sortie '123 456,79 !'

// Appelé avec Number
echo Number::format('123456.7890', [
    'places' => 2,
    'before' => '¥ ',
    'after' => ' !'
]);
// Sortie '¥ 123,456.79 !'

echo Number::format('123456.7890', [
    'locale' => 'fr_FR'
]);
// Sortie '123 456,79 !'
```

`Cake\View\Helper\NumberHelper::ordinal`(*mixed $value*, *array $options = []*)

Cette méthode va afficher un nombre ordinal.

Exemples :

```
echo Number::ordinal(1);
// Affiche '1st'

echo Number::ordinal(2);
// Affiche '2nd'

echo Number::ordinal(2, [
    'locale' => 'fr_FR'
]);
// Affiche '2e'

echo Number::ordinal(410);
// Affiche '410th'
```

Formatage Des Différences

`Cake\View\Helper\NumberHelper::formatDelta(mixed $value, mixed $options=[])`

Cette méthode affiche les différences en valeur comme un nombre signé :

```
// Appelé avec NumberHelper
$this->Number->formatDelta($value, $options);

// Appelé avec Number
Number::formatDelta($value, $options);
```

Le paramètre `$value` est le nombre que vous planifiez sur le formatage de sortie. Avec aucun `$options` fourni, le nombre 1236.334 sortirait 1,236. Notez que la valeur de precision par défaut est aucune décimale.

Le paramètre `$options` prend les mêmes clés que `Number::format()` lui-même :

Option	Description
places	Nombre de décimales à utiliser. ex : 2
precision	Nombre maximal de décimale à utiliser. ex :2
pattern	Un modèle de formatage ICU à utiliser pour formatter le nombre. ex : #,###.00
locale	Le nom de la locale utilisée pour formatter le nombre, ie. « fr_FR ».
before	Chaine de caractères à placer avant le nombre.
after	Chaine de caractères à placer après le nombre.

Example :

```
// Appelé avec NumberHelper
echo $this->Number->formatDelta('123456.7890', [
    'places' => 2,
    'before' => '[',
    'after' => ']'
]);
// Sortie '[+123,456.79]'

// Appelé avec Number
echo Number::formatDelta('123456.7890', [
    'places' => 2,
```

(suite sur la page suivante)

(suite de la page précédente)

```
    'before' => '[',
    'after' => ']'
]);
// Sortie '[+123,456.79]'
```

Avertissement

Les symboles sont en UTF-8.

Paginator

class Cake\View\Helper**PaginatorHelper**(*View $view*, *array $config = []*)

Le PaginatorHelper est utilisé pour présenter des contrôles de pagination comme les numéros de pages et les liens suivant/précédent. Il travaille en tamdem avec `PaginatorComponent`.

Voir aussi *Pagination* pour des informations sur la façon de créer des jeux de données paginés et faire des requêtes paginées.

Templates de PaginatorHelper

En interne, PaginatorHelper utilise une série simple de templates HTML pour générer les balises. Vous pouvez modifier ces templates pour personnaliser le HTML généré par PaginatorHelper.

Templates utilise des placeholders de style `{{var}}`. Il est important de ne pas ajouter d'espaces autour du `{{}}` ou les remplacements ne fonctionneront pas.

Charger les Templates à partir d'un Fichier

Lors de l'ajout de PaginatorHelper dans votre controller, vous pouvez définir la configuration de "templates" pour définir un fichier de template à charger. Cela vous permet de personnaliser plusieurs templates et de garder votre code DRY :

```
// Dans votre fichier AppView.php
public function initialize(): void
{
    ...
    $this->loadHelper('Paginator', ['templates' => 'paginator-templates']);
}
```

Cela va charger le fichier qui se trouve dans **config/paginator-templates.php**. Regardez l'exemple ci-dessous pour voir à quoi doit ressembler le fichier. Vous pouvez aussi charger les templates à partir d'un plugin en utilisant la *syntaxe de plugin* :

```
// Dans votre fichier AppView.php
public function initialize(): void
{
    ...
    $this->loadHelper('Paginator', ['templates' => 'MyPlugin.paginator-templates']);
}
```

Si vos templates sont dans l'application principale ou dans un plugin, vos fichiers de templates devraient ressembler à ceci :

```
return [
    'number' => '<a href="{{url}}">{{text}}</a>',
];
```

Changer les Templates à la Volée

`Cake\View\Helper\PaginatorHelper::setTemplates($templates)`

Cette méthode vous permet de changer les templates utilisés par PaginatorHelper à la volée. Ceci peut être utile quand vous voulez personnaliser des templates pour l'appel d'une méthode particulière :

```
// Lire la valeur du template actuel.
$result = $this->Paginator->getTemplates('number');

// Avant 3.4
$result = $this->Paginator->templates('number');

// Changez un template
$this->Paginator->templates([
    'number' => '<em><a href="{{url}}">{{text}}</a></em>'
]);
```

Avertissement

Les chaînes de template contenant un signe pourcentage (%) nécessitent une attention spéciale, vous devriez préfixer ce caractère avec un autre pourcentage pour qu'il ressemble à %%. La raison est que les templates sont compilés en interne pour être utilisés avec `sprintf()`. Exemple : "<div style= »width :{{size}}%% »>{{content}} "

Noms de Templates

PaginatorHelper utilise les templates suivants :

- `nextActive` L'état activé pour un lien généré par next().
- `nextDisabled` L'état désactivé pour next().
- `prevActive` L'état activé pour un lien généré par prev().
- `prevDisabled` L'état désactivé pour prev()
- `counterRange` Le template counter() utilisé quand format == range.
- `counterPages` The template counter() utilisé quand format == pages.
- `first` Le template utilisé pour un lien généré par first().
- `last` Le template utilisé pour un lien généré par last()
- `number` Le template utilisé pour un lien généré par numbers().
- `current` Le template utilisé pour la page courante.
- `ellipsis` Le template utilisé pour des ellipses générées par numbers().
- `sort` Le template pour un lien trié sans direction.
- `sortAsc` Le template pour un lien trié avec une direction ascendante.
- `sortDesc` Le template pour un lien trié avec une direction descendante.

Création de liens de tri

`Cake\View\Helper\PaginatorHelper::sort($key, $title = null, $options = [])`

Paramètres

- **$key** (`string`) – Le nom de la clé du jeu d'enregistrement qui doit être triée.

— **$title** (string) – Titre du lien. Si $title est null, $key sera utilisée pour le titre et sera générée par inflexion.
— **$options** (array) – Options pour le tri des liens.

Génère un lien de tri. Définit le nom ou les paramètres de la chaîne de recherche pour le tri et la direction. Les liens par défaut fourniront un tri ascendant. Après le premier clique, les liens générés avec `sort()` gèreront le changement de direction automatiquement. Les liens de tri par défaut ascendant. Si le jeu de résultat est trié en ascendant avec la clé spécifiée le liens retourné triera en "décroissant".

Les clés acceptées pour `$options` :
— `escape` Si vous voulez que le contenu soit encodé en HTML, `true` par défaut.
— `model` Le model à utiliser, par défaut à `PaginatorHelper::defaultModel()`.
— `direction` La direction par défaut à utiliser quand ce lien n'est pas actif.
— `lock` Verrouiller la direction. Va seulement utiliser la direction par défaut, par défaut à `false`.
En considérant que vous paginez des posts, qu'ils sont sur la page un :

```
echo $this->Paginator->sort('user_id');
```

Sortie :

```
<a href="/posts/index?page=1&amp;sort=user_id&amp;direction=asc">User Id</a>
```

Vous pouvez utiliser le paramètre title pour créer des textes personnalisés pour votre lien :

```
echo $this->Paginator->sort('user_id', 'User account');
```

Sortie :

```
<a href="/posts/index?page=1&amp;sort=user_id&amp;direction=asc">User account</a>
```

Si vous utilisez de l'HTML comme des images dans vos liens rappelez-vous de paramétrer l'échappement :

```
echo $this->Paginator->sort(
  'user_id',
  '<em>User account</em>',
  ['escape' => false]
);
```

Sortie :

```
<a href="/posts/index?page=1&amp;sort=user_id&amp;direction=asc"><em>User account</em></
↪a>
```

L'option de direction peut être utilisée pour paramétrer la direction par défaut pour un lien. Une fois qu'un lien est activé, il changera automatiquement de direction comme habituellement :

```
echo $this->Paginator->sort('user_id', null, ['direction' => 'desc']);
```

Sortie

```
<a href="/posts/index?page=1&amp;sort=user_id&amp;direction=desc">User Id</a>
```

L'option lock peut être utilisée pour verrouiller le tri dans la direction spécifiée :

```
echo $this->Paginator->sort('user_id', null, ['direction' => 'asc', 'lock' => true]);
```

`Cake\View\Helper\PaginatorHelper::sortDir(string $model = null, mixed $options = [])`

récupère la direction courante du tri du jeu d'enregistrement.

`Cake\View\Helper\PaginatorHelper::sortKey(string $model = null, mixed $options = [])`

récupère la clé courante selon laquelle le jeu d'enregistrement est trié.

Création des liens de page numérotés

`Cake\View\Helper\PaginatorHelper::numbers($options = [])`

Retourne un ensemble de nombres pour le jeu de résultat paginé. Utilise un modulo pour décider combien de nombres à présenter de chaque coté de la page courante. Par défaut 8 liens de chaque coté de la page courante seront créés si cette page existe. Les liens ne seront pas générés pour les pages qui n'existent pas. La page courante n'est pas un lien également.

Les options supportées sont :

- `before` Contenu a insérer avant les nombres.
- `after` Contenu a insérer après les nombres.
- `model` Model pour lequel créer des nombres, par défaut à `PaginatorHelper::defaultModel()`.
- `modulus` combien de nombres à inclure sur chacun des cotés de la page courante, par défaut à 8.
- `first` Si vous voulez que les premiers liens soit générés, définit à un entier pour définir le nombre de "premier" liens à générer. Par défaut à `false`. Si une chaîne est définie un lien pour la première page sera générée avec la valeur comme titre :

```
echo $this->Paginator->numbers(['first' => 'First page']);
```

- `last` Si vous voulez que les derniers liens soit générés, définit à un entier pour définir le nombre de "dernier" liens à générer. Par défaut à `false`. Suit la même logique que l'option `first`. il y a méthode *`last()`* à utiliser séparément si vous le voulez.

Bien que cette méthode permette beaucoup de personnalisation pour ses sorties, elle peut aussi être appelée sans aucun paramètre :

```
echo $this->Paginator->numbers();
```

En utilisant les options first et last vous pouvez créer des liens pour le début et la fin du jeu de page. Le code suivant pourrait créer un jeu de liens de page qui inclut les liens des deux premiers et deux derniers résultats de pages :

```
echo $this->Paginator->numbers(['first' => 2, 'last' => 2]);
```

Création de liens de sauts

En plus de générer des liens qui vont directement sur des numéros de pages spécifiques, vous voudrez souvent des liens qui amènent vers le lien précédent ou suivant, première et dernière pages dans le jeu de données paginées.

`Cake\View\Helper\PaginatorHelper::prev($title = '<< Previous', $options = [])`

Paramètres

- **$title** (`string`) – Titre du lien.
- **$options** (`mixed`) – Options pour le lien de pagination.

Génère un lien vers la page précédente dans un jeu d'enregistrements paginés.

`$options` supporte les clés suivantes :

- `escape` Si vous voulez que le contenu soit encodé en HTML, par défaut à `true`.
- `model` Le model à utiliser, par défaut `PaginatorHelper::defaultModel()`.
- `disabledTitle` Le texte à utiliser quand le lien est désactivé. Par défaut, la valeur du paramètre `$title`.

Un simple exemple serait :

```
echo $this->Paginator->prev(' << ' . __('previous'));
```

Si vous étiez actuellement sur la secondes pages des posts (articles), vous obtenez le résultat suivant :

```
<li class="prev">
    <a rel="prev" href="/posts/index?page=1&amp;sort=title&amp;order=desc">
        &lt;&lt; previous
    </a>
</li>
```

S'il n'y avait pas de page précédente vous obtenez :

```
<li class="prev disabled"><span>&lt;&lt; previous</span></li>
```

Pour changer les templates utilisés par cette méthode, regardez *Templates de PaginatorHelper*.

`Cake\View\Helper\PaginatorHelper::next`(*$title = 'Next >>'*, *$options = []*)

Cette méthode est identique a `prev()` avec quelques exceptions. il crée le lien pointant vers la page suivante au lieu de la précédente. elle utilise aussi `next` comme valeur d'attribut rel au lieu de `prev`.

`Cake\View\Helper\PaginatorHelper::first`(*$first = '<< first'*, *$options = []*)

Retourne une première ou un nombre pour les premières pages. Si une chaîne est fournie, alors un lien vers la première page avec le texte fourni sera créé :

```
echo $this->Paginator->first('< first');
```

Ceci crée un simple lien pour la première page. Ne retournera rien si vous êtes sur la première page. Vous pouvez aussi utiliser un nombre entier pour indiquer combien de premier liens paginés vous voulez générer :

```
echo $this->Paginator->first(3);
```

Ceci créera des liens pour les 3 premières pages, une fois la troisième page ou plus atteinte. Avant cela rien ne sera retourné.

Les paramètres d'option acceptent ce qui suit :
— `model` Le model à utiliser par défaut PaginatorHelper : :defaultModel().
— `escape` Si le contenu HTML doit être échappé ou pas. `true` par défaut.

`Cake\View\Helper\PaginatorHelper::last`(*$last = 'last >>'*, *$options = []*)

Cette méthode fonctionne très bien comme la méthode *`first()`*. Elle a quelques différences cependant. Elle ne générera pas de lien si vous êtes sur la dernière page avec la valeur chaîne `$last`. Pour une valeur entière de `$last` aucun lien ne sera généré une fois que l'utilisateur sera dans la zone des dernières pages.

Créer des Liens de Header

PaginatorHelper peut être utilisé pour créer des liens de pagination pour la balise `<head>` de votre page :

```
// Va créer des liens "précédent" / "suivant" pour le Model courant.
echo $this->Paginator->meta();

// Va créer des liens précédent / suivant et "premier" / "dernier"
// pour le Model courant.
echo $this->Paginator->meta(['first' => true, 'last' => true]);
```

Vérifier l'Etat de la Pagination

`Cake\View\Helper\PaginatorHelper::current(string $model = null)`

récupère la page actuelle pour le jeu d'enregistrement du model donné :

```
// Ou l'URL est: http://example.com/comments/view/page:3
echo $this->Paginator->current('Comment');
// la sortie est 3
```

`Cake\View\Helper\PaginatorHelper::hasNext(string $model = null)`

Retourne `true` si le résultat fourni n'est pas sur la dernière page.

`Cake\View\Helper\PaginatorHelper::hasPrev(string $model = null)`

Retourne `true` si le résultat fourni n'est pas sur la première page.

`Cake\View\Helper\PaginatorHelper::hasPage(string $model = null, integer $page = 1)`

Retourne `true` si l'ensemble de résultats fourni a le numéro de page fourni par `$page`.

`Cake\View\Helper\PaginatorHelper::total(string $model = null)`

Retourne le nombre total de pages pour le `model` passé en paramètre.

Création d'un compteur de page

`Cake\View\Helper\PaginatorHelper::counter($options = [])`

Retourne une chaîne compteur pour le jeu de résultat paginé. En Utilisant une chaîne formatée fournie et un nombre d'options vous pouvez créer des indicateurs et des éléments spécifiques de l'application indiquant ou l'utilisateur se trouve dans l'ensemble de données paginées.

Il y a un certain nombre d'options supportées pour `counter()`. celles supportées sont :

- `format` Format du compteur. Les formats supportés sont "range", "pages" et custom. Par défaut à pages qui pourrait ressortir comme "1 of 10". Dans le mode custom la chaîne fournie est analysée (parsée) et les jetons sont remplacées par des valeurs réelles. Les jetons autorisés sont :
  - `{{page}}` - la page courante affichée.
  - `{{pages}}` - le nombre total de pages.
  - `{{current}}` - le nombre actuel d'enregistrements affichés.
  - `{{count}}` - le nombre total d'enregistrements dans le jeu de résultat.
  - `{{start}}` - le nombre de premier enregistrement affichés.
  - `{{end}}` - le nombre de dernier enregistrements affichés.
  - `{{model}}` - La forme plurielle du nom de model. Si votre model était "RecettePage", `{{model}}` devrait être "recipe pages".

  Vous pouvez aussi fournir simplement une chaîne à la méthode counter en utilisant les jetons autorisés. Par exemple :

  ```
  echo $this->Paginator->counter(
      'Page {{page}} of {{pages}}, showing {{current}} records out of
       {{count}} total, starting on record {{start}}, ending on {{end}}'
  );
  ```

  En définissant "format" à "range" donnerait en sortie "1 - 3 of 13" :

  ```
  echo $this->Paginator->counter([
      'format' => 'range'
  ]);
  ```

- `model` Le nom du model en cours de pagination, par défaut à `PaginatorHelper::defaultModel()`. Ceci est utilisé en conjonction avec la chaîne personnalisée de l'option "format".

Générer des Url de Pagination

`Cake\View\Helper\PaginatorHelper::generateUrl(array $options = [], $model = null, $full = false)`

Retourne par défaut une chaine de l'URL de pagination complète pour utiliser dans contexte non-standard (ex. JavaScript) :

```
echo $this->Paginator->generateUrl(['sort' => 'title']);
```

Créer une Liste Déroulante de Limites

`Cake\View\Helper\PaginatorHelper::limitControl(array $limits = [], $default = null, array $options = [])`

Créer un `select` qui permet de changer le paramètre `limit` de la query :

```
// Utilise le défaut.
echo $this->Paginator->limitControl();

// Permet de définir les limites que vous souhaitez.
echo $this->Paginator->limitControl([25 => 25, 50 => 50]);

// Limites personnalisées et set l'option sélectionnée
echo $this->Paginator->limitControl([25 => 25, 50 => 50], $user->perPage);
```

Cela générera un `form` qui sera automatiquement soumis lors d'un changement de valeur sur le `select`.

Configurer les Options de Pagination

`Cake\View\Helper\PaginatorHelper::options($options = [])`

Définit toutes les options pour le PaginatorHelper Helper. Les options supportées sont :

- `url` L'URL de l'action de pagination. "url" comporte quelques sous options telles que :
  - `sort` La clé qui décrit la façon de trier les enregistrements.
  - `direction` La direction du tri. Par défaut à "ASC".
  - `page` Le numéro de page à afficher.

  Les options mentionnées ci-dessus peuvent être utilisées pour forcer des pages/directions particulières. Vous pouvez aussi ajouter des contenu d'URL supplémentaires dans toutes les URLs générées dans le helper :

  ```
  $this->Paginator->options([
      'url' => [
          'sort' => 'email',
          'direction' => 'desc',
          'page' => 6,
          'lang' => 'en'
      ]
  ]);
  ```

  Ce qui se trouve ci-dessus ajoutera `en` comme paramètre de route pour chacun des liens que le helper va générer. Il créera également des liens avec des tris, direction et valeurs de page spécifiques. Par défaut PaginatorHelper fusionnera cela dans tous les paramètres passés et nommés. Ainsi vous n'aurez pas à le faire dans chacun des fichiers de vue.
- `escape` Définit si le HTMl du champ titre des liens doit être échappé. Par défaut à `true`.
- `model` Le nom du model en cours de pagination, par défaut à `PaginatorHelper::defaultModel()`.

Exemple d'Utilisation

C'est à vous de décider comment afficher les enregistrements à l'utilisateur, mais la plupart des fois, ce sera fait à l'intérieur des tables HTML. L'exemple ci-dessous suppose une présentation tabulaire, mais le PaginatorHelper disponible dans les vues n'a pas toujours besoin d'être limité en tant que tel.

Voir les détails sur PaginatorHelper [130] dans l" API. Comme mentionné précédemment, le PaginatorHelper offre également des fonctionnalités de tri qui peuvent être intégrées dans vos en-têtes de colonne de table :

```
<!-- templates/Posts/index.php -->
<table>
    <tr>
        <th><?= $this->Paginator->sort('id', 'ID') ?></th>
        <th><?= $this->Paginator->sort('title', 'Title') ?></th>
    </tr>
       <?php foreach ($recipes as $recipe): ?>
    <tr>
        <td><?= $recipe->id ?> </td>
        <td><?= h($recipe->title) ?> </td>
    </tr>
    <?php endforeach; ?>
</table>
```

Les liens en retour de la méthode `sort()` du `PaginatorHelper` permettent aux utilisateurs de cliquer sur les entêtes de table pour faire basculer l'ordre de tri des données d'un champ donné.

Il est aussi possible de trier une colonne basée sur des associations :

```
<table>
    <tr>
        <th><?= $this->Paginator->sort('title', 'Title') ?></th>
        <th><?= $this->Paginator->sort('Authors.name', 'Author') ?></th>
    </tr>
       <?php foreach ($recipes as $recipe): ?>
    <tr>
        <td><?= h($recipe->title) ?> </td>
        <td><?= h($recipe->name) ?> </td>
    </tr>
    <?php endforeach; ?>
</table>
```

L'ingrédient final pour l'affichage de la pagination dans les vues est l'addition de pages de navigation, aussi fournies par le Helper de Pagination :

```
// Montre les numéros de page
<?= $this->Paginator->numbers() ?>

// Montre les liens précédent et suivant
<?= $this->Paginator->prev('« Previous') ?>
<?= $this->Paginator->next('Next »') ?>

// affiche X et Y, ou X est la page courante et Y est le nombre de pages
<?= $this->Paginator->counter() ?>
```

130. https://api.cakephp.org/4.x/class-Cake.View.Helper.PaginatorHelper.html

Le texte de sortie de la méthode counter() peut également être personnalisé en utilisant des marqueurs spéciaux :

```
<?= $this->Paginator->counter([
    'format' => 'Page {{page}} of {{pages}}, showing {{current}} records out of
            {{count}} total, starting on record {{start}}, ending on {{end}}'
]) ?>
```

Paginer Plusieurs Résultats

Si vous *faîtes des requêtes de pagination multiple* vous devrez définir l'option `model` quand vous générez les éléments de la pagination. Vous pouvez soit utiliser l'option `model` sur chaque appel de méthode que vous faîtes au `PaginatorHelper`, soit utiliser `options()` pour définir le model par défaut :

```
// Passe l'option model
echo $this->Paginator->sort('title', ['model' => 'Articles']);

// Définit le model par défaut.
$this->Paginator->options(['defaultModel' => 'Articles']);
echo $this->Paginator->sort('title');
```

En utilisant l'option `model`, `PaginatorHelper` va automatiquement utiliser le `scope` défini quand la reqûete a été paginée.

Text

class `Cake\View\Helper\`**`TextHelper`**(*View $view*, *array $config = []*)

TextHelper possède des méthodes pour rendre le texte plus utilisable et sympa dans vos vues. Il aide à activer les liens, à formater les URLs, à créer des extraits de texte autour des mots ou des phrases choisies, mettant en évidence des mots clés dans des blocs de texte et tronquer élégamment de longues étendues de texte.

Lier les Adresses Email

`Cake\View\Helper\TextHelper::`**`autoLinkEmails`**(*string $text*, *array $options = []*)

Ajoute les liens aux adresses email bien formées dans $text, selon toutes les options définies dans `$options` (regardez *`HtmlHelper::link()`*) :

```
$myText = 'Pour plus d'informations sur nos pâtes et desserts fameux,
    contactez info@example.com';
$linkedText = $this->Text->autoLinkEmails($myText);
```

Sortie :

```
Pour plus d'informations sur nos pâtes et desserts fameux,
contactez <a href="mailto:info@example.com">info@example.com</a>
```

Cette méthode échappe automatiquement ces inputs. Utilisez l'option `escape` pour la désactiver si nécessaire.

Lier les URLs

`Cake\View\Helper\TextHelper::`**`autoLinkUrls`**(*string $text*, *array $options = []*)

De même que dans `autoLinkEmails()`, seule cette méthode cherche les chaînes de caractères qui commence par https, http, ftp, ou nntp et les liens de manière appropriée.

Cette méthode échappe automatiquement son input. Utilisez l'option `escape` pour la désactiver si nécessaire.

Lier à la fois les URLs et les Adresses Email

`Cake\View\Helper\TextHelper::autoLink(string $text, array $options = [])`

Exécute la fonctionnalité dans les deux `autoLinkUrls()` et `autoLinkEmails()` sur le `$text` fourni. Tous les URLs et emails sont liés de manière appropriée donnée par `$options` fourni.

Cette méthode échappe automatiquement son input. Utilisez l'option `escape` pour la désactiver si nécessaire.

Convertir du Texte en Paragraphes

`Cake\View\Helper\TextHelper::autoParagraph(string $text)`

Ajoute <p> autour du texte où un double retour ligne est trouvé, et <br> quand un simple retour ligne est rencontré :

```
$myText = 'For more information
regarding our world-famous pastries and desserts.

contact info@example.com';
$formattedText = $this->Text->autoParagraph($myText);
```

Output :

```
<p>Pour plus d\'information<br />
selon nos célèbres pâtes et desserts.</p>
<p>contact info@example.com</p>
```

Subrillance de Sous-Chaîne

`Cake\View\Helper\TextHelper::highlight(string $haystack, string $needle, array $options = [])`

Mettre en avant `$needle` dans `$haystack` en utilisant la chaîne spécifique `$options['format']` ou une chaîne par défaut.

Options :
- `format` - chaîne la partie de html avec laquelle la phrase sera mise en exergue.
- `html` - booléen Si `true`, va ignorer tous les tags HTML, s'assurant que seul le bon texte est mise en avant.

Exemple :

```
// appelé avec TextHelper
echo $this->Text->highlight(
    $lastSentence,
    'using',
    ['format' => '<span class="highlight">\1</span>']
);

// appelé avec Text
use Cake\Utility\Text;

echo Text::highlight(
    $lastSentence,
    'using',
    ['format' => '<span class="highlight">\1</span>']
);
```

Sortie :

```
Highlights $needle in $haystack <span class="highlight">using</span> the
$options['format'] string specified or a default string.
```

Retirer les Liens

`Cake\View\Helper\TextHelper::stripLinks($text)`

Enlève le `$text` fourni de tout lien HTML.

Tronquer le Texte

`Cake\View\Helper\TextHelper::truncate(string $text, int $length = 100, array $options)`

Si `$text` est plus long que `$length`, cette méthode le tronque à la longueur `$length` et ajoute un suffixe `'ellipsis'`, si défini. Si `'exact'` est passé à `false`, le truchement va se faire au premier espace après le point où `$length` a dépassé. Si `'html'` est passé à `true`, les balises html seront respectés et ne seront pas coupés.

`$options` est utilisé pour passer tous les paramètres supplémentaires, et a les clés suivantes possibles par défaut, celles-ci étant toutes optionnelles :

```
[
    'ellipsis' => '...',
    'exact' => true,
    'html' => false
]
```

Exemple :

```
// appelé avec TextHelper
echo $this->Text->truncate(
    'The killer crept forward and tripped on the rug.',
    22,
    [
        'ellipsis' => '...',
        'exact' => false
    ]
);

// appelé avec Text
App::uses('Text', 'Utility');
echo Text::truncate(
    'The killer crept forward and tripped on the rug.',
    22,
    [
        'ellipsis' => '...',
        'exact' => false
    ]
);
```

Sortie :

```
The killer crept...
```

Tronquer une chaîne par la fin

`Cake\View\Helper\TextHelper::tail(string $text, int $length = 100, array $options)`

Si `$text` est plus long que `$length`, cette méthode retire une sous-chaîne initiale avec la longueur de la différence et ajoute un préfixe `'ellipsis'`, s'il est défini. Si `'exact'` est passé à `false`, le truchement va se faire au premier espace avant le moment où le truchement aurait été fait.

`$options` est utilisé pour passer tous les paramètres supplémentaires, et a les clés possibles suivantes par défaut, toutes sont optionnelles :

```
[
    'ellipsis' => '...',
    'exact' => true
]
```

Exemple :

```
$sampleText = 'I packed my bag and in it I put a PSP, a PS3, a TV, ' .
    'a C# program that can divide by zero, death metal t-shirts'

// appelé avec TextHelper
echo $this->Text->tail(
    $sampleText,
    70,
    [
        'ellipsis' => '...',
        'exact' => false
    ]
);

// appelé avec Text
App::uses('Text', 'Utility');
echo Text::tail(
    $sampleText,
    70,
    [
        'ellipsis' => '...',
        'exact' => false
    ]
);
```

Sortie :

```
...a TV, a C# program that can divide by zero, death metal t-shirts
```

Générer un Extrait

`Cake\View\Helper\TextHelper::excerpt(string $haystack, string $needle, integer $radius=100, string $ellipsis="...")`

Génère un extrait de `$haystack` entourant le `$needle` avec un nombre de caractères de chaque côté déterminé par `$radius`, et préfixé/suffixé avec `$ellipsis`. Cette méthode est spécialement pratique pour les résultats de recherches. La chaîne requêtée ou les mots clés peuvent être montrés dans le document résultant :

```
// appelé avec TextHelper
echo $this->Text->excerpt($lastParagraph, 'method', 50, '...');

// appelé avec Text
use Cake\Utility\Text;

echo Text::excerpt($lastParagraph, 'méthode', 50, '...');
```

Génère : :

…$radius,et préfixé/suffixé avec $ellipsis. Cette méthode est spécialement pratique pour les résultats de r…

Convertir un tableau sous la forme d'une phrase

`Cake\View\Helper\TextHelper::toList(array $list, $and='and', $separator=', ')`

Crée une liste séparée avec des virgules, où les deux derniers items sont joins avec "and" :

```
$colors = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet'];

// appelé avec TextHelper
echo $this->Text->toList($colors);

// appelé avec Text
use Cake\Utility\Text;

echo Text::toList($colors);
```

Sortie :

```
red, orange, yellow, green, blue, indigo et violet
```

Time

`class Cake\View\Helper\TimeHelper(View $view, array $config = [])`

Le TimeHelper vous permet, comme il l'indique de gagner du temps. Il permet le traitement rapide des informations se rapportant au temps. Le Helper Time a deux tâches principales qu'il peut accomplir :

1. Il peut formater les chaînes de temps.
2. Il peut tester le temps.

Utiliser le Helper

Une utilisation courante de Time Helper est de compenser la date et le time pour correspondre au time zone de l'utilisateur. Utilisons un exemple de forum. Votre forum a plusieurs utilisateurs qui peuvent poster des messages depuis n'importe quelle partie du monde. Une façon de gérer le temps est de sauvegarder toutes les dates et les times à GMT+0 or UTC. Décommenter la ligne `date_default_timezone_set('UTC');` dans **config/bootstrap.php** pour s'assurer que le time zone de votre application est défini à GMT+0.

Ensuite, ajoutez un time zone à votre table users et faîtes les modifications nécessaires pour permettre à vos utilisateurs de définir leur time zone. Maintenant que nous connaissons le time zone de l'utilisateur connecté, nous pouvons corriger la date et le temps de nos posts en utilisant le TimeHelper :

```
echo $this->Time->format(
  $post->created,
  \IntlDateFormatter::FULL,
  false,
  $user->time_zone
);
// Affichera 'Saturday, August 22, 2011 at 11:53:00 PM GMT'
// pour un utilisateur dans GMT+0. Cela affichera,
// 'Saturday, August 22, 2011 at 03:53 PM GMT-8:00'
// pour un utilisateur dans GMT-8
```

La plupart des fonctionnalités de TimeHelper sont des interfaces rétro-compatibles pour les applications qui sont mises à jour à partir des versions anciennes de CakePHP. Comme l'ORM retourne des instances `Cake\I18n\Time` pour chaque colonne `timestamp` et `datetime`, vous pouvez utiliser les méthodes ici pour faire la plupart des tâches. Par exemple, pour en apprendre plus sur les chaines de formatage, jetez un oeil à la méthode Cake\I18n\Time::i18nFormat() [131].

Url

class Cake\View\Helper**UrlHelper**(*View $view*, *array $config = []*)

UrlHelper vous permet de générer des liens pour vos autres Helpers. C'est aussi un endroit unique pour personnaliser la façon dont les URLs sont générées en surchargeant le helper du cœur avec celui d'une application. Regardez la section *Faire des Alias de Helpers* pour voir comment faire.

Générer des URLs

Cake\View\Helper\UrlHelper::**build**(*mixed $url = null*, *boolean|array $full = false*)

Cette méthode retourne une URL pointant vers la combinaison du controller et de l'action. Si `$url` est vide, elle retourne `REQUEST_URI`, dans les autre cas, elle génère le lien utilisant le controller et l'action. Si `fullBase` vaut `true`, le lien fourni contiendra le chemin complet menant à la page :

```
echo $this->Url->build([
    'controller' => 'Posts',
    'action' => 'view',
    'bar'
]);

// Affiche
/posts/view/bar
```

D'autres exemples d'utilisation :

URL avec une extension :

```
echo $this->Url->build([
    'controller' => 'Posts',
    'action' => 'list',
    '_ext' => 'rss'
]);

// Affiche
/posts/list.rss
```

131. https://api.cakephp.org/4.x/class-Cake.I18n.Time.html#i18nFormat()

URL (commençant par "/") avec le chemin complet :

```
echo $this->Url->build('/posts', ['fullBase' => true]);

// Affiche
http://somedomain.com/posts
```

URL avec des paramètres GET et des ancres :

```
echo $this->Url->build([
    'controller' => 'Posts',
    'action' => 'search',
    '?' => ['foo'=> 'bar'],
    '#' => 'first'
]);

// Affiche
/posts/search?foo=bar#first
```

L'exemple du dessus utilise la clé ? qui est utile quand vous voulez être explicite sur les paramètres query string que vous utilisez ou si vous voulez un query string qui partage un nom avec un de vos placeholders de route.

URL utilisant une route labellisée :

```
echo $this->Url->build(['_name' => 'produit-page', 'slug' => 'i-m-slug']);

// Il faut que la route soit configurée comme suit:
// $router->connect(
//     '/produits/:slug',
//     [
//         'controller' => 'Produits',
//         'action' => 'view'
//     ],
//     [
//         '_name' => 'produit-page'
//     ]
// );
/produits/i-m-slug
```

Le deuxième paramètre vous permet de définir l'option qui contrôle l'échappement du HTML et si vous souhaitez ou non que le chemin de base soit ajouté :

```
$this->Url->build('/posts', [
    'escape' => false,
    'fullBase' => true
]);
```

Si vous générez des URLs pour du CSS, du Javascript ou des fichiers image, il existe des méthodes d'helper pour chacun de ces types d'assets :

```
// Affiche /img/icon.png
$this->Url->image('icon.png');

// Affiche /js/app.js
$this->Url->script('app.js');
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Affiche /css/app.css
$this->Url->css('app.css');
```

Pour de plus amples informations, voir Router::url [132] dans l'API.

Configurer les Helpers

Dans CakePHP, vous chargez les helpers en les déclarant dans une classe view. Une classe `AppView` est fournie avec chaque application CakePHP et est le lieu idéal pour charger les helpers :

```
class AppView extends View
{
    public function initialize(): void
    {
        parent::initialize();
        $this->loadHelper('Html');
        $this->loadHelper('Form');
        $this->loadHelper('Flash');
    }
}
```

Le chargement des helpers depuis les plugins utilise la *syntaxe de plugin* utilisée partout ailleurs dans CakePHP :

```
$this->loadHelper('Blog.Comment');
```

Vous n'avez pas à charger explicitement les helpers fournis par CakePHP ou de votre application. Ces helpers seront chargés paresseusement (lazily) au moment de la première utilisation. Par exemple :

```
// Charge le FormHelper s'il n'a pas été chargé précédemment.
$this->Form->create($article);
```

A partir d'une vue de plugin, les helpers de plugin peuvent également être chargés paresseusement. Par exemple, les templates de vues dans le plugin "Blog", peuvent charger paresseusement les helpers provenant du même plugin.

Chargement Conditionnel des Helpers

Vous pouvez utiliser le nom de l'action courante pour charger conditionnellement des helpers :

```
class AppView extends View
{
    public function initialize(): void
    {
        parent::initialize();
        if ($this->request->getParam('action') === 'index') {
            $this->loadHelper('ListPage');
        }
    }
}
```

Vous pouvez également utiliser la méthode `beforeRender()` de vos controllers pour charger des helpers :

132. https://api.cakephp.org/4.x/class-Cake.Routing.Router.html#_url

```
class ArticlesController extends AppController
{
    public function beforeRender(Event $event)
    {
        parent::beforeRender($event);
        $this->viewBuilder()->helpers(['MyHelper']);
    }
}
```

Options de Configuration

Vous pouvez passer des options de configuration dans les helpers. Ces options peuvent être utilisées pour définir les valeurs d'attributs ou modifier le comportement du helper :

```
namespace App\View\Helper;

use Cake\View\Helper;
use Cake\View\View;

class AwesomeHelper extends Helper
{
    public function initialize(array $config): void
    {
        debug($config);
    }
}

class AwesomeController extends AppController
{
    public $helpers = ['Awesome' => ['option1' => 'value1']];
}
```

Les options peuvent être spécifiées lors de la déclaration des helpers dans le controller comme montré ci-dessous :

```
namespace App\Controller;

use App\Controller\AppController;

class AwesomeController extends AppController
{
    public $helpers = ['Awesome' => ['option1' => 'value1']];
}
```

Par défaut, toutes les options de configuration sont fusionnées avec la propriété `$_defaultConfig`. Cette propriété doit définir les valeurs par défaut de toute configuration dont votre helper a besoin. Par exemple :

```
namespace App\View\Helper;

use Cake\View\Helper;
use Cake\View\StringTemplateTrait;

class AwesomeHelper extends Helper
{
```

(suite sur la page suivante)

(suite de la page précédente)

```
    use StringTemplateTrait;

    protected $_defaultConfig = [
        'templates' => [
            'label' => '<label for="{{for}}">{{content}}</label>',
        ],
    ];
}
```

Toute configuration fournie au constructeur de votre helper sera fusionnée avec les valeurs par défaut pendant la construction et les données fusionnées seront définies à `_config`. Vous pouvez utiliser la méthode `config()` pour lire la configuration actuelle :

```
// Lit l'option de config autoSetCustomValidity .
$class = $this->Awesome->config('autoSetCustomValidity ');
```

L'utilisation de la configuration du helper vous permet de configurer de manière déclarative vos helpers et de garder la logique de configuration en dehors des actions de votre controller. Si vous avez des options de configuration qui ne peuvent pas être inclues comme une partie de la classe de déclaration, vous pouvez les définir dans le callback `beforeRender()` de votre controller :

```
class PostsController extends AppController
{
    public function beforeRender(Event $event)
    {
        parent::beforeRender($event);
        $builder = $this->viewBuilder();
        $builder->helpers([
            'CustomStuff' => $this->_getCustomStuffConfig()
        ]);
    }
}
```

Faire des Alias de Helpers

Une configuration habituelle à utiliser est l'option `className`, qui vous permet de créer des alias de helpers dans vos vues. Cette fonctionnalité est utile quand vous voulez remplacer `$this->Html` ou une autre référence du Helper habituel avec une implémentation personnalisée :

```
// src/View/AppView.php
class AppView extends View
{
    public function initialize(): void
    {
        $this->loadHelper('Html', [
            'className' => 'MyHtml'
        ]);
    }
}

// src/View/Helper/MyHtmlHelper.php
```

(suite sur la page suivante)

(suite de la page précédente)

```
namespace App\View\Helper;

use Cake\View\Helper\HtmlHelper;

class MyHtmlHelper extends HtmlHelper
{
    // Ajout de code pour surcharger le HtmlHelper du cœur
}
```

Ce qui est au-dessus va faire un *alias* de `MyHtmlHelper` vers `$this->Html` dans vos vues.

> **Note**
>
> Faire un alias remplace cette instance partout où le helper est utilisé, ainsi que dans les autres Helpers.

Utiliser les Helpers

Une fois que vous avez configuré les helpers que vous souhaitiez utiliser, dans votre controller, chaque helper est exposé en propriété publique dans la vue. Par exemple, si vous utilisiez `HtmlHelper`, vous serez capable d'y accéder en faisant ce qui suit :

```
echo $this->Html->css('styles');
```

Ce qui est au-dessus appellera la méthode `css` du HtmlHelper. Vous pouvez accéder à n'importe quel helper chargé en utilisant `$this->{$helperName}`.

Charger les Helpers à la Volée

Il peut y avoir des cas où vous aurez besoin de charger dynamiquement un helper depuis l'intérieur d'une vue. Pour cela, vous pouvez utiliser le `Cake\View\HelperRegistry` :

```
// Les deux solutions fonctionnent.
$mediaHelper = $this->loadHelper('Media', $mediaConfig);
$mediaHelper = $this->helpers()->load('Media', $mediaConfig);
```

Le HelperCollection est une *registry* et supporte l'API collection utilisée partout ailleurs dans CakePHP.

Méthodes de Callback

Les Helpers disposent de plusieurs callbacks qui vous permettent d'augmenter le processus de rendu de vue. Allez voir la documentation de *Classe Helper* et *Événements système* pour plus d'informations.

Créer des Helpers

Vous pouvez créer des classes de helper personnalisées pour les utiliser dans votre application ou dans vos plugins. Comme la plupart des components de CakePHP, les classes de helper ont quelques conventions :

- Les fichiers de classe Helper doivent être placés dans **src/View/Helper**. Par exemple : **src/View/Helper/LinkHelper.php**
- Les classes Helper doivent être suffixées avec `Helper`. Par exemple : `LinkHelper`.
- Quand vous référencez les noms de classe helper, vous devez omettre le suffixe `Helper`. Par exemple : `$this->loadHelper('Link');`.

Vous devrez étendre `Helper` pour vous assurer que les choses fonctionnent correctement :

```
/* src/View/Helper/LinkHelper.php */
namespace App\View\Helper;

use Cake\View\Helper;

class LinkHelper extends Helper
{
    public function makeEdit($title, $url)
    {
        // La logique pour créer le lien spécialement formaté se place ici
    }
}
```

Inclure d'autres Helpers

Vous souhaitez peut-être utiliser quelques fonctionnalités déjà existantes dans un autre helper. Pour faire cela, vous pouvez spécifier les helpers que vous souhaitez utiliser avec un tableau `$helpers`, formaté comme vous le feriez dans un controller :

```
/* src/View/Helper/LinkHelper.php (utilisant d'autres helpers) */

namespace App\View\Helper;

use Cake\View\Helper;

class LinkHelper extends Helper
{
    public $helpers = ['Html'];

    public function makeEdit($title, $url)
    {
        // Utilise le Helper Html pour afficher la sortie
        // des données formatées:

        $link = $this->Html->link($title, $url, ['class' => 'edit']);

        return '<div class="editOuter">' . $link . '</div>';
    }
}
```

Utiliser votre Helper

Une fois que vous avez créé votre helper et l'avez placé dans **src/View/Helper/**, vous pouvez le charger dans vos vues :

```
class AppView extends View
{
    public function initialize(): void
    {
        parent::initialize();
        $this->loadHelper('Link');
    }
}
```

Une fois que votre helper est chargé, vous pouvez l'utiliser dans vos vues en accédant à l'attribut de vue correspondant :

```
<!-- fait un lien en utilisant le nouveau helper -->
<?= $this->Link->makeEdit('Changez cette Recette', '/recipes/edit/5') ?>
```

> **Note**
>
> `HelperRegistry` va tenter de charger automatiquement les helpers qui ne sont pas spécifiquement identifiés dans votre `Controller`.

Accéder aux variables de la View dans votre Helper

Si vous voulez accéder à une variable de la View dans votre helper, vous pouvez utiliser `$this->_View->viewVars`, comme illustré ci-dessous :

```
class AwesomeHelper extends Helper
{

    public $helpers = ['Html'];

    public someMethod()
    {
        // Définit la meta description
        echo $this->Html->meta(
            'description', $this->_View->viewVars['metaDescription'], ['block' => 'meta']
        );
    }
}
```

Rendre un Element de Vue dans votre Helper

Si vous souhaiter rendre un Element dans votre Helper, vous pouvez utiliser **$this->_View->element()** comme ceci :

```
class AwesomeHelper extends Helper
{
    public someFunction()
    {
        // Affiche directement dans votre helper
        echo $this->_View->element('/path/to/element',['foo'=>'bar','bar'=>'foo']);

        // ou le retourne dans votre vue
        return $this->_View->element('/path/to/element',['foo'=>'bar','bar'=>'foo']);
    }
}
```

Classe Helper

class Helper

Callbacks

En implémentant une méthode de callback dans un helper, CakePHP va automatiquement inscrire votre helper à l'évènement correspondant. A la différence des versions précédentes de CakePHP, vous *ne* devriez *pas* appeler `parent` dans vos callbacks, puisque la classe Helper de base n'implémente aucune des méthodes de callback.

`Helper::beforeRenderFile(Event $event, $viewFile)`

Est appelé avant que tout fichier de vue soit rendu. Cela inclut les elements, les vues, les vues parentes et les layouts.

`Helper::afterRenderFile(Event $event, $viewFile, $content)`

Est appelé après que tout fichier de vue est rendu. Cela inclut les elements, les vues, les vues parentes et les layouts. Un callback peut modifier et retourner `$content` pour changer la manière dont le contenu rendu est affiché dans le navigateur.

`Helper::beforeRender(Event $event, $viewFile)`

La méthode `beforeRender()` est appelée après la méthode `beforeRender()` du controller, mais avant les rendus du controller de la vue et du layout. Reçoit le fichier à rendre en argument.

`Helper::afterRender(Event $event, $viewFile)`

Est appelé après que la vue est rendu, mais avant que le rendu du layout ait commencé.

`Helper::beforeLayout(Event $event, $layoutFile)`

Est appelé avant que le rendu du layout commence. Reçoit le nom du fichier layout en argument.

`Helper::afterLayout(Event $event, $layoutFile)`

Est appelée après que le rendu du layout est fini. Reçoit le nom du fichier layout en argument.

CHAPITRE 12

Accès Base de Données & ORM

La manipulation de données de la base de donnés dans CakePHP est réalisée avec deux types d'objets principaux. Les premiers sont des **repositories** ou **table objects**. Ces objets fournissent un accès aux collections de données. Ils vous permettent de sauvegarder de nouveaux enregistrements, de modifier/supprimer des enregistrements existants, définir des relations et d'effectuer des opérations en masse. Les seconds types d'objets sont les **entities**. Les Entities représentent des enregistrements individuels et vous permettent de définir des comportements et des fonctionnalités au niveau des lignes/enregistrements.

Ces deux classes sont habituellement responsables de la gestion de presque tout ce qui concerne les données, leur validité, les interactions et l'évolution du flux d'informations dans votre domaine de travail.

L'ORM intégré dans CakePHP se spécialise dans les bases de données relationnelles, mais peut être étendu pour supporter des sources de données alternatives.

L'ORM de CakePHP emprunte des idées et concepts à la fois aux patterns de ActiveRecord et de Datamapper. Il a pour objectif de créer une implémentation hybride qui combine les aspects des deux patterns pour créer un ORM rapide et facile d'utilisation.

Avant de commencer à explorer l'ORM, assurez-vous de *configurer votre connexion à la base de données*.

Exemple Rapide

Pour commencer, vous n'avez à écrire aucun code. Si vous suivez les *conventions de CakePHP pour vos tables de base de données*, vous pouvez simplement commencer à utiliser l'ORM. Par exemple si vous voulez charger des données de la table `articles`, il suffit de créer votre table de classe `Articles`. Créez le fichier **src/Model/Table/ArticlesTable.php** avec le code suivant :

```
<?php
namespace App\Model\Table;

use Cake\ORM\Table;
```

(suite sur la page suivante)

(suite de la page précédente)

```
class ArticlesTable extends Table
{
}
```

Ensuite, dans un controller ou une command CakePHP pourra créer une instance pour nous :

```
public function uneMethode()
{
    $resultset = $this->fetchTable('Articles')->find()->all();

    foreach ($resultset as $row) {
        echo $row->title;
    }
}
```

Dans d'autres contextes, vous pouvez utiliser `LocatorAwareTrait`, qui ajoute des accesseurs pour les tables de l'ORM :

```
use Cake\ORM\Locator\LocatorAwareTrait;

public function uneMethode()
{
    $articles = $this->getTableLocator()->get('Articles');
    // le reste du code.
}
```

Depuis une méthode statique, vous pouvez utiliser `FactoryLocator` pour obtenir le locator de la table :

```
$articles = TableRegistry::getTableLocator()->get('Articles');
```

Les classes de tables représentent des **collections** ou **entities**. À présent, créons une classe d'entity pour nos Articles. Les classes Entity vous permettent de définir des accesseurs et mutateurs, une logique personnalisé pour les enregistrements pris individuellement, et bien d'autres choses. Commençons par ajouter ceci à **src/Model/Entity/Article.php** après la balise d'ouverture `<?php` :

```
namespace App\Model\Entity;

use Cake\ORM\Entity;

class Article extends Entity
{
}
```

Les Entities utilisent la version CamelCase du nom de la table comme nom de classe par défaut. Maintenant que nous avons créé notre classe entity, quand nous chargeons les entities de la base de données, nous obtenons les instances de notre nouvelle classe Article :

```
use Cake\ORM\Locator\LocatorAwareTrait;

$articles = $this->getTableLocator()->get('Articles');
$resultset = $articles->find()->all();

foreach ($resultset as $row) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
    // Chaque ligne est maintenant une instance de notre classe Article.
    echo $row->title;
}
```

CakePHP utilise des conventions de nommage pour relier les classes Table et Entity. Si vous avez besoin de personnaliser l'entity qu'une table utilise, vous pouvez utiliser la méthode `entityClass()` pour définir un nom de classe spécifique.

Regardez les chapitres sur *Les Objets Table* et les *Entities* pour plus d'informations sur la façon d'utiliser les objets table et les entities dans votre application.

Pour en savoir plus

Notions de Base de Base de Données

La couche d'accès à la base de données de CakePHP fournit une abstraction et une aide sur la plupart des aspects des traitements de bases de données relationnelles telles que le maintien des connexions au serveur, la construction de requêtes, la protection contre les injections SQL, l'inspection et la modification des schémas, et avec le débogage et le profilage des requêtes envoyées à la base de données.

Tour Rapide

Les fonctions décrites dans ce chapitre illustrent les possibilités de l'API de bas niveau d'accès à la base de données. Si vous souhaitez plutôt en apprendre plus sur l'ORM complet, vous pouvez lire les sections portant sur le *Query Builder* et *Les Objets Table*.

La manière la plus simple de créer une connexion à la base de données est d'utiliser une chaine `DSN` :

```
use Cake\Datasource\ConnectionManager;

$dsn = 'mysql://root:password@localhost/my_database';
ConnectionManager::setConfig('default', ['url' => $dsn]);
```

Vous pouvez commencer à utiliser l'objet de connexion aussitôt après l'avoir créé :

```
$connection = ConnectionManager::get('default');
```

> **Note**
>
> Pour en savoir plus sur les bases de données supportées, consultez le chapitre *installation notes*.

Exécuter des Instructions Select

Exécuter une instruction SQL pure est un jeu d'enfant :

```
use Cake\Datasource\ConnectionManager;

$connection = ConnectionManager::get('default');
$results = $connection->execute('SELECT * FROM articles')->fetchAll('assoc');
```

Vous pouvez utiliser des requêtes préparées pour insérer des paramètres :

```
$results = $connection
    ->execute('SELECT * FROM articles WHERE id = :id', ['id' => 1])
    ->fetchAll('assoc');
```

Il est également possible d'utiliser des types de données complexes en tant qu'arguments :

```
use Cake\Datasource\ConnectionManager;
use DateTime;

$connection = ConnectionManager::get('default');
$results = $connection
    ->execute(
        'SELECT * FROM articles WHERE created >= :created',
        ['created' => new DateTime('1 day ago')],
        ['created' => 'datetime']
    )
    ->fetchAll('assoc');
```

Au lieu d'écrire du SQL manuellement, vous pouvez utiliser le générateur de requêtes :

```
$results = $connection
    ->newQuery()
    ->select('*')
    ->from('articles')
    ->where(['created >' => new DateTime('1 day ago')], ['created' => 'datetime'])
    ->order(['title' => 'DESC'])
    ->execute()
    ->fetchAll('assoc');
```

Exécuter des Instructions Insert

Insérer une ligne dans une base de données est habituellement l'affaire de quelques lignes :

```
use Cake\Datasource\ConnectionManager;
use DateTime;

$connection = ConnectionManager::get('default');
$connection->insert('articles', [
    'title' => 'A New Article',
    'created' => new DateTime('now')
], ['created' => 'datetime']);
```

Exécuter des Instructions Update

Mettre à jour une ligne de base de données est tout aussi intuitif. L'exemple suivant procédera à la mise à jour de l'article comportant l'**id** 10 :

```
use Cake\Datasource\ConnectionManager;
$connection = ConnectionManager::get('default');
$connection->update('articles', ['title' => 'New title'], ['id' => 10]);
```

Exécuter des Instructions Delete

De même, la méthode `delete()` est utilisée pour supprimer des lignes de la base de données. L'exemple suivant procédera à suppression de l'article comportant l'**id** 10 :

```
use Cake\Datasource\ConnectionManager;
$connection = ConnectionManager::get('default');
$connection->delete('articles', ['id' => 10]);
```

Configuration

Par convention, les connexions à la base de données sont configurées dans **config/app.php**. L'information de connexion définie dans ce fichier est envoyée au `Cake\DataSource\ConnectionManager` créant la configuration de la connexion que votre application utilisera. Un exemple d'information sur la connexion se trouve dans **config/app.default.php**. La configuration de la connexion pourrait par exemple ressembler à ceci :

```
'Datasources' => [
    'default' => [
        'className' => 'Cake\Database\Connection',
        'driver' => 'Cake\Database\Driver\Mysql',
        'persistent' => false,
        'host' => 'localhost',
        'username' => 'my_app',
        'password' => 'secret',
        'database' => 'my_app',
        'encoding' => 'utf8mb4',
        'timezone' => 'UTC',
        'cacheMetadata' => true,
    ]
],
```

Le code ci-dessus va créer une connexion "default", avec les paramètres fournis. Vous pouvez définir autant de connexions que vous le souhaitez dans votre fichier de configuration. Vous pouvez aussi définir des connexions supplémentaires à la volée en utilisant `Cake\DataSource\ConnectionManager::setConfig()`. Voici un exemple :

```
use Cake\Datasource\ConnectionManager;

ConnectionManager::setConfig('default', [
    'className' => 'Cake\Database\Connection',
    'driver' => 'Cake\Database\Driver\Mysql',
    'persistent' => false,
    'host' => 'localhost',
    'username' => 'my_app',
    'password' => 'secret',
    'database' => 'my_app',
    'encoding' => 'utf8mb4',
    'timezone' => 'UTC',
    'cacheMetadata' => true,
]);
```

Les options de configuration peuvent également être fournies en tant que chaîne *DSN*. C'est utile lorsque vous travaillez avec des variables d'environnement ou des fournisseurs *PaaS* :

```
ConnectionManager::setConfig('default', [
    'url' => 'mysql://my_app:secret@localhost/my_app?encoding=utf8&timezone=UTC&
↪cacheMetadata=true',
]);
```

Lorsque vous utilisez une chaîne DSN, vous pouvez définir des paramètres/options supplémentaires en tant qu'arguments de query string.

Par défaut, tous les objets Table vont utiliser la connexion `default`. Pour utiliser une autre connexion, reportez-vous à *la configuration des connexions*.

La configuration de la base de données supporte de nombreuses clés. Voici la liste complète :

className
: Nom de classe complet (incluant le *namespace*) de la classe qui représente une connexion au serveur de base de données. Cette classe a pour rôle de charger le pilote de base de données, de fournir les mécanismes de transaction et de préparer les requêtes SQL (entres autres choses).

driver
: Le nom de la classe du pilote utilisé pour implémenter les spécificités d'un moteur de bases de données. Cela peut être soit un nom de classe court en utilisant la *syntaxe de plugin*, soit un nom complet avec namespace, soit une instance du pilote déjà construite. Les exemples de noms de classe courts sont Mysql, Sqlite, Postgres, et Sqlserver.

persistent
: S'il faut utiliser ou non une connexion persistante à la base de données. Cette option n'est pas supportée par SqlServer. Une exception est lancée si vous essayez de définir `persistent` à `true` sur SqlServer.

host
: Le nom d'hôte du serveur de base de données (ou une adresse IP).

username
: Le nom d'utilisateur pour votre compte.

password
: Le mot de passe pour le compte.

database
: Le nom de la base de données à utiliser pour cette connexion. Éviter d'utiliser `.` dans votre nom de base de données. CakePHP ne supporte pas `.` dans les noms de base de données parce que cela complique l'échappement des identifiants. Les chemins vers vos bases de données SQLite doivent être absolus (par exemple `ROOT . DS . 'my_app.db'`) pour éviter les erreurs de chemins incorrects à cause de chemins relatifs.

port (*optionnel*)
: Le port TCP ou le socket Unix utilisé pour se connecter au serveur.

encoding
: Indique le jeu de caractères à utiliser lors de l'envoi d'instructions SQL au serveur. L'encodage par défaut est celui de la base de données pour toutes les bases de données autres que DB2.

timezone
: Le timezone du serveur.

schema
: Utilisé pour spécifier le schema à utiliser pour les bases de données PostgreSQL.

unix_socket
: Utilisé par les drivers qui le supportent pour se connecter via les fichiers socket Unix. Si vous utilisez PostgreSQL et que vous voulez utiliser les sockets Unix, laissez la clé host vide.

ssl_key
: Le chemin vers le fichier de clé SSL (supporté seulement par MySQL).

ssl_cert
: Le chemin vers le fichier du certificat SSL (supporté seulement par MySQL).

ssl_ca
: Le chemin vers le fichier de l'autorité de certification SSL (supporté seulement par MySQL).

init
Une liste de requêtes qui doivent être envoyées au serveur de la base de données lorsque la connexion est créée.

log
Défini à `true` pour activer les logs des requêtes. Si activé, les requêtes seront écrites au niveau `debug` avec le scope `queriesLog`.

quoteIdentifiers
Défini à `true` si vous utilisez des mots réservés ou des caractères spéciaux dans les noms de tables ou de colonnes. Si cette option est activée, les identificateurs seront quotés lors de la génération du SQL dans les requêtes construites avec le *Query Builder*. Notez que ceci diminue la performance parce que chaque requête a besoin d'être traversée et manipulée avant d'être exécutée.

flags
Un tableau associatif de constantes PDO qui doivent être passées à l'instance PDO sous-jacente. Regardez la documentation de PDO pour les flags supportés par le pilote que vous utilisez.

cacheMetadata
Soit un booléen `true`, soit une chaîne contenant la configuration du cache pour stocker les métadonnées. Désactiver la mise en cache des métadonnées n'est pas conseillé et peut entraîner de faibles performances. Consultez la section sur *La Mise en Cache des Métadonnées* pour plus d'information.

mask
Définit les droits sur le fichier de base de données généré (seulement supporté par SQLite)

cache
Le drapeau `cache` à envoyer à SQLite.

mode
La valeur du drapeau `mode` à envoyer à SQLite.

Au point où nous en sommes, vous pouvez aller voir *Conventions de CakePHP*. Le nommage correct de vos tables (et de quelques colonnes) peut vous offrir des fonctionnalités utiles sans aucun effort et vous éviter d'avoir à faire de la configuration. Par exemple, si vous nommez votre table de base de données big_boxes, votre table BigBoxesTable, et votre controller BigBoxesController, tout fonctionnera ensemble automatiquement. Par convention, utilisez les underscores, les minuscules et les formes plurielles pour vos noms de table de la base de données - par exemple : bakers, pastry_stores, et savory_cakes.

Note

Si votre serveur MySQL est configuré avec `skip-character-set-client-handshake` alors vous DEVEZ utiliser la clé de configuration `flags` pour définir votre encodage de caractères. Par exemple :

```
'flags' => [\PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES utf8']
```

Gérer les Connexions

class Cake\Datasource**ConnectionManager**

La classe `ConnectionManager` agit comme un registre pour accéder aux connexions à la base de données que votre application. Elle fournit un endroit où les autres objets peuvent obtenir des références aux connexions existantes.

Accéder à des Connexions

static Cake\Datasource\ConnectionManager::**get**(*$name*)

Une fois configurées, les connexions peuvent être récupérées en utilisant `Cake\Datasource\ConnectionManager::get()`. Cette méthode va construire et charger une connexion si elle n'a pas été déjà construite avant, ou retourner la connexion connue existante :

```
use Cake\Datasource\ConnectionManager;

$connection = ConnectionManager::get('default');
```

La tentative de chargement de connexions qui n'existent pas va lancer une exception.

Créer des Connexions à l'Exécution

En utilisant `setConfig()` et `get()`, vous pouvez créer à tout moment de nouvelles connexions qui ne sont pas définies dans votre fichier de configuration :

```
ConnectionManager::setConfig('ma_connexion', $config);
$connection = ConnectionManager::get('ma_connexion');
```

Consultez le chapitre sur la *configuration* pour plus d'informations sur les données de configuration utilisées lors de la création de connexions.

Types de Données

class `Cake\Database\`**`TypeFactory`**

Puisque tous les fournisseurs de base de données n'intègrent pas la même définition des types de données, ou pas les mêmes noms pour des types de données similaires, CakePHP fournit un ensemble de types de données abstraits à utiliser avec la couche de la base de données. Les types supportés par CakePHP sont :

string
: Correspond au type `VARCHAR`. Avec SQL Server, c'est le type `NVARCHAR` qui est utilisé.

char
: Correspond au type `CHAR`. Avec SQL Server, c'est le type `NCHAR` qui est utilisé.

text
: Correspond aux types `TEXT`.

uuid
: Correspond au type UUID si une base de données en fournit un, sinon cela générera un champ `CHAR(36)`.

binaryuuid
: Correspond au type UUID si la base de données en fournit un, sinon cela générera un champ `BINARY(16)`.

integer
: Correspond au type `INTEGER` fourni par la base de données. BIT n'est pour l'instant pas supporté.

smallinteger
: Correspond au type `SMALLINT` fourni par la base de données.

tinyinteger
: Correspond au type `TINYINT` ou `SMALLINT` fourni par la base de données. Sur MySQL `TINYINT(1)` sera traité comme un booléen.

biginteger
: Correspond au type `BIGINT` fourni par la base de données.

float
: Correspond soit à `DOUBLE`, soit à `FLOAT` selon la base de données. L'option `precision` peut être utilisée pour définir la précision utilisée.

decimal
: Correspond au type `DECIMAL`. Supporte les options `length` et `precision`. Les valeurs du type *decimal* sont représentées par des chaînes de texte (et non par des *float* comme on pourrait s'y attendre). Cela vient du fait que les types décimaux sont utilisés pour réprésenter des valeurs numériques exactes dans les bases de données, alors que l'utilisation de type flottants en PHP peut potentiellement entraîner des pertes de précision.

  Si vous voulez que les valeurs soient représentées par des *float* dans votre code PHP, envisagez plutôt d'utiliser des types de colonnes *FLOAT* ou *DOUBLE* dans votre base de données. Ensuite, selon l'utilisation que vous en

ferez, vous pourrez faire correspondre explicitement vos colonnes décimales à un type *float* dans votre schéma de table.

boolean
Correspond au `BOOLEAN` sauf pour MySQL, où `TINYINT(1)` est utilisé pour représenter les booléens. `BIT(1)` n'est pour l'instant pas supporté.

binary
Correspond au type `BLOB` ou `BYTEA` fourni par la base de données.

date
Correspond au type de colonne natif `DATE`. La valeur de retour de ce type de colonne est `Cake\I18n\Date` qui étend la classe native `DateTime`.

datetime
Consultez *Type DateTime*.

datetimefractional
Consultez *Type DateTime*.

timestamp
Correspond au type `TIMESTAMP`.

timestampfractional
Correspond au type `TIMESTAMP(N)`.

time
Correspond au type `TIME` dans toutes les bases de données.

json
Correspond au type `JSON` s'il est disponible, sinon il correspond à `TEXT`.

Ces types sont utilisés à la fois pour les fonctionnalités de reflection de schema fournies par CakePHP, et pour les fonctionnalités de génération de schema que CakePHP utilise lors des fixtures de test.

Chaque type peut aussi fournir des fonctions de traduction entre les représentations PHP et SQL. Ces méthodes sont invoquées selon les spécifications de type fournies lorsque les requêtes sont créées. Par exemple une colonne qui est marquée en "datetime" va automatiquement convertir les paramètres d'entrée d'instances `DateTime` en timestamp ou en chaînes de dates formatées. De même, les colonnes "binary" vont accepter des transmissions de fichiers, et générer un fichier lors de la lecture des données.

Type DateTime

class `Cake\Database\`**`DateTimeType`**

Correspond à un type de colonne natif `DATETIME`. Dans PostgreSQL et SQL Server, il s'agit du type `TIMESTAMP`. La valeur de retour par défaut de ce type de colonne est `Cake\I18n\FrozenTime` qui étend la classe intégrée `DateTimeImmutable` et Chronos [133].

`Cake\Database\DateTimeType::`**`setTimezone`**(*string|\DateTimeZone|null $timezone*)

Si le fuseau horaire de votre serveur de base de données ne correspond pas au fuseau horaire PHP de votre application, vous pouvez utiliser cette méthode pour spécifier le fuseau horaire de votre base de données. Ce fuseau horaire sera alors utilisé lors de la conversion des objets PHP en chaîne de date de la base de données et vice-versa.

class `Cake\Database\`**`DateTimeFractionalType`**

Peut être utilisé pour mettre en correspondance des colonnes de date et heure qui contiennent des microsecondes, telles que `DATETIME(6)` dans MySQL. Pour utiliser ce type, vous devez l'ajouter en tant que type mappé :

```
// dans config/bootstrap.php
use Cake\Database\TypeFactory;
use Cake\Database\Type\DateTimeFractionalType;
```

(suite sur la page suivante)

133. https://github.com/cakephp/chronos

(suite de la page précédente)

```
// Remplacer le type de date par défaut par un type plus précis.
TypeFactory::map('datetime', DateTimeFractionalType::class);
```

class Cake\Database**DateTimeTimezoneType**

Peut être utilisé pour mapper des colonnes date et heure qui contiennent des fuseaux horaires comme `TIMESTAMPTZ` dans PostgreSQL. Pour utiliser ce type, vous devez l'ajouter en tant que type mappé :

```
// dans config/bootstrap.php
use Cake\Database\TypeFactory;
use Cake\Database\Type\DateTimeTimezoneType;

// Remplacer le type de date par défaut par un type plus précis.
TypeFactory::map('datetime', DateTimeTimezoneType::class);
```

Ajouter des Types Personnalisés

class Cake\Database**TypeFactory**

static Cake\Database\TypeFactory::**map**(*$name*, *$class*)

Si vous avez besoin d'utiliser des types spécifiques qui ne sont pas fournis CakePHP, vous pouvez ajouter de nouveaux types au système de types de CakePHP. Ces classes de type doivent implémenter les méthodes suivantes :

- `toPHP` : Convertit la valeur spécifiée depuis un type de base de données vers un type PHP équivalent.
- `toDatabase` : Convertit la valeur spécifiée depuis un type PHP vers un type acceptable par la base de données.
- `toStatement` : Convertit la valeur spécifiée vers son équivalent pour la Statement.
- `marshal` : Transforme des données à plat en objets PHP.

Pour remplir l'interface basique, vous pouvez étendre `Cake\Database\Type`. Par exemple, si nous souhaitions ajouter un type JSON, nous pourrions créer la classe de type suivante :

```
// Dans src/Database/Type/JsonType.php

namespace App\Database\Type;

use Cake\Database\DriverInterface;
use Cake\Database\Type\BaseType;
use PDO;

class JsonType extends BaseType
{
    public function toPHP($value, DriverInterface $driver)
    {
        if ($value === null) {
            return null;
        }

        return json_decode($value, true);
    }

    public function marshal($value)
    {
        if (is_array($value) || $value === null) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
            return $value;
        }

        return json_decode($value, true);
    }

    public function toDatabase($value, DriverInterface $driver)
    {
        return json_encode($value);
    }

    public function toStatement($value, DriverInterface $driver)
    {
        if ($value === null) {
            return PDO::PARAM_NULL;
        }

        return PDO::PARAM_STR;
    }

}
```

Par défaut, la méthode `toStatement` va traiter les valeurs en chaines qui vont fonctionner pour notre nouveau type.

Connecter des Types Personnalisés à la Reflection et Génération de Schéma

Une fois que nous avons créé notre nouveau type, nous avons besoin de l'ajouter dans la correspondance de type. Pendant le bootstrap de notre application, nous devrions faire ce qui suit :

```
use Cake\Database\TypeFactory;

TypeFactory::map('json', 'App\Database\Type\JsonType');
```

Nous avons ensuite deux façons d'utiliser notre type dans nos modèles.

1. La première façon est d'écraser les données de schéma reflected pour utiliser notre nouveau type.
2. La deuxième est d'implémenter `Cake\Database\Type\ColumnSchemaAwareInterface` et de définir le type de colonne SQL et la logique de reflection.

Écraser le schéma reflected avec notre type personnalisé va activer dans la couche de base de données de CakePHP la conversion automatique de nos données JSON lors de la création de requêtes. Dans votre *méthode getSchema()* de votre Table, ajoutez ceci :

```
use Cake\Database\Schema\TableSchemaInterface;

class WidgetsTable extends Table
{
    public function getSchema(): TableSchemaInterface
    {
        $schema = parent::getSchema();
        $schema->setColumnType('widget_prefs', 'json');

        return $schema;
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
}
```

Le fait d'implémenter `ColumnSchemaAwareInterface` vous donne plus de contrôle sur les types personnalisés Cela évite de réécrire les définitions de schéma si votre type a une définition de colonne SQL ambiguë. Par exemple, notre type JSON pourrait être utilisé pour chaque colonne `TEXT` ayant un commentaire spécifique :

```
// dans src/Database/Type/JsonType.php

namespace App\Database\Type;

use Cake\Database\DriverInterface;
use Cake\Database\Type\BaseType;
use Cake\Database\Type\ColumnSchemaAwareInterface;
use Cake\Database\Schema\TableSchemaInterface;
use PDO;

class JsonType extends BaseType
    implements ColumnSchemaAwareInterface
{
    // les autres méthodes d'avant

    /**
     * Convertit la définition de schéma abstrait en un code SQL spécifique
     * au pilote pouvant être utilisé dans une instruction CREATE TABLE.
     *
     * Returner null va faire retomber vers les types intégrés à CakePHP.
     */
    public function getColumnSql(
        TableSchemaInterface $schema,
        string $column,
        DriverInterface $driver
    ): ?string {
        $data = $schema->getColumn($column);
        $sql = $driver->quoteIdentifier($column);
        $sql .= ' JSON';
        if (isset($data['null']) && $data['null'] === false) {
            $sql .= ' NOT NULL';
        }

        return $sql;
    }

    /**
     * Convertit les données de la colonnes renvoyées par la reflection du
     * schéma en données abstraites de schéma.
     *
     * Returner null va faire retomber vers les types intégrés à CakePHP.
     */
    public function convertColumnDefinition(
        array $definition,
        DriverInterface $driver
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ): ?array {
        return [
            'type' => $this->_name,
            'length' => null,
        ];
    }
```

La donnée `$definition` passée à `convertColumnDefinition()` contiendra les clés suivantes. Toutes les clés existeront mais seront susceptibles de contenir `null` si la clé n'a pas de valeur pour le pilote de base de données actuel :

- `length` La longueur d'une colonne, si disponible.
- `precision` La précision d'une colonne, si disponible.
- `scale` Peut être inclus pour les connexions SQLServer.

Faire correspondre des types de données personnalisés aux expressions SQL

L'exemple précédent fait correspondre un type de données personnalisé pour une colonne de type "json" qui est facilement représenté sous la forme d'une chaîne de texte dans une instruction SQL. Les types de données complexes ne peuvent pas être représentées sous la forme de chaînes/entiers dans des requêtes SQL. Quand vous travaillez avec ces types de données, votre class Type doit implémenter l'interface `Cake\Database\Type\ExpressionTypeInterface`. Cette interface permet de représenter une valeur de votre type de données personnalisé comme une expression SQL. Comme exemple, nous allons construire une simple classe Type pour manipuler le type de données `POINT` de MysQL. En premier lieu, nous allons définir un objet "value" que nous allons pouvoir utiliser pour représenter les données de `POINT` en PHP :

```
// dans src/Database/Point.php
namespace App\Database;

// Notre objet de valeur est immuable.
class Point
{
    protected $_lat;
    protected $_long;

    // Méthode de fabrique.
    public static function parse($value)
    {
        // Analyse les données WKB de MySQL.
        $unpacked = unpack('x4/corder/Ltype/dlat/dlong', $value);

        return new static($unpacked['lat'], $unpacked['long']);
    }

    public function __construct($lat, $long)
    {
        $this->_lat = $lat;
        $this->_long = $long;
    }

    public function lat()
    {
        return $this->_lat;
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    public function long()
    {
        return $this->_long;
    }
}
```

Maintenant que notre objet "value" créé, nous avons besoin d'une classe Type pour faire correspondre les données dans cet objet et les expressions SQL :

```
namespace App\Database\Type;

use App\Database\Point;
use Cake\Database\DriverInterface;
use Cake\Database\Expression\FunctionExpression;
use Cake\Database\ExpressionInterface;
use Cake\Database\Type\BaseType;
use Cake\Database\Type\ExpressionTypeInterface;

class PointType extends BaseType implements ExpressionTypeInterface
{
    public function toPHP($value, DriverInterface $d)
    {
        return Point::parse($value);
    }

    public function marshal($value)
    {
        if (is_string($value)) {
            $value = explode(',', $value);
        }
        if (is_array($value)) {
            return new Point($value[0], $value[1]);
        }

        return null;
    }

    public function toExpression($value): ExpressionInterface
    {
        if ($value instanceof Point) {
            return new FunctionExpression(
                'POINT',
                [
                    $value->lat(),
                    $value->long()
                ]
            );
        }
        if (is_array($value)) {
            return new FunctionExpression('POINT', [$value[0], $value[1]]);
        }
```

(suite sur la page suivante)

(suite de la page précédente)

```
        // Manipulations d'autres cas.
    }

    public function toDatabase($value, DriverInterface $driver)
    {
        return $value;
    }
}
```

La classe ci-dessus fait plusieurs choses intéressantes :

- La méthode `toPHP` se charge de convertir les résulats de la requête SQL en un objet "value".
- La méthode `marshal` se charge de convertir des données, comme celles de la requête, dans notre objet "value". Nous allons accepter des chaînes comme `'10.24,12.34` ainsi que des tableaux.
- La méthode `toExpression` se charge de convertir notre objet "value" dans des expressions SQL équivalentes. Dans notre exemple, le SQL résultant devrait être quelque chose comme `POINT(10.24, 12.34)`.

Une fois que nous avons construit notre type personnalisé, nous allons *connecter notre type à notre classe de table*.

Activer les Objets DateTime Immutables

Du fait que les objets Date/Time sont facilement mutables, CakePHP vous permet d'activer des objets immutables. Le meilleur endroit pour cela est le fichier **config/bootstrap.php**

```
TypeFactory::build('datetime')->useImmutable();
TypeFactory::build('date')->useImmutable();
TypeFactory::build('time')->useImmutable();
TypeFactory::build('timestamp')->useImmutable();
```

> **Note**
>
> Dans les nouvelles applications, les objets immutables seront activés par défaut.

Classes de Connexion

class `Cake\Database\`**`Connection`**

Les classes de connexion fournissent une interface simple pour interagir avec les connexions à la base de données de façon cohérente. Elles ont pour objectif d'être une interface plus abstraite de la couche du pilote et de fournir des fonctionnalités pour l'exécution des requêtes, le logging des requêtes, et l'utilisation de transactions.

Exécuter des Requêtes

`Cake\Database\Connection::`**`query`**(*$sql*)

Une fois que vous avez un objet de connexion, vous allez probablement vouloir réaliser quelques requêtes avec. La couche d'abstraction de CakePHP fournit des fonctionnalités par-dessus PDO et les pilotes natifs. Ces fonctionnalités fournissent une interface similaire à PDO. Il y a plusieurs façons de lancer les requêtes selon le type de requête que vous souhaitez et selon le type de résultat que vous attendez en retour. La méthode la plus basique est `query()` qui vous permet de lancer des requêtes SQL déjà complètes :

```
$statement = $connection->query('UPDATE articles SET published = 1 WHERE id = 2');
```

`Cake\Database\Connection::execute($sql, $params, $types)`

La méthode `query` n'accepte pas de paramètres supplémentaires. Si vous avez besoin de paramètres supplémentaires, vous devrez utiliser la méthode `execute()`, ce qui permet d'utiliser des placeholders :

```
$statement = $connection->execute(
    'UPDATE articles SET published = ? WHERE id = ?',
    [1, 2]
);
```

Sans aucun typage des informations, `execute` va supposer que tous les placeholders sont des chaînes de texte. Si vous avez besoin de lier des types de données spécifiques, vous pouvez utiliser leur nom de type abstrait lors de la création d'une requête :

```
$statement = $connection->execute(
    'UPDATE articles SET published_date = ? WHERE id = ?',
    [new DateTime('now'), 2],
    ['date', 'integer']
);
```

`Cake\Database\Connection::newQuery()`

Cela vous permet d'utiliser des types de données riches dans vos applications et de les convertir convenablement en instructions SQL. La dernière manière de créer des requêtes, et la plus flexible, est d'utiliser le *Query Builder*. Cette approche vous permet de construire des requêtes complexes et expressives sans avoir à utiliser du SQL spécifique à la plateforme :

```
$query = $connection->newQuery();
$query->update('articles')
    ->set(['published' => true])
    ->where(['id' => 2]);
$statement = $query->execute();
```

Quand vous utilisez le query builder, aucun SQL ne sera envoyé au serveur de base de données jusqu'à ce que la méthode `execute()` soit appelée, ou que la requête soit itérée. Itérer une requête va d'abord l'exécuter et ensuite démarrer l'itération sur l'ensemble des résultats :

```
$query = $connection->newQuery();
$query->select('*')
    ->from('articles')
    ->where(['published' => true]);

foreach ($query as $row) {
    // Faire quelque chose avec la ligne.
}
```

Note

Vous pouvez utiliser `all()` pour récupérer l'ensemble de résultats d'une requête SELECT à partir d'une instance de *`Cake\ORM\Query`*.

Utiliser les Transactions

Les objets de connexion vous fournissent quelques moyens simples pour faire des transactions sur la base de données. Le moyen le plus basique est d'utiliser les méthodes `begin`, `commit` et `rollback`, qui correspondent à leurs équivalents SQL :

```
$connection->begin();
$connection->execute('UPDATE articles SET published = ? WHERE id = ?', [true, 2]);
$connection->execute('UPDATE articles SET published = ? WHERE id = ?', [false, 4]);
$connection->commit();
```

`Cake\Database\Connection::transactional(callable $callback)`

En plus de cette interface, les instances de connexion fournissent aussi la méthode `transactional()` qui simplifie considérablement la gestion des appels begin/commit/rollback :

```
$connection->transactional(function ($connection) {
    $connection->execute('UPDATE articles SET published = ? WHERE id = ?', [true, 2]);
    $connection->execute('UPDATE articles SET published = ? WHERE id = ?', [false, 4]);
});
```

En plus des requêtes basiques, vous pouvez exécuter des requêtes plus complexes en utilisant soit le *Query Builder*, soit *Les Objets Table*. La méthode transactionnelle fera les traitements suivants :

- Appel de `begin`.
- Appelle la closure fournie.
- Si la closure lance une exception, un rollback sera délivré. L'exception originelle sera relancée.
- Si la closure retourne `false`, un rollback sera délivré.
- Si la closure s'exécute avec succès, la transaction sera commitée.

Interagir avec les Requêtes

Lors de l'utilisation de l'API de plus bas niveau, vous rencontrerez souvent des objets *statement*. Ces objets vous permettent de manipuler les requêtes sous-jacentes préparées par le pilote. Après avoir créé et exécuté un objet query, ou en utilisant `execute()`, vous devriez avoir une instance `StatementDecorator`. Elle enveloppe l'objet *statement* basique sous-jacent et fournit quelques fonctionnalités supplémentaires.

Préparer une Statement

Vous pouvez créer un objet *statement* en utilisant `execute()`, ou `prepare()`. La méthode `execute()` retourne une *statement* à laquelle sont reliées les valeurs des paramètres. Tandis que `prepare()` retourne une *statement* incomplète :

```
// Les statements à partir de execute auront déjà des valeurs liées aux paramètres.
$statement = $connection->execute(
    'SELECT * FROM articles WHERE published = ?',
    [true]
);

// Les statements à partir de prepare auront des placeholders pour les paramètres.
// Vous avez besoin de lier les paramètres avant de tenter de les exécuter.
$statement = $connection->prepare('SELECT * FROM articles WHERE published = ?');
```

Une fois que vous avez préparé une *statement*, vous pouvez lier les données supplémentaires et l'exécuter.

Lier les Valeurs (Binding)

Une fois que vous avez créé une requête préparée, vous aurez probablement besoin de lier des données supplémentaires. Vous pouvez lier plusieurs valeurs en une fois en utilisant la méthode `bind`, ou lier des éléments individuels en utilisant `bindValue` :

```
$statement = $connection->prepare(
    'SELECT * FROM articles WHERE published = ? AND created > ?'
);

// Lier plusieurs valeurs
$statement->bind(
    [true, new DateTime('2013-01-01')],
    ['boolean', 'date']
);

// Lier une valeur unique
$statement->bindValue(1, true, 'boolean');
$statement->bindValue(2, new DateTime('2013-01-01'), 'date');
```

Lors de la création de requêtes, vous pouvez aussi utiliser des tableaux associatifs plutôt que des clés de position :

```
$statement = $connection->prepare(
    'SELECT * FROM articles WHERE published = :published AND created > :created'
);

// Lier plusieurs valeurs
$statement->bind(
    ['published' => true, 'created' => new DateTime('2013-01-01')],
    ['published' => 'boolean', 'created' => 'date']
);

// Lier une valeur unique
$statement->bindValue('published', true, 'boolean');
$statement->bindValue('created', new DateTime('2013-01-01'), 'date');
```

Avertissement

Vous ne pouvez pas mixer les clés de position et les clés nommées dans la même requête.

Exécuter & Récupérer les Colonnes

Après avoir préparé une requête et y avoir lié des données, vous pouvez l'exécuter et récupérer les lignes. La requêtes devra être exécutée en utilisant la méthode `execute()`. Après l'avoir exécutée, vous pouver récupérer les résultats en utilisant `fetch()`, `fetchAll()` ou en itérant sur la requête :

```
$statement->execute();

// Lire une ligne.
$row = $statement->fetch('assoc');

// Lire toutes les lignes.
```

(suite sur la page suivante)

(suite de la page précédente)

```
$rows = $statement->fetchAll('assoc');

// Lire les lignes en faisant une itération.
foreach ($statement as $row) {
    // Faire quelque chose
}
```

> **Note**
>
> Lire les lignes par itération va récupérer les lignes dans le mode "both". Cela signifie que les résultats que vous aurez seront indexés à la fois numériquement et de manière associative.

Récupérer le Nombre de Lignes

Après avoir exécuté une requête, vous pouvez récupérer le nombre de lignes affectées :

```
$rowCount = count($statement);
$rowCount = $statement->rowCount();
```

Vérifier les Codes d'Erreur

Si votre requête a échoué, vous pouvez obtenir des informations sur l'erreur en utilisant les méthodes `errorCode()` et `errorInfo()`. Ces méthodes fonctionnent de la même façon que celles fournies par PDO :

```
$code = $statement->errorCode();
$info = $statement->errorInfo();
```

Générer des Logs de Requêtes

Le logs de requêtes peuvent être activés lors de la configuration de votre connexion en définissant l'option `log` à `true`. Vous pouvez aussi changer le log de requêtes à la volée, en utilisant `enableQueryLogging` :

```
// Active les logs des requêtes.
$connection->enableQueryLogging(true);

// Stoppe les logs des requêtes
$connection->enableQueryLogging(false);
```

Quand les logs de requêtes sont activés, les requêtes sont loguées dans *Cake\Log\Log* en utilisant le niveau "debug", et le scope "queriesLog". Vous aurez besoin d'avoir un logger configuré pour capter ce niveau et ce scope. Faire des logs vers `stderr` peut être utile lorsque vous travaillez sur les tests unitaires, et les logs de fichiers/syslog peuvent être utiles lorsque vous travaillez avec des requêtes web :

```
use Cake\Log\Log;

// Logs vers la Console
Log::setConfig('queries', [
    'className' => 'Console',
    'stream' => 'php://stderr',
    'scopes' => ['queriesLog']
]);
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Logs vers des Fichiers
Log::setConfig('queries', [
    'className' => 'File',
    'path' => LOGS,
    'file' => 'queries.log',
    'scopes' => ['queriesLog']
]);
```

> **Note**
>
> Les logs des requêtes sont à utiliser seulement pour le débogage/développement. Vous ne devriez jamais laisser les logs de requêtes activés en production car cela va avoir un impact négatif sur les performances de votre application.

Échapper les Identificateurs

Par défaut CakePHP **ne** quote **pas** les identificateurs dans les requêtes SQL générées. La raison en est que l'ajout de quotes autour des identificateurs a quelques inconvénients :

- Par-dessus tout la performance - Ajouter des quotes est bien plus lent et complexe que de ne pas le faire.
- Pas nécessaire dans la plupart des cas - Dans des bases de données récentes qui suivent les conventions de CakePHP, il n'y a pas de raison de quoter les identificateurs.

Si vous utilisez un schema datant un peu qui nécessite de quoter les identificateurs, vous pouvez l'activer en utilisant le paramètre `quoteIdentifiers` dans votre *Configuration*. Vous pouvez aussi activer cette fonctionnalité à la volée :

```
$connection->getDriver()->enableAutoQuoting();
```

Quand elle est activée, la fonctionnalité d'échappement va entraîner des traversées supplémentaires de requêtes qui vont convertir tous les identificateurs en objets `IdentifierExpression`.

> **Note**
>
> Les portions de code SQL contenues dans les objets QueryExpression ne seront pas modifiées.

La Mise en Cache des Métadonnées

L'ORM de CakePHP utilise la réflexivité des bases de données pour déterminer le schéma, les index et les clés étrangères de votre application. Comme ces méta-données changent peu fréquemment et qu'il peut être lourd d'y accéder, elles sont habituellement mises en cache. Par défaut, les méta-données sont stockées dans la configuration de cache `_cake_model_`. Vous pouvez définir une configuration de cache personnalisée en utilisant l'option `cacheMetatdata` dans la configuration de la source de données :

```
'Datasources' => [
    'default' => [
        // Autres clés ici.

        // Utilise la config de cache 'orm_metadata' pour les méta-données.
        'cacheMetadata' => 'orm_metadata',
    ]
],
```

Vous pouvez aussi configurer le cache des méta-données à l'exécution avec la méthode `cacheMetadata()` :

```
// Désactive le cache
$connection->cacheMetadata(false);

// Active le cache
$connection->cacheMetadata(true);

// Utilise une config de cache personnalisée
$connection->cacheMetadata('orm_metadata');
```

CakePHP intègre aussi un outil CLI pour gérer les mises en cache de méta-données. Consultez le chapitre *Shell du Cache du Schéma* pour plus d'information.

Créer des Bases de Données

Si vous voulez créer une connexion sans sélectionner de base de données, vous pouvez omettre le nom de la base de données :

```
$dsn = 'mysql://root:password@localhost/';
```

Vous pouvez maintenant utiliser votre objet de connexion pour exécuter des requêtes qui créent/modifient des bases de données. Par exemple pour créer une base de données :

```
$connection->query("CREATE DATABASE IF NOT EXISTS my_database");
```

> **Note**
>
> Lorsque vous créez une base de données, il est recommandé de définir le jeu de caractères ainsi que les paramètres de collation. Si ces valeurs sont manquantes, la base de données utilisera les valeurs par défaut du système quelles qu'elles soient.

Query Builder

class Cake\ORM**Query**

Le constructeur de requête de l'ORM fournit une interface facile à utiliser pour créer et lancer les requêtes. En arrangeant les requêtes ensemble, vous pouvez créer des requêtes avancées en utilisant les unions et les sous-requêtes avec facilité.

Sous le capot, le constructeur de requête utilise les requêtes préparées de PDO qui protègent contre les attaques d'injection SQL.

L'Objet Query

La façon la plus simple de créer un objet `Query` est d'utiliser `find()` à partir d'un objet `Table`. Cette méthode va retourner une requête incomplète prête à être modifiée. Vous pouvez aussi utiliser un objet de connexion à une table pour accéder à un constructeur de requête de plus bas niveau qui n'inclut pas les fonctionnalités de l'ORM, si nécessaire. Consultez la section *Exécuter des Requêtes* pour plus d'informations :

```
use Cake\ORM\Locator\LocatorAwareTrait;

$articles = $this->getTableLocator()->get('Articles');

// Commence une nouvelle requête.
$query = $articles->find();
```

Depuis un controller, vous pouvez utiliser la variable de table créée automatiquement par le système de conventions :

```
// À l'intérieur de ArticlesController.php

$query = $this->Articles->find();
```

Récupérer les Lignes d'une Table

```
use Cake\ORM\Locator\LocatorAwareTrait;

$query = $this->getTableLocator()->get('Articles')->find();

foreach ($query->all() as $article) {
    debug($article->title);
}
```

Pour les exemples restants, imaginez que `$articles` est une *Table*. Quand vous êtes dans des controllers, vous pouvez utiliser `$this->Articles` plutôt que `$articles`.

Presque toutes les méthodes d'un objet `Query` retournent la requête elle-même. cela signifie que les objets `Query` sont lazy, et ne seront pas exécutés à moins que vous ne lui disiez de le faire :

```
$query->where(['id' => 1]); // Retourne le même objet query
$query->order(['title' => 'DESC']); // Toujours le même objet, aucun SQL exécuté
```

Vous pouvez bien sûr chainer les méthodes que vous appelez sur les objets Query :

```
$query = $articles
    ->find()
    ->select(['id', 'name'])
    ->where(['id !=' => 1])
    ->order(['created' => 'DESC']);

foreach ($query->all() as $article) {
    debug($article->created);
}
```

Si vous essayez d'appeler `debug()` sur un objet Query, vous verrez son état interne et le SQL qui sera exécuté dans la base de données :

```
debug($articles->find()->where(['id' => 1]));

// Affiche
// ...
// 'sql' => 'SELECT * FROM articles where id = ?'
// ...
```

Vous pouvez exécuter une requête directement sans avoir à utiliser `foreach`. La façon la plus simple est d'appeler les méthodes `all()` ou `toArray()` :

```
$resultsIteratorObject = $articles
    ->find()
    ->where(['id >' => 1])
    ->all();
```

(suite sur la page suivante)

(suite de la page précédente)

```
foreach ($resultsIteratorObject as $article) {
    debug($article->id);
}

$resultsArray = $articles
    ->find()
    ->where(['id >' => 1])
    ->all()
    ->toList();

foreach ($resultsArray as $article) {
    debug($article->id);
}

debug($resultsArray[0]->title);
```

Dans l'exemple ci-dessus, `$resultsIteratorObject` sera une instance de `Cake\ORM\ResultSet`, un objet que vous pouvez itérer et sur lequel vous pouvez appliquer plusieurs méthodes d'extraction ou de traversée.

Souvent, il n'y a pas besoin d'appeler `all()`, vous pouvez juste itérer l'objet Query pour récupérer ses résultats. les objets Query peuvent également être utilisés directement en tant qu'objet résultat. Essayer d'itérer la requête en utilisant `toList()` ou n'importe qu'elle méthode héritée de *Collection*, provoquera l'exécution de la requête et la récupération des résultats.

Récupérez une Ligne Unique d'une Table

Vous pouvez utilisez la méthode `first()` pour récupérer le premier résultat dans la requête :

```
$article = $articles
    ->find()
    ->where(['id' => 1])
    ->first();

debug($article->title);
```

Récupérer une Liste de Valeurs à Partir d'une Colonne

```
// Utilise la méthode extract() à partir de la libraire collections
// Ceci exécute aussi la requête
$allTitles = $articles->find()->all()->extract('title');

foreach ($allTitles as $title) {
    echo $title;
}
```

Vous pouvez aussi récupérer une liste de clés-valeurs à partir du résultat d'une requête :

```
$list = $articles->find('list')->all();
foreach ($list as $id => $title) {
    echo "$id : $title"
}
```

Pour plus d'informations sur la façon de personnaliser les champs utilisés pour remplir la liste, consultez la section *Trouver les Paires de Clé/Valeur*.

Les Requêtes sont des Objets Collection

Une fois que vous êtes familier avec les méthodes de l'objet Query, il est fortement recommandé de consulter la section *Collection* pour améliorer vos compétences sur une traversée efficace des données. En résumé, il est important de se rappeler que tout ce que vous pouvez appeler sur un objet Collection, vous pouvez aussi le faire avec un objet Query :

```
// Utilise la méthode combine() à partir de la libraire collection
// Ceci est équivalent au find('list')
$keyValueList = $articles->find()->combine('id', 'title');

// Un exemple avancé
$results = $articles->find()
    ->where(['id >' => 1])
    ->order(['title' => 'DESC'])
    ->map(function ($row) { // map() est une méthode de collection, elle exécute la␣
↪requête
        $row->trimmedTitle = trim($row->title);

        return $row;
    })
    ->combine('id', 'trimmedTitle') // combine() est une autre méthode de collection
    ->toArray(); // Aussi une méthode de la librairie collection

foreach ($results as $id => $trimmedTitle) {
    echo "$id : $trimmedTitle";
}
```

Comment les Requêtes sont Évaluées Lazily

Les objets Query sont évalués paresseusement (*lazily*). Cela signifie qu'une requête n'est pas exécutée jusqu'à ce qu'une des actions suivantes soit lancée :

- La requête est itérée avec `foreach()`.
- La méthode `execute()` est appelée. Elle retourne l'objet d'instruction (*statement*) sous-jacent, et est faite pour être utilisée avec les requêtes insert/update/delete.
- La méthode `first()` est appelée. Elle retourne le premier résultat correspondant à l'instruction `SELECT` (elle ajoute `LIMIT 1` à la requête).
- La méthode `all()` est appelée. Elle retourne l'ensemble des résultats et peut seulement être utilisée avec les instructions `SELECT`.
- La méthode `toList()` ou `toArray()` est appelée.

Tant qu'aucune de ces conditions n'est rencontrée, la requête peut être modifiée sans qu'aucun nouveau code SQL ne soit envoyé à la base de données. Cela signifie aussi que si une Query n'a pas été évaluée, aucun SQL ne sera envoyé à la base de données. Une fois exécutée, la modification suivie de la ré-évaluation de la requête va entraîner l'exécution du SQL supplémentaire. Si une requête est exécutée plusieurs fois d'affilée sans avoir été modifiée entre-temps, elle renvoie la même référence.

Si vous souhaitez jeter un œil sur le SQL que CakePHP génère, vous pouvez activer les *logs de requête*.

Récupérer vos Données

CakePHP permet de construire simplement des requêtes `SELECT`. La méthode `select()` vous permet de ne récupérer que les champs qui vous sont nécessaires :

```
$query = $articles->find();
$query->select(['id', 'title', 'body']);
foreach ($query->all() as $row) {
    debug($row->title);
}
```

Vous pouvez également définir des alias pour vos champs en les définissant dans un tableau associatif :

```
// Génère SELECT id AS pk, titre AS alias_de_titre, contenu ...
$query = $articles->find();
$query->select(['pk' => 'id', 'alias_de_titre' => 'titre', 'contenu']);
```

Pour sélectionner les valeurs distinctes dans des champs, vous pouvez utiliser la méthode `distinct()` :

```
// Génère SELECT DISTINCT pays FROM ...
$query = $articles->find();
$query->select(['pays'])
    ->distinct(['pays']);
```

Pour définir certaines conditions basiques, vous pouvez utiliser la méthode `where()` :

```
// Les conditions sont combinées par défaut avec AND
$query = $articles->find();
$query->where(['titre' => 'Premier Post', 'published' => true]);

// Vous pouvez aussi appeler where() plusieurs fois
$query = $articles->find();
$query->where(['titre' => 'Premier Post'])
    ->where(['published' => true]);
```

Vous pouvez aussi passer une fonction anonyme à la méthode `where()`. La fonction anonyme recevra une instance de `\Cake\Database\Expression\QueryExpression` en premier argument, et `\Cake\ORM\Query` pour le second :

```
$query = $articles->find();
$query->where(function (QueryExpression $exp, Query $q) {
    return $exp->eq('published', true);
});
```

Consultez la section *Conditions Avancées* pour voir comment construire des conditions `WHERE` plus complexes.

Sélectionner Certains Champs d'une Table

// Sélectionne seulement id et titre dans la table articles $articles->find()->select([“id”, “titre”]) ;

Si vous souhaitez sélectionner à nouveau tous les champs d'une table après avoir appelé `select($fields)`, vous pouvez passer l'instance de table à `select()` :

```
// Sélectionne tous les champs de la table articles
// ainsi qu'un champ calculé slug
$query = $articlesTable->find();
```

(suite sur la page suivante)

(suite de la page précédente)

```
$query
    ->select(['slug' => $query->func()->concat(['titre' => 'identifier', '-', 'id' =>
↪'identifier'])])
    ->select($articlesTable); // Sélectionne tous les champs de articles
```

Si vous souhaitez sélectionner tous les champs d'une table sauf quelques-uns, vous pouvez utiliser `selectAllExcept()` :

```
$query = $articlesTable->find();

// Obtenir tous les champs sauf le champ published
$query->selectAllExcept($articlesTable, ['published']);
```

Vous pouvez aussi passer un objet `Association` quand vous travaillez avec des associations.

Utiliser les Fonctions SQL

L'ORM de CakePHP offre une abstraction pour les fonctions SQL les plus communément utilisées. Cette abstraction permet à l'ORM de sélectionner l'implémentation de la fonction voulue spécifique à la plateforme. Par exemple, `concat` est implémentée différemment dans MySQL, PostgreSQL et SQL Server. L'abstraction permet à votre code d'être portable :

```
// Génère SELECT COUNT(*) count FROM ...
$query = $articles->find();
$query->select(['count' => $query->func()->count('*')]);
```

Notez que la plupart des fonctions acceptent un argument supplémentaire pour spécifier le type de données à lier aux arguments et/ou le type de la valeur à retourner, par exemple :

```
$query->select(['minDate' => $query->func()->min('date', ['date']);
```

Pour plus de détails, lisez la documentation de `Cake\Database\FunctionsBuilder`.

Vous pouvez accéder aux *wrappers* existants de plusieurs fonctions SQL grâce à `Query::func()` :

`rand()`
: Génère un nombre aléatoire entre 0 et 1 via SQL.

`sum()`
: Calcule une somme. *Les arguments sont considérés comme des valeurs littérales.*

`avg()`
: Calcule une moyenne. *Les arguments sont considérés comme des valeurs littérales.*

`min()`
: Calcule le minimum d'une colonne. *Les arguments sont considérés comme des valeurs littérales.*

`max()`
: Calcule le maximum d'une colonne. *Les arguments sont considérés comme des valeurs littérales.*

`count()`
: Calcule le nombre de valeurs. *Les arguments sont considérés comme des valeurs littérales.*

`concat()`
: Concatène deux valeurs. *Les arguments sont considérés comme des paramètres liés.*

`coalesce()`
: Renvoie la première expression dont la valeur n'est pas NULL. *Les arguments sont considérés comme des paramètres liés.*

`dateDiff()`
: Récupère la différence entre deux dates/heures. *Les arguments sont considérés comme des paramètres liés.*

now()
Renvoie la date et l'heure courantes par défaut, mais accepte "time" ou "date" comme argument pour récupérer seulement l'heure ou la date courante.

extract()
Renvoie une partie de la date spécifiée dans une expression SQL.

dateAdd()
Ajoute une unité de temps à l'expression de date.

dayOfWeek()
Renvoie une FunctionExpression représentant un appel à la fonction SQL WEEKDAY.

Fonctions de fenêtrage

Ces fonctions de fenêtrage (*window-only*) contiennent une expression de fenêtrage par défaut :

rowNumber()
Renvoie une expression Aggregate pour la fonction SQL `ROW_NUMBER()`.

lag()
Renvoie une expression Aggregate pour la fonction SQL `LAG()`.

lead()
Renvoie une expression Aggregate pour la fonction SQL `LEAD()`.

Quand vous fournissez des arguments pour les fonctions SQL, il y a deux types de paramètres que vous pouvez utiliser : les arguments littéraux et les paramètres liés. Les paramètres d'identification/littéraux vous permettent de référencer les colonnes ou d'autres valeurs littérales en SQL. Les paramètres liés peuvent être utilisés pour ajouter en toute sécurité les données utilisateur aux fonctions SQL. Par exemple :

```
$query = $articles->find()->innerJoinWith('Categories');
$concat = $query->func()->concat([
    'Articles.title' => 'identifier',
    ' - CAT: ',
    'Categories.name' => 'identifier',
    ' - Age: ',
    $query->func()->dateDiff([
        'NOW()' => 'literal',
        'Articles.created' => 'identifier'
    ])
]);
$query->select(['link_title' => $concat]);
```

Vous pouvez faire référence à d'autres colonnes ou expressions SQL littérales aussi bien avec `literal` qu'avec `identifier`, cependant `identifier` ajoutera des quotes dans les cas appropriés si l'auto-quoting a été activé. Les arguments qui ne sont pas marqués comme *literal* ni *identifier* seront utilisés comme des paramètres liés, qui serviront à passer des données utilisateur à la fonction. Le code ci-dessus va générer le SQL suivant sur MySQL :

```
SELECT CONCAT(
    Articles.title,
    :c0,
    Categories.name,
    :c1,
    (DATEDIFF(NOW(), Articles.created))
) FROM articles;
```

La valeur `:c0` aura le texte `' - CAT:'` lié quand la requête est exécutée. L'expression `datediff` a été transcrite en SQL de façon appropriée.

Fonctions Personnalisées

Si `func()` ne propose pas déjà un *wrap* de la fonction SQL dont vous avez besoin, vous pouvez l'appeler directement à travers `func()` et passer des arguments et des données utilisateur en toute sécurité comme décrit ci-dessus. Assurez-vous de passer les bons types d'arguments aux fonctions personnalisées, sans quoi ils seront traités comme des paramètres liés :

```
$query = $articles->find();
$year = $query->func()->year([
    'created' => 'identifier'
]);
$time = $query->func()->date_format([
    'created' => 'identifier',
    "'%H:%i'" => 'literal'
]);
$query->select([
    'yearCreated' => $year,
    'timeCreated' => $time
]);
```

Entraînera :

```
SELECT YEAR(created) as yearCreated,
       DATE_FORMAT(created, '%H:%i') as timeCreated
FROM articles;
```

> **Note**
>
> Utilisez `func()` pour passer des données externes (non fiables) à n'importe quelle fonction SQL.

Trier les résultats

Pour appliquer un tri, vous pouvez utiliser la méthode `order` :

```
$query = $articles->find()
    ->order(['title' => 'ASC', 'id' => 'ASC']);
```

Si vous appelez `order()` plusieurs fois sur la même requête, les clauses s'ajouteront les unes aux autres. Néanmoins, quand vous utiliserez les finders, vous aurez parfois besoin de remplacer la clause `ORDER BY` déjà définie. Passez `Query::OVERWRITE` ou `true` au second paramètre de `order()` (idem pour `orderAsc()` ou `orderDesc()`) :

```
$query = $articles->find()
    ->order(['title' => 'ASC']);
// Plus tard, remplacez la clause ORDER BY au lieu de la compléter.
$query = $articles->find()
    ->order(['created' => 'DESC'], Query::OVERWRITE);
```

Vous pouvez utiliser les méthodes `orderAsc` et `orderDesc` pour trier selon des expressions complexes :

```
$query = $articles->find();
$concat = $query->func()->concat([
    'title' => 'identifier',
    'synopsis' => 'identifier'
```

(suite sur la page suivante)

(suite de la page précédente)

```
]);
$query->orderAsc($concat);
```

Pour construire des clauses de tri complexes, utilisez une Closure :

```
$query->orderAsc(function (QueryExpression $exp, Query $query) {
    return $exp->addCase(/* ... */);
 });
```

Limiter les Résultats

Pour limiter le nombre de lignes, ou définir un offset, utilisez les méthodes `limit()` et `page()` :

```
// Récupérer les lignes 50 à 100
$query = $articles->find()
    ->limit(50)
    ->page(2);
```

Comme vous pouvez le voir sur cet exemple, toutes les méthodes qui modifient la requête proposent une interface fluide, vous permettant de construire une requête par l'appel de méthodes chaînées.

Regroupements - Group et Having

Quand vous utilisez les fonctions d'agrégation comme `count` et `sum`, vous pouvez utiliser les clauses `group by` et `having` :

```
$query = $articles->find();
$query->select([
    'count' => $query->func()->count('view_count'),
    'published_date' => 'DATE(created)'
])
->group('published_date')
->having(['count >' => 3]);
```

Instructions Case

L'ORM offre également l'expression SQL `case`. L'expression `case` permet l'implémentation d'une logique `if ... then ... else` dans votre SQL. Cela peut être utile pour créer des rapports sur des données que vous avez besoin d'additionner ou de compter conditionnellement, ou si vous avez besoin de données spécifiques basées sur une condition.

Si vous vouliez savoir combien d'articles sont publiés dans notre base de données, nous pourrions utiliser le SQL suivant :

```
SELECT
COUNT(CASE WHEN published = 'Y' THEN 1 END) AS number_published,
COUNT(CASE WHEN published = 'N' THEN 1 END) AS number_unpublished
FROM articles
```

Pour faire ceci avec le générateur de requêtes, vous utiliseriez le code suivant :

```
$query = $articles->find();
$publishedCase = $query->newExpr()
    ->case()
    ->when(['published' => 'Y'])
    ->then(1);
$unpublishedCase = $query->newExpr()
    ->case()
    ->when(['published' => 'N'])
    ->then(1);

$query->select([
    'number_published' => $query->func()->count($publishedCase),
    'number_unpublished' => $query->func()->count($unpublishedCase)
]);
```

La méthode `when()` accepte des extraits de code SQL, des conditions en tableau, et des `Closure` pour les situations où vous avez besoin d'une logique supplémentaire pour construire les cas. Si nous souhaitions classer des villes selon des tailles de population PETITE, MOYENNE, ou GRANDE, nous pourrions le faire ainsi :

```
$query = $cities->find();
$sizing = $query->newExpr()->case()
    ->when(['population <' => 100000])
    ->then('PETITE')
    ->when($query->newExpr()->between('population', 100000, 999000))
    ->then('MOYENNE')
    ->when(['population >=' => 999001])
    ->then('GRANDE');
$query = $query->select(['size' => $sizing]);
# SELECT CASE
#   WHEN population < 100000 THEN 'PETITE'
#   WHEN population BETWEEN 100000 AND 999000 THEN 'MOYENNE'
#   WHEN population >= 999001 THEN 'GRANDE'
#   END AS size
```

Il faut que vous fassiez attention lorsque vous incluez dans des expression *case* des données fournies par l'utilisateur, car cela peut créer des vulnérabilités aux injections SQL :

```
// Non sécurisé. *Ne pas* utiliser
$case->when($requestData['published']);

// Au lieu de cela, passez les données utilisateur en valeurs dans un
// tableau de conditions
$case->when(['published' => $requestData['published']]);
```

Pour des scénarios plus complexes, vous pouvez utiliser les objets `QueryExpression` et les valeurs liées :

```
$userValue = $query->newExpr()
    ->case()
    ->when($query->newExpr('population >= :userData'))
    ->then(123, 'integer');

$query->select(['val' => $userValue])
    ->bind(':userData', $requestData['value'], 'integer');
```

L'utilisation des valeurs liées permet d'insérer des données utilisateur en toute sécurité dans des bouts de code SQL bruts complexes. `then()`, `when()` et `else()` essayeront de deviner le type de valeur en se basant sur le type de paramètre. Si vous avez besoin de lier une valeur d'un type différent, vous pouvez déclarer le type souhaité :

```
$case->when(['published' => true])->then('1', 'integer');
```

Vous pouvez créer des conditions `if ... then ... else` en utilisant `else()` :

```
$published = $query->newExpr()
    ->case()
    ->when(['published' => true])
    ->then('Y');
    ->else('N');

# CASE WHEN published = true THEN 'Y' ELSE 'N' END;
```

Il est également possible créer une variable simple en passant une valeur à `case()` :

$published = $query->newExpr()
->case($query->identifier("published")) ->when(true) ->then("Y"); ->else("N");

CASE published WHEN true THEN "Y" ELSE "N" END;

Avant 4.3.0, vous deviez utiliser :

```
$query = $articles->find();
$publishedCase = $query->newExpr()
    ->addCase(
        $query->newExpr()->add(['published' => 'Y']),
        1,
        'integer'
    );
$unpublishedCase = $query->newExpr()
    ->addCase(
        $query->newExpr()->add(['published' => 'N']),
        1,
        'integer'
    );

$query->select([
    'number_published' => $query->func()->count($publishedCase),
    'number_unpublished' => $query->func()->count($unpublishedCase)
]);
```

La fonction `addCase` peut aussi chaîner ensemble plusieurs instructions pour créer une logique `if .. then .. [elseif .. then .. ] [ .. else ]` dans votre SQL.

Si nous souhaitions classer des villes selon des tailles de population PETITE, MOYENNE, ou GRANDE, nous pourrions le faire ainsi :

```
$query = $villes->find()
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->addCase(
            [
                $q->newExpr()->lt('population', 100000),
                $q->newExpr()->between('population', 100000, 999000),
```

(suite sur la page suivante)

(suite de la page précédente)

```
                $q->newExpr()->gte('population', 999001),
            ],
            ['PETITE',  'MOYENNE', 'GRANDE'], # valeurs correspondant aux conditions
            ['string', 'string', 'string']    # type de chaque valeur
        );
    });
# WHERE CASE
#   WHEN population < 100000 THEN 'PETITE'
#   WHEN population BETWEEN 100000 AND 999000 THEN 'MOYENNE'
#   WHEN population >= 999001 THEN 'GRANDE'
#   END
```

S'il y a moins de conditions que de valeurs, `addCase` va automatiquement produire une instruction `if .. then .. else` :

```
$query = $villes->find()
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->addCase(
            [
                $q->newExpr()->eq('population', 0),
            ],
            ['DESERTE', 'INHABITEE'], # valeurs correspondant aux conditions
            ['string', 'string']      # type de chaque valeur
        );
    });
# WHERE CASE
#   WHEN population = 0 THEN 'DESERTE' ELSE 'INHABITEE' END
```

Récupérer des Tableaux plutôt que des Entities

Bien que les ensembles de résultats en objet de l'ORM soient puissants, créer des entities est parfois superflu. Par exemple, quand vous accédez à des données agrégées, la construction d'une Entity n'a pas vraiment de sens. Le processus de conversion des résultats de la base de données en entities est appelé hydratation. Si vous souhaitez désactiver ce processus, vous pouvez faire ceci :

```
$query = $articles->find();
$query->enableHydration(false); // Résultats en tableaux plutôt qu'en entities
$result = $query->toList(); // Exécute la requête et retourne le tableau
```

Après avoir exécuté ces lignes, votre résultat devrait ressembler à quelque chose comme ceci :

```
[
    ['id' => 1, 'title' => 'Premier Article', 'body' => 'Corps de l\'article 1' ...],
    ['id' => 2, 'title' => 'Deuxième Article', 'body' => 'Corps de l\'article 2' ...],
    ...
]
```

Ajouter des Champs Calculés

Après vos requêtes, vous aurez peut-être besoin de faire des traitements postérieurs. Si vous voulez ajouter quelques champs calculés ou des données dérivées, vous pouvez utiliser la méthode `formatResults()`. C'est une façon légère d'appliquer une fonction sur les ensembles de résultats. Si vous avez besoin de plus de contrôle sur le processus, ou si

vous souhaitez réduire les résultats, vous devriez plutôt utiliser la fonctionnalité de *Map/Reduce*. Si vous requêtiez sur une liste de personnes, vous pourriez calculer leur âge avec un formateur de résultats (*result formatter*) :

```
// En supposant que nous avons construit les champs, les conditions et les contain.
$query->formatResults(function (\Cake\Collection\CollectionInterface $results) {
    return $results->map(function ($row) {
        $row['age'] = $row['date_de_naissance']->diff(new \DateTime)->y;

        return $row;
    });
});
```

Comme vous pouvez le voir dans cet exemple, les callbacks de formatage récupéreront un `ResultSetDecorator` en premier argument. Le second argument sera l'instance Query sur laquelle le formateur a été attaché. L'argument `$results` peut être traversé et modifié autant que nécessaire.

Les formateurs de résultat deivent retourner un objet itérateur, qui sera utilisé comme valeur retournée pour la requête. Les fonctions de formatage sont appliquées après que toutes les routines Map/Reduce auront été exécutées. Les formateurs de résultat peuvent aussi être appliqués depuis des associations `contain`. CakePHP va s'assurer que vos formateurs s'appliquent au bon périmètre. Par exemple, ceci fonctionnera comme vous pouvez vous y attendre :

```
// Dans une méthode dans la table Articles
$query->contain(['Auteurs' => function ($q) {
    return $q->formatResults(function (\Cake\Collection\CollectionInterface $auteurs) {
        return $auteurs->map(function ($auteur) {
            $auteur['age'] = $auteur['date_de_naissance']->diff(new \DateTime)->y;

            return $auteur;
        });
    });
}]);

// Récupère les résultats
$results = $query->all();

// Affiche 29
echo $results->first()->auteur->age;
```

Comme vu précédemment, les formateurs attachés aux constructeurs de requête associées sont limités pour agir seulement sur les données de l'association. CakePHP va s'assurer que les valeurs calculées soient insérées dans la bonne entity.

Conditions Avancées

Le constructeur de requête facilite la construction de clauses `where` complexes. Les conditions groupées peuvent être exprimées en fournissant une combinaison de `where()` et d'objets représentant une expression. Pour les requêtes simples, vous pouvez construire des conditions en utilisant un tableau de conditions :

```
$query = $articles->find()
    ->where([
        'auteur_id' => 3,
        'OR' => [['nombre_de_vues' => 2], ['nombre_de_vues' => 3]],
    ]);
```

Ce qui précède générerait le code SQL :

```
SELECT * FROM articles WHERE auteur_id = 3 AND (nombre_de_vues = 2 OR nombre_de_vues = 3)
```

Si vous préférez éviter des tableaux avec de nombreux niveaux imbriqués, vous pouvez utiliser `where()` avec une fonction de rappel pour construire vos requêtes. La fonction de rappel prend une QueryExpression, ce qui vous permet d'utiliser l'interface du constructeur d'expression pour construire des conditions complexes sans utiliser de tableaux. Par exemple :

```
$query = $articles->find()->where(function (QueryExpression $exp, Query $query) {
    // Utilisez add() pour ajouter des conditions multiples sur le même champ.
    $auteur = $query->newExpr()->or(['auteur_id' => 3])->add(['auteur_id' => 2]);
    $published = $query->newExpr()->and(['published' => true, 'nombre_de_vues' => 10]);

    return $exp->or([
        'promoted' => true,
        $query->newExpr()->and([$auteur, $published])
    ]);
});
```

Ce qui précède générerait le code SQL :

```
SELECT *
FROM articles
WHERE (
    (
        (auteur_id = 2 OR auteur_id = 3)
        AND
        (published = 1 AND nombre_de_vues = 10)
    )
    OR promoted = 1
)
```

La `QueryExpression` passée à la fonction de rappel vous permet d'utiliser à la fois les **combinators** et les **conditions** pour construire votre expression globale.

Combinators
: Ils créent de nouveaux objets `QueryExpression` et définissent la façon dont les conditions ajoutées à cette expression sont combinées. - `and()` crée un nouvel objet expression qui combine toutes les conditions avec `AND`. - `or()` crée un nouvel objet expression qui combine toutes les conditions avec `OR`.

Conditions
: Elles sont ajoutées à l'expression et combinées automatiquement selon le combinator qui a été utilisé.

La `QueryExpression` passée à la fonction de rappel est par défaut `and()` :

```
$query = $articles->find()
    ->where(function (QueryExpression $exp) {
        return $exp
            ->eq('auteur_id', 2)
            ->eq('published', true)
            ->notEq('spam', true)
            ->gt('nombre_de_vues', 10);
    });
```

Puisque nous avons commencé à utiliser `where()`, nous n'avons pas besoin d'appeler `and()`, puisqu'elle est appelée implicitement. Le code ci-dessus montre quelques nouvelles méthodes de conditions combinées avec `AND`. Le code SQL résultant serait :

```
SELECT *
FROM articles
WHERE (
auteur_id = 2
AND published = 1
AND spam != 1
AND nombre_de_vues > 10)
```

Cependant, si nous souhaitons utiliser les deux conditions `AND` & `OR`, nous pouvons faire ceci :

```
$query = $articles->find()
    ->where(function (QueryExpression $exp) {
        $orConditions = $exp->or(['auteur_id' => 2])
            ->eq('auteur_id', 5);

        return $exp
            ->add($orConditions)
            ->eq('published', true)
            ->gte('nombre_de_vues', 10);
    });
```

Ce qui générerait le code SQL suivant :

```
SELECT *
FROM articles
WHERE (
    (auteur_id = 2 OR auteur_id = 5)
    AND published = 1
    AND nombre_de_vues > 10
)
```

Les **combinators** vous autorisent aussi à passer une fonction de rappel, qui prend en paramètre un nouvel objet expression, si vous voulez scinder le chaînage des méthodes :

```
$query = $articles->find()
    ->where(function (QueryExpression $exp) {
        $orConditions = $exp->or(function (QueryExpression $or) {
            return $or->eq('auteur_id', 2)
                ->eq('auteur_id', 5);
        });

        return $exp
            ->not($orConditions)
            ->lte('nombre_de_vues', 10);
    });
```

Vous pouvez faire une négation des sous-expressions en utilisant `not()` :

```
$query = $articles->find()
    ->where(function (QueryExpression $exp) {
        $orConditions = $exp->or(['author_id' => 2])
            ->eq('author_id', 5);
```

(suite sur la page suivante)

(suite de la page précédente)

```
        return $exp
            ->not($orConditions)
            ->lte('view_count', 10);
    });
```

Ce qui générerait le code SQL suivant :

```
SELECT *
FROM articles
WHERE (
    NOT (auteur_id = 2 OR auteur_id = 5)
    AND view_count <= 10
)
```

Il est aussi possible de construire les expressions en utilisant les fonctions SQL :

```
$query = $articles->find()
    ->where(function (QueryExpression $exp, Query $q) {
        $year = $q->func()->year([
            'created' => 'identifier'
        ]);

        return $exp
            ->gte($year, 2014)
            ->eq('published', true);
    });
```

Ce qui générerait le code SQL suivant :

```
SELECT *
FROM articles
WHERE (
    YEAR(created) >= 2014
    AND published = 1
)
```

Quand vous utilisez les objets expression, vous pouvez utiliser les méthodes suivantes pour créer des conditions :

— `eq()` Crée une condition d'égalité :

```
$query = $villes->find()
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->eq('population', '10000');
    });
# WHERE population = 10000
```

— `notEq()` Crée une condition d'inégalité :

```
$query = $villes->find()
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->notEq('population', '10000');
    });
# WHERE population != 10000
```

— `like()` Crée une condition en utilisant l'opérateur `LIKE` :

```
$query = $villes->find()
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->like('name', '%A%');
    });
# WHERE name LIKE "%A%"
```

— `notLike()` Crée une condition négative de type `LIKE` :

```
$query = $villes->find()
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->notLike('name', '%A%');
    });
# WHERE name NOT LIKE "%A%"
```

— `in()` Crée une condition en utilisant `IN` :

```
$query = $villes->find()
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->in('pays_id', ['AFG', 'USA', 'EST']);
    });
# WHERE pays_id IN ('AFG', 'USA', 'EST')
```

— `notIn()` Crée une condition négative en utilisant `IN` :

```
$query = $villes->find()
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->notIn('pays_id', ['AFG', 'USA', 'EST']);
    });
# WHERE pays_id NOT IN ('AFG', 'USA', 'EST')
```

— `gt()` Crée une condition > :

```
$query = $villes->find()
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->gt('population', '10000');
    });
# WHERE population > 10000
```

— `gte()` Crée une condition >= :

```
$query = $villes->find()
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->gte('population', '10000');
    });
# WHERE population >= 10000
```

— `lt()` Crée une condition < :

```
$query = $villes->find()
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->lt('population', '10000');
    });
# WHERE population < 10000
```

— `lte()` Crée une condition <= :

```
$query = $villes->find()
    ->where(function (QueryExpression $exp, Query $q) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        return $exp->lte('population', '10000');
    });
# WHERE population <= 10000
```

— `isNull()` Crée une condition `IS NULL` :

```
$query = $villes->find()
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->isNull('population');
    });
# WHERE (population) IS NULL
```

— `isNotNull()` Crée une condition négative `IS NULL` :

```
$query = $villes->find()
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->isNotNull('population');
    });
# WHERE (population) IS NOT NULL
```

— `between()` Crée une condition `BETWEEN` :

```
$query = $villes->find()
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->between('population', 999, 5000000);
    });
# WHERE population BETWEEN 999 AND 5000000,
```

— `exists()` Crée une condition en utilisant `EXISTS` :

```
$subquery = $villes->find()
    ->select(['id'])
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->equalFields('pays.id', 'villes.pays_id');
    })
  ->andWhere(['population >', 5000000]);

$query = $pays->find()
    ->where(function (QueryExpression $exp, Query $q) use ($subquery) {
        return $exp->exists($subquery);
    });
# WHERE EXISTS (SELECT id FROM villes WHERE pays.id = villes.pays_id AND␣
↪population > 5000000)
```

— `notExists()` Crée une condition négative en utilisant `EXISTS` :

```
$subquery = $villes->find()
    ->select(['id'])
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->equalFields('pays.id', 'villes.pays_id');
    })
  ->andWhere(['population >', 5000000]);

$query = $pays->find()
    ->where(function (QueryExpression $exp, Query $q) use ($subquery) {
        return $exp->notExists($subquery);
```

(suite sur la page suivante)

(suite de la page précédente)

```
    });
# WHERE NOT EXISTS (SELECT id FROM villes WHERE pays.id = villes.pays_id AND␣
↪population > 5000000)
```

Les objets expression devraient couvrir la plupart des fonctions et expressions communément utilisées. Si vous vous sentez incapable de créer les conditions voulues avec des expressions, vous vous sentirez peut-être capable d'utiliser `bind()` pour lier manuellement les paramètres aux conditions :

```
$query = $villes->find()
    ->where([
        'date_debut BETWEEN :debut AND :fin'
    ])
    ->bind(':debut', '2014-01-01', 'date')
    ->bind(':fin',   '2014-12-31', 'date');
```

Dans les cas où vous ne pouvez ou ne voulez pas utiliser les méthodes du constructeur pour créer les conditions que vous voulez, vous pouvez utiliser du code SQL dans des clauses where :

```
// Compare deux champs l'un avec l'autre
$query->where(['Categories.parent_id != Parents.id']);
```

Avertissement

Les noms de champs utilisés dans les expressions et le code SQL ne doivent **jamais** contenir de contenu non fiable, sinon vous ouvrez la porte à une injection SQL. Référez-vous à la section *Utiliser les Fonctions SQL* pour savoir comment inclure des données non fiables de manière sécurisée dans vos appels de fonctions.

Utiliser des Identificateurs dans des Expressions

Quand vous avez besoin de faire référence à une colonne ou à un identificateur SQL dans vos requêtes, vous pouvez utiliser la méthode `identifier()` :

```
$query = $pays->find();
$query->select([
        'year' => $query->func()->year([$query->identifier('created')])
    ])
    ->where(function ($exp, $query) {
        return $exp->gt('population', 100000);
    });
```

Avertissement

Pour prévenir les injections SQL, les expressions Identifier ne doivent jamais contenir de données non fiables.

Créer automatiquement des Clauses IN

Quand vous construisez des requêtes en utilisant l'ORM, vous n'avez généralement pas besoin d'indiquer les types de données des colonnes avec lesquelles vous interagissez, puisque CakePHP peut déduire les types en se basant sur les données du schéma. Si dans vos requêtes, vous souhaitez que CakePHP convertisse automatiquement l'égalité en comparaisons `IN`, vous devez indiquer les types de données des colonnes :

```
$query = $articles->find()
    ->where(['id' => $ids], ['id' => 'integer[]']);

// Ou bien incluez IN pour caster automatiquement en tableau
$query = $articles->find()
    ->where(['id IN' => $ids]);
```

Ceci va créer automatiquement `id IN (...)` plutôt que `id = ?`. Cela peut être utile quand vous ne savez pas si vous allez récupérer un scalaire ou un tableau de paramètres. Le suffixe `[]` sur un nom de type de données indique au constructeur de requête que vous souhaitez que les données soient gérées en tableau. Si les données ne sont pas un tableau, elles vont d'abord être converties en tableau. Après cela, chaque valeur dans le tableau va être convertie en utilisant le *système de types*. Cela fonctionne aussi avec des types complexes. Par exemple, vous pourriez prendre une liste d'objets DateTime en utilisant :

```
$query = $articles->find()
    ->where(['post_date' => $dates], ['post_date' => 'date[]']);
```

Création Automatique de IS NULL

Quand une valeur dans une condition est supposée être `null`, ou bien une valeur quelconque, vous pouvez utiliser l'opérateur `IS` pour créer automatiquement la bonne expression :

```
$query = $categories->find()
    ->where(['parent_id IS' => $parentId]);
```

Ce code va créer `parent_id` = :c1` ou `parent_id IS NULL` selon le type de `$parentId`.

Création Automatique de IS NOT NULL

Quand une valeur dans une condition est supposée ne pas être `null`, ou ne pas être une valeur quelconque, vous pouvez utiliser l'opérateur `IS NOT` pour créer automatiquement la bonne expression :

```
$query = $categories->find()
    ->where(['parent_id IS NOT' => $parentId]);
```

Ce code va créer `parent_id` != :c1` ou `parent_id IS NOT NULL` selon le type de `$parentId`.

Raw Expressions

Si le constructeur de requêtes ne vous permet pas de construire le code SQL que vous souhaitez, vous pouvez utiliser les objets `Expression` pour ajouter directement du code SQL à vos requêtes :

```
$query = $articles->find();
$expr = $query->newExpr()->add('1 + 1');
$query->select(['deux' => $expr]);
```

Les objets `Expression` peuvent être utilisés avec n'importe quelle méthode du constructeur de requêtes comme `where()`, `limit()`, `group()`, `select()` et bien d'autres.

> **Avertissement**
>
> Les objets `Expression` vous rendent vulnérable aux injections SQL. Vous devez absolument éviter d'utiliser dans les expressions des données venant d'une source non fiable.

Récupérer les Résultats

Une fois que vous aurez fait votre requête, vous voudrez récupérer les résultats. Il y a plusieurs façons de faire :

```
// Itérer la requête
foreach ($query as $row) {
    // Faire le boulot.
}

// Récupérer les résultats
$results = $query->all();
```

Vous pouvez utiliser toutes les méthodes des *collections* sur vos objets query pour retraiter ou transformer les résultats :

```
// Utilise une des méthodes collection.
$ids = $query->map(function ($row) {
    return $row->id;
});

$maxAge = $query->max(function ($max) {
    return $max->age;
});
```

Vous pouvez utiliser `first()` ou `firstOrFail()` pour récupérer un enregistrement unique. Ces méthodes vont modifier la requête en ajoutant une clause `LIMIT 1` :

```
// Récupère uniquement la première ligne
$row = $query->first();

// Récupère la première ligne ou une exception.
$row = $query->firstOrFail();
```

Retourner le Nombre Total d'Enregistrements

Il est possible d'obtenir le nombre total de lignes trouvées pour un ensemble de conditions en utilisant un seul objet query :

```
$total = $articles->find()->where(['is_active' => true])->count();
```

La méthode `count()` va ignorer les clauses `limit`, `offset` et `page`, donc ce qui suit va retourner les mêmes résultats :

```
$total = $articles->find()->where(['is_active' => true])->limit(10)->count();
```

C'est utile quand vous avez besoin de connaître le nombre de résultats à l'avance, sans avoir à construire un autre objet `Query`. De la même façon, tous les formatages de résultats et les routines map-reduce sont ignorées quand vous utilisez la méthode `count()`.

De plus, il est possible d'obtenir le nombre total de lignes pour une requête contenant des clauses group by sans avoir à réécrire la requête de quelque façon que ce soit. Par exemple, considérons la requête qui permet de récupérer les ids d'article et leurs nombres de commentaires :

```
$query = $articles->find();
$query->select(['Articles.id', $query->func()->count('Comments.id')])
    ->matching('Comments')
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ->group(['Articles.id']);
$total = $query->count();
```

Après avoir compté, la requête peut toujours être utilisée pour récupérer les enregistrements associés :

```
$list = $query->all();
```

Vous voudrez parfois fournir une méthode alternative pour compter le nombre total d'enregistrements d'une requête. Un cas d'utilisation courante de cette fonctionnalité est de fournir une valeur mise en cache ou une estimation du nombre total de lignes, ou de modifier la requête pour retirer les parties coûteuses non nécessaires comme les left joins. Ceci devient particulièrement pratique quand vous utilisez le système de pagination intégré à CakePHP qui appelle la méthode `count()` :

```
$query = $query->where(['is_active' => true])->counter(function ($query) {
    return 100000;
});
$query->count(); // Retourne 100000
```

Dans l'exemple ci-dessus, quand le component Pagination appellera la méthode count, il recevra le nombre de lignes estimé codé en dur.

Mettre en Cache les Résultats Chargés

Quand vous récupérez des entities qui ne changent pas souvent, vous voudrez peut-être mettre en cache les résultats. La classe `Query` facilite cela :

```
$query->cache('recent_articles');
```

va activer la mise en cache des résultats de la requête. Si un seul argument est fourni à `cache()` alors c'est la configuration du cache "default" qui sera utilisée. Vous pouvez contrôler la configuration de cache à utiliser avec le deuxième paramètre :

```
// Nom de la config.
$query->cache('recent_articles', 'dbResults');

// Instance de CacheEngine
$query->cache('recent_articles', $memcache);
```

En plus de supporter les clés statiques, la méthode `cache()` accepte une fonction pour générer la clé. La fonction que vous lui donnez va recevoir la requête en argument. Vous pouvez ensuite lire les aspects de la requête pour générer dynamiquement la clé de mise en cache :

```
// Génère une clé basée sur un checksum simple
// de la clause where de la requête
$query->cache(function ($q) {
    return 'articles-' . md5(serialize($q->clause('where')));
});
```

La méthode cache facilite l'ajout des résultats mis en cache à vos finders personnalisés ou à travers des écouteurs d'évènements.

Quand les résultats d'une requête mise en cache sont récupérés, voici ce qui va se passer :

1. Si la requête a des résultats, ceux-ci vont être retournés.

2. La clé du cache va être déterminée et les données du cache vont être lues. Si les données du cache ne sont pas vides, ces résultats vont être retournés.
3. Si le cache est manquant, la requête sera exécutée, l'événement `Model.beforeFind` sera déclenché, et un nouveau `ResultSet` sera créé. Ce `ResultSet` sera écrit dans le cache et retourné.

Note

Vous ne pouvez pas mettre en cache un résultat de requête streaming.

Charger des Associations

Le constructeur peut vous aider à récupérer les données de plusieurs tables à la fois avec un minimum de requêtes. Pour pouvoir récupérer les données associées, vous devez d'abord configurer les associations entre les tables comme décrit dans la section *Associations - Lier les Tables Ensemble*. Cette technique de requêtes combinées pour récupérer les données associées à partir d'autres tables est appelé **eager loading**.

L'eager loading aide à éviter la plupart des problèmes potentiels de performance qui entourent le lazy loading dans un ORM. Les requêtes générées par eager loading peuvent davantage tirer parti des jointures, ce qui permet de créer des requêtes plus efficaces. Dans CakePHP, vous utilisez la méthode "contain" pour indiquer quelles associations doivent être chargées en eager :

```
// Dans un controller ou une méthode de table.

// En option du find()
$query = $articles->find('all', ['contain' => ['Authors', 'Comments']]);

// En méthode sur un objet query
$query = $articles->find('all');
$query->contain(['Authors', 'Comments']);
```

Ceci va charger les auteurs et commentaires liés à chaque article du *result set*. Vous pouvez charger des associations imbriquées en utilisant les tableaux imbriqués pour définir les associations à charger :

```
$query = $articles->find()->contain([
    'Authors' => ['Addresses'], 'Comments' => ['Authors']
]);
```

Au choix, vous pouvez aussi exprimer des associations imbriquées en utilisant la notation par points :

```
$query = $articles->find()->contain([
    'Authors.Addresses',
    'Comments.Authors'
]);
```

Vous pouvez charger les associations en eager aussi profondément que vous le souhaitez :

```
$query = $produits->find()->contain([
    'Shops.Cities.Countries',
    'Shops.Managers'
]);
```

Vous pouvez sélectionner des champs de toutes les associations en utilisant plusieurs appels à `contain()` :

```
$query = $this->find()->select([
    'Realestates.id',
    'Realestates.title',
    'Realestates.description'
])
->contain([
    'RealestatesAttributes' => [
        'Attributes' => [
            'fields' => [
                // Les champs dépendant d'un alias doivent inclure le préfixe
                // du modèle dans contain() pour être mappés correctement.
                'Attributes__name' => 'attr_name'
            ]
        ]
    ]
])
->contain([
    'RealestatesAttributes' => [
        'fields' => [
            'RealestatesAttributes.realestate_id',
            'RealestatesAttributes.value'
        ]
    ]
])
->where($condition);
```

Si vous avez besoin de réinitialiser les *contain* sur une requête, vous pouvez définir le second argument à `true` :

```
$query = $articles->find();
$query->contain(['Authors', 'Comments'], true);
```

Note

Les noms d'association dans les appels à `contain()` doivent respecter la casse (majuscules/minuscules) avec lequelle votre association a été définie, et non pas selon le nom de la propriété utilisée pour accéder aux données associées depuis l'entity. Par exemple, si vous avez déclaré une association par `belongsTo('Users')`, alors vous devez utiliser `contain('Users')` et pas `contain('users')` ni `contain('user')`.

Passer des Conditions à Contain

Avec l'utilisation de `contain()`, vous pouvez restreindre les données retournées par les associations et les filtrer par conditions. Pour spécifier des conditions, passez une fonction anonyme qui reçoit en premier argument la query, de type `\Cake\ORM\Query` :

```
// Dans un controller ou une méthode de table.
$query = $articles->find()->contain('Comments', function (Query $q) {
    return $q
        ->select(['contenu', 'author_id'])
        ->where(['Comments.approved' => true]);
});
```

Cela fonctionne aussi pour la pagination au niveau du Controller :

```
$this->paginate['contain'] = [
    'Comments' => function (Query $query) {
        return $query->select(['body', 'author_id'])
        ->where(['Comments.approved' => true]);
    }
];
```

Avertissement

Si vous constatez qu'il manque des entités associées, vérifiez que les champs de clés étrangères sont bien sélectionnés dans la requête. Sans les clés étrangères, l'ORM ne peut pas retrouver les lignes correspondantes.

Il est aussi possible de restreindre les associations imbriquées en utilisant la notation par point :

```
$query = $articles->find()->contain([
    'Comments',
    'Authors.Profiles' => function (Query $q) {
        return $q->where(['Profiles.is_published' => true]);
    }
]);
```

Dans cet exemple, vous obtiendrez les auteurs même s'ils n'ont pas de profil publié. Pour ne récupérer que les auteurs avec un profil publié, utilisez *matching()*.

Si vous avez des finders personnalisés dans votre table associée, vous pouvez les utiliser à l'intérieur de `contain()` :

```
// Récupère tous les articles, mais récupère seulement les commentaires qui
// sont approuvés et populaires.
$query = $articles->find()->contain('Comments', function ($q) {
    return $q->find('approved')->find('popular');
});
```

Note

Pour les associations `BelongsTo` et `HasOne`, seules les clauses `where` et `select` sont utilisées lors du chargement par `contain()`. Avec `HasMany` et `BelongsToMany`, toutes les clauses sont valides, telles que `order()`.

Vous pouvez contrôler plus que les simples clauses utilisées par `contain()`. Si vous passez un tableau avec l'association, vous pouvez surcharger `foreignKey`, `joinType` et `strategy`. Reportez-vous à *Associations - Lier les Tables Ensemble* pour plus de détails sur la valeur par défaut et les options de chaque type d'association.

Vous pouvez passer `false` comme nouvelle valeur de `foreignKey` pour désactiver complètement les contraintes liées aux clés étrangères. Utilisez l'option `queryBuilder` pour personnaliser la requête quand vous passez un tableau :

```
$query = $articles->find()->contain([
    'Authors' => [
        'foreignKey' => false,
        'queryBuilder' => function (Query $q) {
            return $q->where(/* ... */); // Conditions complètes pour le filtrage
        }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ]
]);
```

Si vous avez limité les champs que vous chargez avec `select()` mais que vous souhaitez aussi charger les champs des associations avec contain, vous pouvez passer l'objet association à `select()` :

```
// Sélectionne id & title de articles, mais aussi tous les champs de Users.
$query = $articles->find()
    ->select(['id', 'title'])
    ->select($articles->Users)
    ->contain(['Users']);
```

Autre possibilité, si vous avez des associations multiples, vous pouvez utiliser `enableAutoFields()` :

```
// Sélectionne id & title de articles, mais tous les champs de
// Users, Comments et Tags.
$query->select(['id', 'title'])
    ->contain(['Comments', 'Tags'])
    ->enableAutoFields(true)
    ->contain(['Users' => function(Query $q) {
        return $q->autoFields(true);
    }]);
```

Trier les Associations Contain

Quand vous chargez des associations HasMany et BelongsToMany, vous pouvez utiliser l'option `sort` pour trier les données dans ces associations :

```
$query->contain([
    'Comments' => [
        'sort' => ['Comments.created' => 'DESC']
    ]
]);
```

Filtrer selon les Données Associées

Un cas de requête couramment utilisé avec les associations consiste à trouver les enregistrements qui correspondent à certaines données associées. Par exemple si vous avez une association "Articles belongsToMany Tags", vous voudrez probablement trouver les Articles qui portent le tag *CakePHP*. C'est extrêmement simple à faire avec l'ORM de CakePHP :

```
// Dans un controller ou une méthode de table.

$query = $articles->find();
$query->matching('Tags', function ($q) {
    return $q->where(['Tags.name' => 'CakePHP']);
});
```

Vous pouvez aussi appliquer cette stratégie aux associations HasMany. Par exemple si "Authors HasMany Articles", vous pouvez trouver tous les auteurs ayant publié un article récemment en écrivant ceci :

```
$query = $authors->find();
$query->matching('Articles', function ($q) {
    return $q->where(['Articles.created >=' => new DateTime('-10 days')]);
});
```

La syntaxe de `contain()`, qui doit déjà vous être familière, permet aussi de filtrer des associations imbriquées :

```
// Dans un controller ou une méthode de table.
$query = $produits->find()->matching(
    'Shops.Cities.Countries', function ($q) {
        return $q->where(['Countries.name' => 'Japon']);
    }
);

// Récupère les articles qui ont été commentés par 'markstory',
// en passant une variable.
// Utiliser la notation par points plutôt que des appels imbriqués à matching()
$username = 'markstory';
$query = $articles->find()->matching('Comments.Users', function ($q) use ($username) {
    return $q->where(['username' => $username]);
});
```

> **Note**
>
> Dans la mesure où cette fonction va créer un `INNER JOIN`, il serait judicieux d'utiliser `distinct` dans la requête. Sinon, vous risquez d'obtenir des doublons si les conditions posées ne l'excluent pas par principe. Dans notre exemple, cela peut être le cas si un utilisateur commente plusieurs fois le même article.

Les données des associations qui correspondent aux conditions (données *matchées*) seront disponibles dans l'attribut `_matchingData` des entities. Si vous utilisez à la fois `match` et `contain` sur la même association, vous pouvez vous attendre à avoir à la fois la propriété `_matchingData` et la propriété standard d'association dans vos résultats.

Utiliser innerJoinWith

Utiliser la fonction `matching()`, comme nous l'avons vu précédemment, va créer un `INNER JOIN` avec l'association spécifiée et va aussi charger les champs dans un ensemble de résultats.

Il peut arriver que vous veuillez utiliser `matching()` mais que vous n'êtes pas intéressé par le chargement des champs de l'association. Dans ce cas, vous pouvez utiliser `innerJoinWith()` :

```
$query = $articles->find();
$query->innerJoinWith('Tags', function ($q) {
    return $q->where(['Tags.name' => 'CakePHP']);
});
```

La méthode `innerJoinWith()` fonctionne de la même manière que `matching()`, ce qui signifie que vous pouvez utiliser la notation par points pour faire des jointures pour les associations imbriquées profondément :

```
$query = $products->find()->innerJoinWith(
    'Shops.Cities.Countries', function ($q) {
        return $q->where(['Countries.name' => 'Japon']);
    }
);
```

Si vous voulez à la fois poser des conditions sur certains champs de l'association et charger d'autres champs de cette même association, vous pouvez parfaitement combiner `innerJoinWith()` et `contain()`. L'exemple ci-dessous filtre les Articles qui ont des Tags spécifiques et charge ces Tags :

```
$filter = ['Tags.name' => 'CakePHP'];
$query = $articles->find()
    ->distinct($articles->getPrimaryKey())
    ->contain('Tags', function (Query $q) use ($filter) {
        return $q->where($filter);
    })
    ->innerJoinWith('Tags', function (Query $q) use ($filter) {
        return $q->where($filter);
    });
```

Note

Si vous utilisez `innerJoinWith()` et que vous voulez sélectionner des champs de cette association avec `select()`, vous devez utiliser un alias pour les noms des champs :

```
$query
    ->select(['country_name' => 'Countries.name'])
    ->innerJoinWith('Countries');
```

Sinon, vous verrez les données dans `_matchingData`, comme cela a été décrit ci-dessous à propos de `matching()`. C'est un angle mort de `matching()`, qui ne sait pas que vous avez sélectionné des champs.

Avertissement

Vous ne devez pas combiner `innerJoinWith()` and `matching()` pour la même association. Cela produirait de multiple requêtes `INNER JOIN` et ne réaliserait pas ce que vous en attendez.

Utiliser notMatching

L'opposé de `matching()` est `notMatching()`. Cette fonction va changer la requête pour qu'elle filtre les résultats qui n'ont pas de relation avec l'association spécifiée :

```
// Dans un controller ou une méthode de table.

$query = $articlesTable
    ->find()
    ->notMatching('Tags', function ($q) {
        return $q->where(['Tags.name' => 'ennuyeux']);
    });
```

L'exemple ci-dessus va trouver tous les articles qui n'ont pas été taggés avec le mot `ennuyeux`. Vous pouvez aussi utiliser cette méthode avec les associations HasMany. Vous pouvez, par exemple, trouver tous les auteurs qui n'ont publié aucun article dans les 10 derniers jours :

```
$query = $authorsTable
    ->find()
    ->notMatching('Articles', function ($q) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        return $q->where(['Articles.created >=' => new \DateTime('-10 days')]);
    });
```

Il est aussi possible d'utiliser cette méthode pour filtrer les enregistrements qui ne matchent pas des associations imbriquées. Par exemple, vous pouvez trouver les articles qui n'ont pas été commentés par un utilisateur précis :

```
$query = $articlesTable
    ->find()
    ->notMatching('Comments.Users', function ($q) {
        return $q->where(['username' => 'jose']);
    });
```

Puisque les articles n'ayant absolument aucun commentaire satisfont aussi cette condition, vous aurez intérêt à combiner `matching()` et `notMatching()` dans cette requête. L'exemple suivant recherchera les articles ayant au moins un commentaire, mais non commentés par un utilisateur précis :

```
$query = $articlesTable
    ->find()
    ->notMatching('Comments.Users', function ($q) {
        return $q->where(['username' => 'jose']);
    })
    ->matching('Comments');
```

Note

Comme `notMatching()` va créer un `LEFT JOIN`, vous pouvez envisager d'appeler `distinct` sur la requête pour éviter d'obtenir des lignes dupliquées.

Gardez à l'esprit que le contraire de la fonction `matching()`, `notMatching()`, ne va pas ajouter de données à la propriété `_matchingData` dans les résultats.

Utiliser leftJoinWith

Dans certaines situations, vous aurez à calculer un résultat à partir d'une association, sans avoir à charger tous ses enregistrements. Par exemple, si vous voulez charger le nombre total de commentaires d'un article, en parallèle des données de l'article, vous pouvez utiliser la fonction `leftJoinWith()` :

```
$query = $articlesTable->find();
$query->select(['total_comments' => $query->func()->count('Comments.id')])
    ->leftJoinWith('Comments')
    ->group(['Articles.id'])
    ->enableAutoFields(true);
```

Le résultat de cette requête contiendra les données de l'article et la propriété `total_comments` pour chacun d'eux.

`leftJoinWith()` peut aussi être utilisée avec des associations imbriquées. C'est utile par exemple pour rechercher, pour chaque auteur, le nombre d'articles taggés avec un certain mot :

```
$query = $authorsTable
    ->find()
    ->select(['total_articles' => $query->func()->count('Articles.id')])
    ->leftJoinWith('Articles.Tags', function ($q) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        return $q->where(['Tags.name' => 'redoutable']);
    })
    ->group(['Authors.id'])
    ->enableAutoFields(true);
```

Cette fonction ne va charger aucune colonne des associations spécifiées dans les résultats.

Ajouter des Jointures

En plus de charger les données liées avec `contain()`, vous pouvez aussi ajouter des jointures supplémentaires avec le constructeur de requête :

```
$query = $articles->find()
    ->join([
        'table' => 'comments',
        'alias' => 'c',
        'type' => 'LEFT',
        'conditions' => 'c.article_id = articles.id',
    ]);
```

Vous pouvez ajouter plusieurs jointures en même temps en passant un tableau associatif avec plusieurs `join` :

```
$query = $articles->find()
    ->join([
        'c' => [
            'table' => 'comments',
            'type' => 'LEFT',
            'conditions' => 'c.article_id = articles.id',
        ],
        'u' => [
            'table' => 'users',
            'type' => 'INNER',
            'conditions' => 'u.id = articles.user_id',
        ]
    ]);
```

Comme vu précédemment, lors de l'ajout de `join`, l'alias peut être la clé du tableau externe. Les conditions `join` peuvent être aussi exprimées en tableau de conditions :

```
$query = $articles->find()
    ->join([
        'c' => [
            'table' => 'comments',
            'type' => 'LEFT',
            'conditions' => [
                'c.created >' => new DateTime('-5 days'),
                'c.moderated' => true,
                'c.article_id = articles.id'
            ]
        ],
    ], ['c.created' => 'datetime', 'c.moderated' => 'boolean']);
```

Lors de la création de `join` à la main, et l'utilisation de conditions à partir d'un tableau, vous devez fournir les types de données pour chaque colonne dans les conditions du `join`. En fournissant les types de données pour les conditions de

`join`, l'ORM peut convertir correctement les types de données en code SQL. En plus de `join()` vous pouvez utiliser `rightJoin()`, `leftJoin()` et `innerJoin()` pour créer les jointures :

```
// Jointure avec un alias et des conditions littérales
$query = $articles->find();
$query->leftJoin(
    ['Auteurs' => 'auteurs'],
    ['Auteurs.id = Articles.auteur_id']);

// Jointure avec un alias, tableau de conditions, et types
$query = $articles->find();
$query->innerJoin(
    ['Auteurs' => 'auteurs'],
    [
    'Auteurs.promoted' => true,
    'Auteurs.created' => new DateTime('-5 days'),
    'Auteurs.id = Articles.auteur_id'
    ],
    ['Auteurs.promoted' => 'boolean', 'Auteurs.created' => 'datetime']);
```

Notez que si vous définissez l'option `quoteIdentifiers` à `true` quand vous configurez votre `Connection`, les conditions de jointure entre deux champs de tables doivent être définies de cette manière :

```
$query = $articles->find()
    ->join([
        'c' => [
            'table' => 'comments',
            'type' => 'LEFT',
            'conditions' => [
                'c.article_id' => new \Cake\Database\Expression\IdentifierExpression(
↪'articles.id')
            ]
        ],
    ]);
```

Cela permet de s'assurer que tous les `identifiers` seront bien quotés dans la requête générée, permettant d'éviter des erreurs avec certains drivers (PostgreSQL notamment).

Insérer des Données

Contrairement aux exemples précédents, vous ne devez pas utiliser `find()` pour créer des requêtes d'insertion. À la place, créez un nouvel objet `Query` en utilisant `query()` :

```
$query = $articles->query();
$query->insert(['titre', 'corps'])
    ->values([
        'titre' => 'Premier post',
        'corps' => 'Un corps de texte'
    ])
    ->execute();
```

Pour insérer plusieurs lignes en une seule requête, vous pouvez chaîner la méthode `values()` autant de fois que nécessaire :

```
$query = $articles->query();
$query->insert(['titre', 'corps'])
    ->values([
        'titre' => 'Premier post',
        'corps' => 'Un corps de texte'
    ])
    ->values([
        'titre' => 'Second post',
        'corps' => 'Un autre corps de texte'
    ])
    ->execute();
```

Généralement, il est plus facile d'insérer des données en utilisant les entities et `save()`. En composant des requêtes `SELECT` et `INSERT` ensemble, vous pouvez créer des requêtes du style `INSERT INTO ... SELECT` :

```
$select = $articles->find()
    ->select(['titre', 'corps', 'published'])
    ->where(['id' => 3]);

$query = $articles->query()
    ->insert(['titre', 'corps', 'published'])
    ->values($select)
    ->execute();
```

Note

L'insertion d'enregistrements avec le constructeur de requêtes ne va pas déclencher les événements comme `Model.afterSave`. À la place, vous pouvez utiliser *l'ORM pour sauvegarder les données*.

Mettre à Jour les Données

Comme pour les requêtes d'insertion, vous ne devez pas utiliser `find()` pour créer des requêtes de mise à jour. A la place, créez un nouvel objet `Query` en utilisant `query()` :

```
$query = $articles->query();
$query->update()
    ->set(['published' => true])
    ->where(['id' => $id])
    ->execute();
```

Généralement, il est plus facile de mettre à jour des données en utilisant des entities et `patchEntity()`.

Note

La mise à jour d'enregistrements avec le constructeur de requêtes ne va pas déclencher les événements comme `Model.afterSave`. À la place, vous pouvez utiliser *l'ORM pour sauvegarder des données*.

Suppression des Données

Comme pour les requêtes d'insertion, vous ne devez pas utiliser `find()` pour créer des requêtes de suppression. À la place, créez un nouvel objet de requête en utilisant `query()` :

```
$query = $articles->query();
$query->delete()
    ->where(['id' => $id])
    ->execute();
```

Généralement, il est plus facile de supprimer les données en utilisant les entities et *delete()*.

Prévention des Injections SQL

Bien que l'ORM et les couches d'abstraction de base de données vous prémunisse de la plupart des problèmes relatifs aux injections SQL, il est toujours possible de vous rendre vulnérable en raison d'une utilisation inappropriée.

Lorsque vous utilisez des tableaux de conditions, la clé (la partie à gauche) aussi bien que les valeurs simples ne doivent pas contenir de données utilisateur :

```
$query->where([
    // Utiliser cette clé est dangereux car elle sera insérée telle quelle
    // dans la requête générée
    $userData => $value,

    // Même chose pour les valeurs simples : il est
    // dangereux de les utiliser avec des données utilisateur
    $userData,
    "MATCH (comment) AGAINST ($userData)",
    'created < NOW() - ' . $userData
]);
```

Lorsque vous utilisez le constructeur d'expressions, les noms de colonnes ne doivent pas contenir de données provenant d'utilisateurs :

```
$query->where(function (QueryExpression $exp) use ($userData, $values) {
    // Dans toutes les expressions, les noms de colonnes ne sont pas sûrs.
    return $exp->in($userData, $values);
});
```

Lorsque vous construisez des expressions de fonctions, les noms de fonctions ne doivent jamais contenir de données provenant d'utilisateurs :

```
// Non sécurisé.
$query->func()->{$userData}($arg1);

// L'utilisation d'un tableau de données utilisateurs
// dans une fonction n'est également pas sécurisée
$query->func()->coalesce($userData);
```

Les expressions brutes ne sont jamais sécurisées :

```
$expr = $query->newExpr()->add($userData);
$query->select(['deux' => $expr]);
```

Lier les Valeurs (Binding)

Il est possible de vous prémunir contre de nombreuses situations à risque en utilisant les *bindings*. De la même manière que vous pouvez *lier des valeurs pour les requêtes préparées*, des valeurs peuvent être liées aux requêtes en utilisant la méthode `Cake\Database\Query::bind()`.

L'exemple ci-dessous est une alternative sûre à la version donnée plus haut, qui était vulnérable à une injection SQL :

```
$query
    ->where([
        'MATCH (comment) AGAINST (:userData)',
        'created < NOW() - :moreUserData'
    ])
    ->bind(':userData', $userData, 'string')
    ->bind(':moreUserData', $moreUserData, 'datetime');
```

Note

Contrairement à `Cake\Database\StatementInterface::bindValue()`, `Query::bind()` a besoin que vous passiez les « placeholders » en incluant les deux-points (`:`) !

Requêtes Plus Complexes

Si votre application a besoin de recourir à des requêtes plus complexes, vous pouvez en écrire de nombreuses manières avec le constructeur de requêtes ORM.

Unions

Les unions sont créées en composant une ou plusieurs requêtes select ensemble :

```
$inReview = $articles->find()
    ->where(['a_relire' => true]);

$unpublished = $articles->find()
    ->where(['published' => false]);

$unpublished->union($inReview);
```

Vous pouvez créer les requêtes `UNION ALL` en utilisant la méthode `unionAll()` :

```
$inReview = $articles->find()
    ->where(['a_relire' => true]);

$unpublished = $articles->find()
    ->where(['published' => false]);

$unpublished->unionAll($inReview);
```

Sous-Requêtes

Les sous-requêtes vous permettent de combiner des requêtes et de construire des conditions et des résultats basés sur les résultats d'autres requêtes :

```
$matchingComment = $articles->getAssociation('Comments')->find()
    ->select(['article_id'])
    ->distinct()
    ->where(['comment LIKE' => '%CakePHP%']);

$query = $articles->find()
    ->where(['id IN' => $matchingComment]);
```

Les sous-requêtes sont acceptées partout où une expression de requête peut être utilisée. Par exemple, dans les méthodes `select()` et `join()`. L'exemple ci-dessus utilise un objet `Orm\Query` standard qui générera des alias. Ces alias peuvent complexifier le référencement des résultats dans la requête englobante. Depuis 4.2.0 vous pouvez utiliser `Table::subquery()` pour créer une instance de requête spécialisée qui ne générera pas d'alias :

```
$comments = $articles->getAssociation('Comments')->getTarget();

$matchingComment = $comments->subquery()
    ->select(['article_id'])
    ->distinct()
    ->where(['comment LIKE' => '%CakePHP%']);

$query = $articles->find()
    ->where(['id IN' => $matchingComment]);
```

Ajouter des Clauses de Verrouillage

La plupart des fournisseurs de bases de données relationnelles supportent la pose de verrous pendant les opérations SELECT. Pour cela, vous pouvez utiliser la méthode `epilog()` :

```
// Dans MySQL
$query->epilog('FOR UPDATE');
```

La méthode `epilog()` vous permet d'ajouter du SQL brut à la fin des requêtes. Vous ne devez jamais insérer des données utilisateur dans `epilog()`.

Fonctions de Fenêtrage (Window Functions)

Les fonctions de fenêtrage (*window functions*) vous permettent d'effectuer des calculs en utilisant des lignes relatives à la ligne courante. Elles sont couramment utilisées pour calculer des totaux ou des positions dans des sous-ensembles de lignes dans une requête. Par exemple si vous nous voulons connaître la date du plus ancien et du plus récent commentaires sur chaque article, nous pouvons utiliser des fonctions de fenêtrage :

```
$query = $articles->find();
$query->select([
    'Articles.id',
    'Articles.titre',
    'Articles.user_id'
    'premier_commentaire' => $query->func()
        ->min('Comments.created')
        ->partition('Comments.article_id'),
    'dernier_commentaire' => $query->func()
        ->max('Comments.created')
        ->partition('Comments.article_id'),
```

(suite sur la page suivante)

(suite de la page précédente)

```
])
->innerJoinWith('Comments');
```

Ce code générerait du SQL ressemblant à :

```
SELECT
    Articles.id,
    Articles.titre,
    Articles.user_id
    MIN(Comments.created) OVER (PARTITION BY Comments.article_id) AS premier_commentaire,
    MAX(Comments.created) OVER (PARTITION BY Comments.article_id) AS dernier_commentaire,
FROM articles AS Articles
INNER JOIN comments AS Comments
```

Les expressions de fenêtrage peuvent s'appliquer à la plupart des fonctions d'agrégation. Toutes les fonctions d'agrégation pour lesquelle cake propose une abstraction avec un wrapper dans `FunctionsBuilder` retourneront une `AggregateExpression` qui vous permet d'y rattacher une expression de fenêtrage. Vous pouvez créer des fonctions d'agrégation personnalisées avec `FunctionsBuilder::aggregate()`.

Ce sont les fonctionnalités de fenêtrage les plus communément supportées. La plupart des fonctionnalités sont fournies par `AggregateExpresion`, mais assurez-vous de suivre la documentation de votre base de données sur leur utilisation et leurs restrictions.

- `order($fields)` Trie les groupes agrégés de la même façon que ORDER BY.
- `partition($expressions)` Ajoute une ou plusieurs partitions à la fenêtre à partir des noms de colonnes.
- `rows($start, $end)` Définit un offset de lignes qui précèdent et/ou suivent la ligne en cours et qui devraient être incluses dans la fonction d'agrégation.
- `range($start, $end)` Définit une plage de valeurs de lignes qui précèdent et/ou suivent la ligne en cours et qui devraient être incluses dans la fonction d'agrégation. Cela évalue les valeurs sur la base du champ de `order()`.

Si vous avez besoin d'utiliser une même expression de fenêtrage à plusieurs reprises, vous pouvez créer des fenêtres nommées en utilisant la méthode `window()` :

```
$query = $articles->find();

// Définit une fenêtre nommée
$query->window('article_concerne', function ($window, $query) {
    $window->partition('Comments.article_id');

    return $window;
});

$query->select([
    'Articles.id',
    'Articles.titre',
    'Articles.user_id'
    'premier_commentaire' => $query->func()
        ->min('Comments.created')
        ->over('article_concerne'),
    'dernier_commentaire' => $query->func()
        ->max('Comments.created')
        ->over('article_concerne'),
]);
```

Common Table Expressions

Les *Common Table Expressions* ou CTE sont utiles pour construire des requêtes dans lesquelles vous devez rassembler les résultats de plusieurs petites requêtes. Ils peuvent avoir la même utilité que les vues de base de données ou que les résultats de sous-requêtes. Les *Common Table Expression* se différencient des tables dérivées et des vues sur plusieurs points :

1. Contrairement aux vues, vous n'avez pas besoin de maintenir un schéma pour les CTE. Le schéma est basé implicitement sur le result set de l'expression.
2. Vous pouvez faire plusieurs références aux résultats d'une CTE sans dégradation de performance, contrairement aux jointures de sous-requêtes.

À titre d'exemple, récupérons une liste de clients et le nombre de commandes qu'ils ont passées. En SQL nous utiliserions :

```
WITH commandes_par_client AS (
    SELECT COUNT(*) AS nb_commandes, client_id FROM commandes GROUP BY client_id
)
SELECT nom, commandes_par_client.nb_commandes
FROM clients
INNER JOIN commandes_par_client ON commandes_par_client.client_id = clients.id
```

Pour construire cette requête avec le constructeur de requêtes, nous utiliserions :

```
// Démarrer la requête finale
$query = $this->Clients->find();

// Attacher une common table expression
$query->with(function ($cte) {
    // Créer une sous-requête à utiliser dans notre CTE
    $q = $this->Commandes->subquery();
    $q->select([
        'nb_commandes' => $q->func()->count('*'),
        'client_id'
    ])
    ->group('client_id');

    // Attacher la nouvelle requête à notre CTE
    return $cte
        ->name('commandes_par_client')
        ->query($q);
});

// Terminer la construction de la requête finale
$query->select([
    'name',
    'nb_commandes' => 'commandes_par_client.nb_commandes',
])
->join([
    // Définir la jointure avec notre CTE
    'commandes_par_client' => [
        'table' => 'commandes_par_client',
        'conditions' => 'commandes_par_client.client_id = Clients.id'
    ]
]);
```

Exécuter des Requêtes Complexes

Bien que le constructeur de requêtes permette la construction de la plupart des requêtes, les requêtes très complexes peuvent être fastidieuses et compliquées à construire. Vous pourrez être tenté *d'exécuter votre propre code SQL directement*.

L'exécution directe de code SQL vous permet d'affiner la requête qui sera lancée. Cependant, cela vous empêchera d'utiliser `contain` ou toute autre fonctionnalité de plus haut niveau de l'ORM.

Les Objets Table

class Cake\ORM**Table**

Les objets Table fournissent un accès à la collection des entities stockées dans une table spécifique. Chaque table dans votre application devra avoir une classe Table associée qui est utilisée pour interagir avec une table donnée. Si vous n'avez pas besoin de personnaliser le comportement d'une table donnée, CakePHP va générer une instance Table à utiliser pour vous.

Avant d'essayer d'utiliser les objets Table et l'ORM, vous devriez vous assurer que vous avez configuré votre *connexion à la base de données*.

Utilisation Basique

Pour commencer, créez une classe Table. Ces classes se trouvent dans **src/Model/Table**. Les Tables sont une collection de type model spécifique aux bases de données relationnelles, et sont l'interface principale pour votre base de données dans l'ORM de CakePHP. La classe table la plus basique devrait ressembler à ceci :

```
// src/Model/Table/ArticlesTable.php
namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{
}
```

Notez que nous ne disons pas à l'ORM quelle table utiliser pour notre classe. Par convention, les objets Table vont utiliser une table avec la notation en minuscule et avec des underscores pour le nom de la classe. Dans l'exemple du dessus, la table `articles` va être utilisée. Si notre classe table était nommée `BlogPosts`, votre table serait nommée `blog_posts`. Vous pouvez spécifier la table en utilisant la méthode `setTable()` :

```
namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{

    public function initialize(array $config): void
    {
        $this->setTable('my_table');
    }

}
```

Aucune convention d'inflexion ne sera appliquée quand on spécifie une table. Par convention, l'ORM s'attend aussi à ce que chaque table ait une clé primaire avec le nom de `id`. Si vous avez besoin de modifier ceci, vous pouvez utiliser la méthode `setPrimaryKey()` :

```
namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{
    public function initialize(array $config): void
    {
        $this->setPrimaryKey('my_id');
    }
}
```

Personnaliser la Classe Entity qu'une Table Utilise

Par défaut, les objets table utilisent une classe entity basée sur les conventions de nommage. Par exemple, si votre classe de table est appelée `ArticlesTable` l'entity sera `Article`. Si la classe table est `PurchaseOrdersTable` l'entity sera `PurchaseOrder`. Cependant si vous souhaitez utiliser une entity qui ne suit pas les conventions, vous pouvez utiliser la méthode `setEntityClass()` pour changer les choses :

```
class PurchaseOrdersTable extends Table
{
    public function initialize(array $config): void
    {
        $this->setEntityClass('App\Model\Entity\PO');
    }
}
```

Comme vu dans les exemples ci-dessus, les objets Table ont une méthode `initialize()` qui est appelée à la fin du constructeur. Il est recommandé d'utiliser cette méthode pour placer la logique d'initialisation au lieu de surcharger le constructeur.

Obtenir les Instances d'une Classe Table

Avant de pouvoir requêter sur une table, vous aurez besoin d'obtenir une instance de la table. Vous pouvez faire ceci en utilisant la classe `TableLocator` :

```
// Dans un controller
$articles = $this->getTableLocator()->get('Articles');
```

Le TableLocator fournit les diverses dépendances pour construire la table, et maintient un registre de toutes les instances de table construites, facilitant la construction de relations et la configuration l'ORM. Regardez *Utiliser le TableLocator* pour plus d'informations.

Si votre classe table est dans un plugin, assurez-vous d'utiliser le bon nom pour votre classe table. Ne pas le faire peut entraîner des résultats non voulus dans les règles de validation, ou que les callbacks ne soient pas récupérés car une classe par défaut est utilisée à la place de votre classe souhaitée. Pour charger correctement les classes table de votre plugin, utilisez ce qui suit :

```
// Table de plugin
$articlesTable = $this->getTableLocator()->get('PluginName.Articles');
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Table de plugin préfixé par Vendor
$articlesTable = $this->getTableLocator()->get('VendorName/PluginName.Articles');
```

Callbacks du Cycle de Vie

Comme vous l'avez vu ci-dessus les objets table déclenchent un certain nombre d'events. Les events sont utiles si vous souhaitez ajouter de la logique dans l'ORM sans faire de sous-classe et sans réécrire les méthodes. Les écouteurs (*listeners*) d'events peuvent être définis dans les classes de table ou de behavior. Vous pouvez aussi utiliser le gestionnaire d'events d'une table pour y lier des écouteurs dedans.

Lors de l'utilisation des méthodes de callback, les behaviors attachés dans la méthode `initialize()` déclencheront leurs écouteurs **avant** que les méthodes de callback de la table ne soient déclenchées. Ceci suit la même séquence que les controllers et les components.

Pour ajouter un écouteur d'events à une classe Table ou à un Behavior, implémentez simplement les signatures de méthodes comme décrit ci-dessus. Consultez les *Événements système* pour avoir plus de détails sur la façon d'utiliser le sous-système d'events :

```
// Dans un controller
$articles->save($article, ['variablePerso1' => 'votreValeur1']);

// Dans ArticlesTable.php
public function afterSave(Event $event, EntityInterface $entity, ArrayObject $options)
{
    $variablePerso = $options['variablePerso1']; // 'votreValeur1'
    $options['variablePerso2'] = 'votreValeur2';
}

public function afterSaveCommit(Event $event, EntityInterface $entity, ArrayObject
↪$options)
{
    $variablePerso = $options['variablePerso1']; // 'votreValeur1'
    $variablePerso = $options['variablePerso2']; // 'votreValeur2'
}
```

Liste des Events

- `Model.initialize`
- `Model.beforeMarshal`
- `Model.afterMarshal`
- `Model.beforeFind`
- `Model.buildValidator`
- `Model.buildRules`
- `Model.beforeRules`
- `Model.afterRules`
- `Model.beforeSave`
- `Model.afterSave`
- `Model.afterSaveCommit`
- `Model.beforeDelete`
- `Model.afterDelete`
- `Model.afterDeleteCommit`

initialize

`Cake\ORM\Table::initialize(EventInterface $event, ArrayObject $data, ArrayObject $options)`

L'event `Model.initialize` est déclenché après que les méthodes de constructeur et initialize ont été appelées. Les classes `Table` n'écoutent pas cet event par défaut, et utilisent plutôt la méthode hook `initialize`.

Pour répondre à l'event `Model.initialize`, vous pouvez créer une classe écouteur qui implémente `EventListenerInterface` :

```
use Cake\Event\EventListenerInterface;
class ModelInitializeListener implements EventListenerInterface
{
    public function implementedEvents()
    {
        return [
            'Model.initialize' => 'initializeEvent',
        ];
    }
    public function initializeEvent($event): void
    {
        $table = $event->getSubject();
        // faire quelque chose ici
    }
}
```

et attacher l'écouteur au `EventManager` ainsi :

```
use Cake\Event\EventManager;
$listener = new ModelInitializeListener();
EventManager::instance()->attach($listener);
```

Ceci va appeler `initializeEvent` quand une classe `Table` est construite.

beforeMarshal

`Cake\ORM\Table::beforeMarshal(EventInterface $event, ArrayObject $data, ArrayObject $options)`

L'event `Model.beforeMarshal` est déclenché avant que les données de requête ne soient converties en entities. Consultez la documentation *Modifier les Données de la Requête Avant de Construire les Entities* pour plus d'informations.

afterMarshal

`Cake\ORM\Table::afterMarshal(EventInterface $event, EntityInterface $entity, ArrayObject $data, ArrayObject $options)`

L'event `Model.afterMarshal` est déclenché après que les données de requête ont été converties en entities.Les gestionnaires d'events obtiendront les entities converties, les données originales de la requête et les options fournies à `patchEntity()` ou `newEntity()`.

beforeFind

`Cake\ORM\Table::beforeFind(EventInterface $event, Query $query, ArrayObject $options, $primary)`

L'event `Model.beforeFind` est lancé avant chaque opération find. En arrêtant l'événement et en alimentant la requête avec un jeu de résultats personnalisé, vous pouvez ignorer complètement l'opération de recherche :

```
public function beforeFind(EventInterface $event, Query $query, ArrayObject $options,
↪$primary)
{
    if (/* ... */) {
        $event->stopPropagation();
        $query->setResult(new \Cake\Datasource\ResultSetDecorator([]));

        return;
    }
    // ...
}
```

Dans cet exemple, aucun autre événement `beforeFind` ne sera déclenché sur la table associée ou ses comportements attachés (bien que les événements de comportement soient généralement appelés plus tôt compte tenu de leurs priorités par défaut), et la requête renverra le jeu de résultats vide qui a été transmis via `Query::setResult()`.

Tout changement fait à l'instance `$query` sera retenu pour le reste du find. Le paramètre `$primary` indique si oui ou non ceci est la requête racine ou une requête associée. Un event `Model.beforeFind` sera déclenché dans toutes les associations participant à la requête. Pour les associations qui utilisent des jointures, une requête factice sera fournie. Dans votre écouteur d'event, vous pouvez définir des champs supplémentaires, des conditions, des jointures ou des formateurs de résultat. Ces options/fonctionnalités seront copiées dans la requête racine.

Dans les versions précédentes de CakePHP, il y avait un callback `afterFind`, qui a été remplacé par les fonctionnalités de *Modifier les Résultats avec Map/Reduce* et les constructeurs d'entity.

buildValidator

`Cake\ORM\Table::buildValidator(EventInterface $event, Validator $validator, $name)`

L'event `Model.buildValidator` est déclenché lorsque le validator `$name` est créé. Les behaviors peuvent utiliser ce hook pour ajouter des méthodes de validation.

buildRules

`Cake\ORM\Table::buildRules(EventInterface $event, RulesChecker $rules)`

L'event `Model.buildRules` est déclenché après qu'une instance de règles a été créée et après que la méthode `buildRules()` de la table a été appelée.

beforeRules

`Cake\ORM\Table::beforeRules(EventInterface $event, EntityInterface $entity, ArrayObject $options, $operation)`

L'event `Model.beforeRules` est déclenché avant que les règles n'aient été appliquées à une entity. En stoppant cet event, vous pouvez retourner la valeur finale de l'opération de vérification des règles.

afterRules

`Cake\ORM\Table::afterRules(EventInterface $event, EntityInterface $entity, ArrayObject $options, $result, $operation)`

L'event `Model.afterRules` est déclenché après que les règles soient appliquées à une entity. En stoppant cet event, vous pouvez retourner la valeur finale de l'opération de vérification des règles.

beforeSave

`Cake\ORM\Table::beforeSave(EventInterface $event, EntityInterface $entity, ArrayObject $options)`

L'event `Model.beforeSave` est déclenché avant que chaque entity ne soit sauvegardée. Stopper cet event va annuler l'opération de sauvegarde. Quand l'event est stoppé, le résultat de l'event sera retourné. La manière de stopper un event est documentée *ici*.

afterSave

`Cake\ORM\Table::afterSave(EventInterface $event, EntityInterface $entity, ArrayObject $options)`

L'event `Model.afterSave` est déclenché après qu'une entity ne soit sauvegardée.

afterSaveCommit

`Cake\ORM\Table::afterSaveCommit(EventInterface $event, EntityInterface $entity, ArrayObject $options)`

L'event `Model.afterSaveCommit` est lancé après que la transaction, dans laquelle l'opération de sauvegarde est fournie, a été committée. Il est aussi déclenché pour des sauvegardes non atomic, quand les opérations sur la base de données sont implicitement committées. L'event est déclenché seulement pour la table primaire sur laquelle `save()` est directement appelée. L'event n'est pas déclenché si une transaction est démarrée avant l'appel de save.

beforeDelete

`Cake\ORM\Table::beforeDelete(EventInterface $event, EntityInterface $entity, ArrayObject $options)`

L'event `Model.beforeDelete` est déclenché avant qu'une entity ne soit supprimée. En stoppant cet event, vous allez annuler l'opération de suppression. Quand l'event est stoppé le résultat de l'event sera retourné.

afterDelete

`Cake\ORM\Table::afterDelete(EventInterface $event, EntityInterface $entity, ArrayObject $options)`

L'event `Model.afterDelete` est déclenché après qu'une entity a été supprimée.

afterDeleteCommit

`Cake\ORM\Table::afterDeleteCommit(EventInterface $event, EntityInterface $entity, ArrayObject $options)`

L'event `Model.afterDeleteCommit` est lancé après que la transaction, dans laquelle l'opération de sauvegarde est fournie, a été committée. Il est aussi déclenché pour des suppressions non atomic, quand les opérations sur la base de données sont implicitement committées. L'event est décenché seulement pour la table primaire sur laquelle `delete()` est directement appelée. L'event n'est pas déclenché si une transaction est démarrée avant l'appel de delete.

Stopper des Events de Table

Pour empêcher la sauvegarde de se poursuivre, arrêtez simplement la propagation de l'event dans votre callback :

```
public function beforeSave(EventInterface $event, EntityInterface $entity, ArrayObject
↪$options)
{
    if (...) {
        $event->stopPropagation();
        $event->setResult(false);

        return;
    }
    ...
}
```

Alternativement, vous pouvez aussi renvoyer false depuis votre callback. Cela a le même effet d'arrêt de la propagation.

Priorités de Callbacks

Quand vous utilisez des events sur vos tables et vos behaviors, ayez en tête la priorité et l'ordre dans lequel les écouteurs sont attachés. Les events des behaviors sont attachés avant ceux des tables. Avec les priorités par défaut, cela signifie que les callbacks de behaviors seront déclenchés **avant** l'event de la table ayant le même nom.

À titre d'exemple, si votre Table utilise `TreeBehavior`, la méthode `TreeBehavior::beforeDelete()` sera appelée avant la méthode `beforeDelete()` de votre table, et ne pourra pas travailler avec les nœuds enfantsde l'enregistrement qui est en train d'être supprimé dans la méthode de votre Table.

Vous avez plusieurs façons de gérer les priorités d'events :

1. Changez la priorité des écouteurs d'un Behavior en utilisant l'option `priority`. Cela modifiera la priorité de **toutes** les méthodes de callback dans le Behavior :

   ```
   // Dans la méthode initialize() d'une Table
   $this->addBehavior('Tree', [
       // La valeur par défaut est 10, et les écouteurs sont déclenchés de
       // la plus faible valeur de priorité à la plus haute.
       'priority' => 2,
   ]);
   ```

2. Modifiez la priorité dans votre classe `Table` en utilisant la méthode `Model.implementedEvents()`. Cela vous permet d'assigner une priorité différente pour chaque fonction de callback :

   ```
   // Dans une classe Table.
   public function implementedEvents()
   {
       $events = parent::implementedEvents();
       $events['Model.beforeDelete'] = [
           'callable' => 'beforeDelete',
           'priority' => 3
       ];
   ```

Behaviors

`Cake\ORM\Table::addBehavior($name, array $options = [])`

Les Behaviors fournissent un moyen de créer des parties de logique réutilisables horizontalement liées aux classes table. Vous vous demandez peut-être pourquoi les behaviors sont des classes classiques et non des traits. La raison principale tient aux écouteurs d'event. Alors que les traits permettent de réutiliser des parties de logique, ils compliqueraient la liaison des events.

Pour ajouter un behavior à votre table, vous pouvez appeler la méthode `addBehavior()`. Généralement, le meilleur endroit pour le faire est dans la méthode `initialize()` :

```
namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{
    public function initialize(array $config): void
    {
        $this->addBehavior('Timestamp');
    }
}
```

Comme pour les associations, vous pouvez utiliser la *syntaxe de plugin* et fournir des options de configuration supplémentaires :

```
namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{
    public function initialize(array $config): void
    {
        $this->addBehavior('Timestamp', [
            'events' => [
                'Model.beforeSave' => [
                    'created_at' => 'new',
                    'modified_at' => 'always'
                ]
            ]
        ]);
    }
}
```

Vous pouvez en savoir plus sur les behaviors, y compris sur les behaviors fournis par CakePHP dans le chapitre sur les *Behaviors (Comportements)*.

Configurer les Connexions

Par défaut, toutes les instances de table utilisent la connexion à la base de données `default`. Si votre application utilise plusieurs connexions à la base de données, vous voudrez peut-être configurer quelle table utilise quelle connexion. C'est le rôle de la méthode `defaultConnectionName()` :

```
namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{
    public static function defaultConnectionName(): string {
        return 'replica_db';
    }
}
```

> **Note**
>
> La méthode `defaultConnectionName()` **doit** être statique.

Utiliser le TableLocator

class `Cake\ORM\`**`TableLocator`**

Comme nous l'avons vu précédemment, la classe TableLocator fournit un registre/fabrique facile d'utilisation pour accéder aux instances des tables de vos applications. Elle fournit aussi quelques autres fonctionnalités utiles.

Configurer les Objets Table

`Cake\ORM\TableLocator::`**`get`**(*$alias*, *$config*)

Lors du chargement des tables à partir du registry, vous pouvez personnaliser leurs dépendances, ou utiliser les objets factices en fournissant un tableau `$options` :

```
$articles = FactoryLocator::get('Table')->get('Articles', [
    'className' => 'App\Custom\ArticlesTable',
    'table' => 'my_articles',
    'connection' => $connectionObject,
    'schema' => $schemaObject,
    'entityClass' => 'Custom\EntityClass',
    'eventManager' => $eventManager,
    'behaviors' => $behaviorRegistry
]);
```

Remarquez les paramètres de configurations de la connexion et du schéma, ils ne sont pas des valeurs de type string mais des objets. La connexion va prendre un objet `Cake\Database\Connection` et un schéma `Cake\Database\Schema\Collection`.

> **Note**
>
> Si votre table fait aussi une configuration supplémentaire dans sa méthode `initialize()`, ces valeurs vont écraser celles fournies au registre.

Vous pouvez aussi pré-configurer le registre en utilisant la méthode `setConfig()`. Les données de configuration sont stockées *par alias*, et peuvent être surchargées par une méthode `initialize()` de l'objet :

```
FactoryLocator::get('Table')->setConfig('Users', ['table' => 'my_users']);
```

> **Note**
>
> Vous pouvez configurer une table avant ou pendant la **première** fois où vous accédez à l'alias. Faire ceci après que le registre est rempli n'aura aucun effet.

Vider le Registre

`Cake\ORM\TableLocator::clear()`

Pendant les cas de test, vous voudrez vider le registre. Faire ceci est souvent utile quand vous utilisez les objets factices, ou modifiez les dépendances d'une table :

```
FactoryLocator::get('Table')->clear();
```

Configurer le Namespace pour Localiser les Classes de l'ORM

Si vous n'avez pas suivi les conventions, il est probable que vos classes Table ou Entity ne soient pas detectées par CakePHP. Pour régler cela, vous pouvez définir un namespace avec la méthode `Cake\Core\Configure::write`. Par exemple :

```
/src
    /App
        /My
            /Namespace
                /Model
                    /Entity
                    /Table
```

Serait configuré avec :

```
Cake\Core\Configure::write('App.namespace', 'App\My\Namespace');
```

Entities

class `Cake\ORM\`**`Entity`**

Tandis que les *objets Table* représentent et fournissent un accès à une collection d'objets, les entities représentent des lignes individuelles ou des objets de domaine dans votre application. Les entities contiennent des méthodes pour manipuler et accéder aux données qu'elles contiennent. Les champs sont aussi accessibles en tant que propriétés de l'objet.

Les entities sont créées pour vous par CakePHP à chaque fois que vous faites une itération sur l'objet query renvoyé par `find()` sur un objet table, ou quand vous appelez les méthodes `all()` ou `first()` sur l'objet query.

Créer des Classes Entity

Vous n'avez pas besoin de créer des classes entity pour utiliser l'ORM dans CakePHP. Cependant si vous souhaitez avoir de la logique personnalisée dans vos entities, vous devrez créer des classes. Par convention, les classes entity se trouvent dans **src/Model/Entity/**. Si notre application a une table `articles`, nous pouvons créer l'entity suivante :

```
// src/Model/Entity/Article.php
namespace App\Model\Entity;

use Cake\ORM\Entity;

class Article extends Entity
{
}
```

Pour l'instant cette entity ne fait pas grand chose. Cependant quand nous chargeons les données de notre table articles, nous obtenons des instances de cette classe.

> **Note**
>
> Si vous ne définissez pas de classe entity, CakePHP va utiliser la classe de base Entity.

Créer des Entities

Les Entities peuvent être instanciées directement :

```
use App\Model\Entity\Article;

$article = new Article();
```

Lorsque vous instanciez une entity, vous pouvez lui passer des champs avec les données que vous voulez y stocker :

```
use App\Model\Entity\Article;

$article = new Article([
    'id' => 1,
    'title' => 'Nouvel Article',
    'created' => new DateTime('now')
]);
```

Pour obtenir une entity vierge, le meilleur moyen est d'appeler `newEmptyEntity()` sur l'objet `Table` :

```
use Cake\ORM\Locator\LocatorAwareTrait;

$article = $this->fetchTable('Articles')->newEmptyEntity();

$article = $this->fetchTable('Articles')->newEntity([
    'id' => 1,
    'title' => 'New Article',
    'created' => new DateTime('now')
]);
```

`$article` sera une instance de `App\Model\Entity\Article`, ou une instance de la classe par défaut `Cake\ORM\Entity` si vous n'avez pas créé la classe `Article`.

> **Note**

Avant CakePHP 4.3, il fallait utiliser `$this->getTableLocator->get('Articles')` pour obtenir une instance de la table.

Accéder aux Données de l'Entity

Les entities fournissent plusieurs façons d'accéder aux données qu'elles contiennent. La plupart du temps, vous accéderez aux données dans une entity en utilisant la notation objet :

```
use App\Model\Entity\Article;

$article = new Article;
$article->title = 'Ceci est mon premier post';
echo $article->title;
```

Vous pouvez aussi utiliser les méthodes `get()` et `set()`.

`Cake\ORM\Entity::set`(*$field*, *$value = null*, *array $options = []*)

`Cake\ORM\Entity::get`(*$field*)

Par exemple :

```
$article->set('title', 'Ceci est mon premier post');
echo $article->get('title');
```

Quand vous utilisez `set()`, vous pouvez mettre à jour plusieurs champs en une seule fois en utilisant un tableau :

```
$article->set([
    'title' => 'Mon premier post',
    'body' => "C'est le meilleur de tous!"
]);
```

Avertissement

Lors de la mise à jour des entities avec des données requêtées, vous devriez faire une liste des champs qui peuvent être définis par assignement de masse.

Vous pouvez vérifier si des champs sont définis dans vos entities avec `has()` :

```
$article = new Article([
    'title' => 'Premier post',
    'user_id' => null
]);
$article->has('title'); // true
$article->has('user_id'); // false
$article->has('undefined'); // false.
```

La méthode `has()` va renvoyer `true` si un champ est défini est a une valeur non null. Vous pouvez utiliser `isEmpty()` et `hasValue()` pour vérifier si un champ contient une valeur "non-empty" :

```
$article = new Article([
    'title' => 'Premier post',
    'user_id' => null
```

(suite sur la page suivante)

(suite de la page précédente)

```
    'text' => '',
    'links' => []
]);
]);
$article->has('title'); // true
$article->isEmpty('title');  // false
$article->hasValue('title'); // true

$article->has('user_id'); // true
$article->isEmpty('user_id');  // true
$article->hasValue('user_id'); // false

$article->has('text'); // true
$article->isEmpty('text');  // true
$article->hasValue('text'); // false

$article->has('links'); // true
$article->isEmpty('links');  // true
$article->hasValue('links'); // false

$article->has('text'); // true
$article->isEmpty('text');  // true
$article->hasValue('text'); // false

$article->has('links'); // true
$article->isEmpty('links');  // true
$article->hasValue('links'); // false
```

Accesseurs & Mutateurs

En plus de l'interface simple get/set, les entities vous permettent de fournir des méthodes accesseurs et mutateurs. Avec ces méthodes, vous pouvez personnaliser la façon dont les champs sont lus ou définis.

Accesseurs

Les accesseurs personnalisent la façon dont les champs seront lus. Ils suivent la convention `_get(NomDuChamp)` où `(NomDuChamp)` est la version CamelCase du nom du champ (les mots sont accollés avec une majuscule pour la première lettre de chacun).

Ils reçoivent la valeur basique stockée dans le tableau `_fields` pour seul argument. Par exemple :

```
namespace App\Model\Entity;

use Cake\ORM\Entity;

class Article extends Entity
{
    protected function _getTitle($title)
    {
        return strtoupper($title);
    }
}
```

Cet exemple convertit en majuscules la valeur du champ `title` à chaque fois qu'il est lu. Il sera exécuté quand vous récupérerez le champ *via* une de ces deux manières :

```
echo $article->title; // renvoie FOO au lieu de foo
echo $article->get('title'); // renvoie FOO au lieu de foo
```

Note

Le code dans vos accesseurs est exécuté à chaque fois que vous faites référence au champ. Vous pouvez utiliser une variable locale de la façon suivante pour le mettre en cache si vous réalisez une opération gourmande en ressources : *$maPropriete = $entity->ma_propriete*.

Avertissement

Les accesseurs seront utilisés lors de la sauvegarde des entities. Faites donc attention lorsque vous définissez des méthodes qui formatent les données car ce sont ces données formatées qui seront sauvegardées.

Mutateurs

Avec les mutateurs, vous pouvez personnaliser la façon dont les champs seront écrits dans l'entity. Ils suivent la convention `_set(NomDuChamp)` où `(NomDuChamp)` est la version CamelCase du nom du champ.

Les méthodes mutateurs doivent toujours retourner la valeur qui doit être stockée dans le champ. Vous pouvez aussi utiliser les mutateurs pour définir simultanément d'autres champs. Quand vous faites cela, soyez vigilant à ne pas introduire de boucles, car CakePHP n'empêchera pas les méthodes mutateurs de faire des boucles infinies. Par exemple :

```
namespace App\Model\Entity;

use Cake\ORM\Entity;
use Cake\Utility\Text;

class Article extends Entity
{
    protected function _setTitle($title)
    {
        $this->slug = Text::slug($title);

        return strtouppercase($title);
    }

}
```

Cet exemple fait deux choses : il stocke une version modifiée de la valeur spécifiée dans le champ `slug` et stocke une version en majuscules dans le champ `titre`. Il sera executé lorsque vous définirez le champ *via* une de ces deux manières :

```
$user->title = 'foo' // définit le champ slug et stocke FOO au lieu de foo
$user->set('title', 'foo'); // définit le champ slug et stocke FOO au lieu de foo
```

Avertissement

Les accesseurs sont également appelés avant que l'entity ne soit persistée dans la base. Si vous souhaitez transformer un champ mais ne pas persister la transformation, il est recommandé d'utiliser les propriétés virtuelles car ces dernières ne seront pas persistées.

Créer des Champs Virtuels

En définissant des accesseurs, vous pouvez fournir un accès à des champs qui n'existent pas réellement. Par exemple si votre table users a des champs `first_name` et `last_name`, vous pouvez créer une méthode pour le nom complet :

```
namespace App\Model\Entity;

use Cake\ORM\Entity;

class User extends Entity
{
    protected function _getFullName()
    {
        return $this->first_name . '  ' . $this->last_name;
    }
}
```

Vous pouvez accéder aux champs virtuels comme s'ils existaient sur l'entity. Le nom du champ sera le nom de la méthode en minuscules, avec des underscores pour séparer les mots (`full_name`) :

```
echo $user->full_name;
echo $user->get('full_name');
```

Souvenez-vous que les champs virtuels ne peuvent pas être utilisés dans les finds. Si vous voulez qu'ils fassent partie des données JSON ou dans des représentations en tableau de vos entités, reportez-vous à la section *Montrer les Champs Virtuels*.

Vérifier si une Entity a été Modifiée

`Cake\ORM\Entity::dirty`(*$field = null*, *$dirty = null*)

Vous pourriez vouloir écrire du code conditionnel basé sur l'existence ou non de modifications dans l'entity. Par exemple, vous pourriez vouloir valider uniquement les champs lorsqu'ils ont été modifiés :

```
// Vérifie si le champ title n'a pas été modifié.
$article->isDirty('title');
```

Vous pouvez également marquer un champ comme ayant été modifié. C'est pratique lorsque vous ajoutez des données dans des champs contenant un tableau car sinon cela ne marque pas automatiquement le champ comme ayant été modifié, seule la redéfinition du tableau complet aurait cet effet :

```
// Ajoute un commentaire et marque le champ comme modifié.
$article->comments[] = $nouveauCommentaire;
$article->setDirty('comments', true);
```

De plus, vous pouvez également baser votre code conditionnel sur les valeurs initiales des champs en utilisant la méthode `getOriginal()`. Cette méthode retournera soit la valeur initiale de la propriété si elle a été modifiée soit la valeur actuelle.

Vous pouvez également vérifier si l'un quelconque des champs de l'entity a été modifié :

```
// Vérifier si l'entity a changé
$article->isDirty();
```

Pour retirer le marquage *dirty* (modifié) des champs d'une entity, vous pouvez utiliser la méthode `clean()` :

```
$article->clean();
```

Lors de la création d'un nouvelle entity, vous pouvez empêcher les champs d'être marqués *dirty* en passant une option supplémentaire :

```
$article = new Article(['title' => 'Nouvel Article'], ['markClean' => true]);
```

Pour récupérer la liste des propriétés *dirty* d'une `Entity`, vous pouvez appeler :

```
$dirtyFields = $entity->getDirty();
```

Erreurs de Validation

Après avoir *sauvegardé une entity*, toute erreur de validation sera stockée sur l'entity elle-même. Vous pouvez accéder à toutes les erreurs de validation en utilisant les méthodes `getErrors()`, `getError()` ou `hasErrors()` :

```
// Récupère toutes les erreurs
$errors = $user->getErrors();

// Récupère les erreurs pour un seul champ.
$errors = $user->getError('password');

// L'entity (ou une entity imbriquée) a-t-elle une erreur ?
$user->hasErrors();

// L'entity racine (uniquement) a-t-elle une erreur ?
$user->hasErrors(false);
```

Les méthodes `setErrors()` et `setError()` peuvent aussi être utilisées pour définir les erreurs sur une entity, ce qui facilite les tests du code qui fonctionne avec des messages d'erreur :

```
$user->setError('password', ['Le mot de passe est obligatoire.']);
$user->setErrors([
    'password' => ['Le mot de passe est obligatoire'],
    'username' => ['Le nom d\'utilisateur est obligatoire']
]);
```

Assignement de Masse

Bien que la définition en masse de champs des entities soit simple et pratique, elle peut créer d'importants problèmes de sécurité. Assigner en masse les données d'utilisateur à partir de la requête dans une entity permet à l'utilisateur de modifier n'importe quelles colonnes (voire toutes). Utiliser des classes entity anonymes ou créer des classes entity avec la commande *Console Bake* de CakePHP ne protège pas contre l'assignement en masse.

La propriété `_accessible` vous permet de fournir une liste des champs et d'indiquer s'ils peuvent être assignés en masse ou non. Les valeurs `true` et `false` indiquent si un champ peut ou ne peut pas être assigné massivement :

```
namespace App\Model\Entity;

use Cake\ORM\Entity;

class Article extends Entity
{
    protected array $_accessible = [
        'title' => true,
        'body' => true
    ];
}
```

En plus des champs réels, il existe un champ spécial * qui définit le comportement par défaut si un champ n'est pas nommé spécifiquement :

```
namespace App\Model\Entity;

use Cake\ORM\Entity;

class Article extends Entity
{
    protected array $_accessible = [
        'title' => true,
        'body' => true,
        '*' => false,
    ];
}
```

Note

Si la propriété * n'est pas définie, elle sera par défaut à `false`.

Éviter la Protection Contre l'Assignement de Masse

Lors de la création d'un nouvelle entity en utilisant le mot clé `new`, vous pouvez lui spécifier de ne pas se protéger contre l'assignement de masse :

```
use App\Model\Entity\Article;

$article = new Article(['id' => 1, 'title' => 'Foo'], ['guard' => false]);
```

Modifier les Champs Protégés à la Volée

Vous pouvez modifier à la volée la liste des champs protégés en utilisant la méthode `setAccess()` :

```
// Rendre user_id accessible.
$article->setAccess('user_id', true);

// Rendre title protégé.
$article->setAccess('title', false);
```

> **Note**
>
> Modifier des champs accessibles agit seulement sur l'instance sur laquelle la méthode est appelée.

Lorsque vous utilisez les méthodes `newEntity()` et `patchEntity()` dans les objets `Table` vous pouvez également utiliser des options pour personnaliser la protection de masse. Référez-vous à la section *Changer les Champs Accessibles* pour plus d'information.

Outrepasser la Protection de Champ

Il arrive parfois que vous souhaitiez permettre un assignement en masse aux champs protégés :

```
$article->set($fields, ['guard' => false]);
```

En définissant l'option `guard` à `false`. vous pouvez ignorer la liste des champs accessibles pour un appel unique de `set()`.

Vérifier si une Entity a été Sauvegardée

Il est souvent nécessaire de savoir si une entity représente une ligne qui est déjà présente en base de données. Pour cela, utilisez la méthode `isNew()` :

```
if (!$article->isNew()) {
    echo 'Cette article a déjà été sauvegardé!';
}
```

Si vous êtes certains qu'une entity a déjà été sauvegardée, vous pouvez utiliser `setNew()` :

```
$article->setNew(false);

$article->setNew(true);
```

Lazy Loading des Associations

Bien que la façon la plus efficace pour accéder à vos associations soit généralement de les charger en eager loading, il peut arriver que vous ayez besoin d'utiliser le lazy loading des données associées. Avant de voir comment utiliser le Lazy loading des associations, nous allons devoir parler des différences entre le chargement eager et lazy :

Eager loading
: Le Eager loading utilise les joins (quand c'est possible) pour récupérer les données de la base de données avec aussi *peu* de requêtes que possible. Quand une requête séparée est nécessaire, comme dans le cas d'une association HasMany, une requête unique est émise pour récupérer *toutes* les données associées pour l'ensemble des objets à récupérer.

Lazy loading
: Le Lazy loading retarde le chargement des données de l'association jusqu'à ce que ce soit absolument nécessaire. Si cela peut certes économiser du temps CPU car des données possiblement non utilisées ne sont pas hydratées dans les objets, cela peut aussi résulter en beaucoup plus de requêtes émises vers la base de données. Par exemple faire des boucles sur un ensemble d'articles et leurs commentaires va fréquemment émettre N requêtes, où N est le nombre d'articles itérés.

Bien que le lazy loading ne soit pas inclus dans l'ORM de CakePHP, vous pouvez tout simplement utiliser un des plugins de la communauté pour le faire. Nous recommandons le plugin LazyLoad [134]

Après avoir ajouté le plugin à votre entity, vous pourrez faire ceci :

134. https://github.com/jeremyharris/cakephp-lazyload

```
$article = $this->Articles->findById($id);

// La propriété commentaires a été chargée en lazy
foreach ($article->comments as $comment) {
    echo $comment->body;
}
```

Créer du Code Réutilisable avec les Traits

Vous pouvez vous retrouver dans un cas où vous avez besoin de la même logique dans plusieurs classes d'entity. Les traits de PHP sont parfaits pour cela. Vous pouvez mettre les traits de votre application dans **src/Model/Entity**. Par convention, les traits dans CakePHP sont suffixés avec `Trait` pour qu'ils soient discernables des classes ou des interfaces. Les traits sont souvent un bon allié des behaviors, vous permettant de fournir des fonctionnalités pour les objets table et entity.

Par exemple si nous avions un plugin SoftDeletable, il pourrait fournir un trait. Ce trait pourrait donner des méthodes pour marquer les entities comme "supprimées", la méthode `softDelete` pouvant être fournie par un trait :

```
// SoftDelete/Model/Entity/SoftDeleteTrait.php

namespace SoftDelete\Model\Entity;

trait SoftDeleteTrait
{
    public function softDelete()
    {
        $this->set('deleted', true);
    }
}
```

Vous pourriez ensuite utiliser ce trait dans votre classe d'entity par une importation et une inclusion :

```
namespace App\Model\Entity;

use Cake\ORM\Entity;
use SoftDelete\Model\Entity\SoftDeleteTrait;

class Article extends Entity
{
    use SoftDeleteTrait;
}
```

Convertir en Tableaux/JSON

Lors de la construction d'APIs, il est probable que vous aurez fréquemment besoin de convertir des entities en tableaux ou en données JSON. CakePHP rend cela très simple :

```
// Obtenir un tableau.
// Les associations seront aussi converties par toArray().
$array = $user->toArray();

// Convertir en JSON
// Les associations seront aussi converties avec le hook jsonSerialize.
$json = json_encode($user);
```

Lors de la conversion d'une entity en JSON, les listes de champs virtuels & cachés sont utilisées. Les entities sont aussi converties récursivement en JSON. Cela signifie que si les entities et leurs associations sont chargées en eager loading, CakePHP va gérer correctement la conversion des données associées dans le bon format.

Montrer les Champs Virtuels

Par défaut, les champs virtuels ne sont pas exportés lors de la conversion des entities en tableaux ou en JSON. Pour exposer les champs virtuels, vous devez les rendre visibles. Lors de la définition de votre classe entity, vous pouvez fournir une liste de champs virtuels qui doivent être exposés :

```
namespace App\Model\Entity;

use Cake\ORM\Entity;

class User extends Entity
{
    protected array $_virtual = ['full_name'];
}
```

Cette liste peut être modifiée à la volée en utilisant la méthode `setVirtual` :

```
$user->setVirtual(['full_name', 'is_admin']);
```

Cacher les Champs

Il arrive souvent que vous ne souhaitiez pas exporter certains champs dans des formats JSON ou en tableau. Par exemple il est souvent mal avisé de montrer les hashs de mot de passe ou les questions de récupération du compte. Lors de la définition d'une classe entity, définissez quels champs doivent être cachés :

```
namespace App\Model\Entity;

use Cake\ORM\Entity;

class User extends Entity
{
    protected $_hidden = ['password'];
}
```

Cette liste peut être modifiée à la volée en utilisant la méthode `setHidden` :

```
$user->setHidden(['password', 'recovery_question']);
```

Stocker des Types Complexes

Les accesseurs et mutateurs n'ont pas pour objectif de contenir de la logique pour sérialiser et desérialiser les données complexes venant de la base de données. Consultez la section *Sauvegarder les Types Complexes* pour comprendre la façon dont votre application peut stocker des types de données complexes comme les tableaux et les objets.

Récupérer les Données et les Ensembles de Résultats

class `Cake\ORM\Table`

Tandis que les objets table fournissent une abstraction autour d'un "dépôt" ou d'une collection d'objets, Les objets "entity" correspondent à ce que vous obtenez quand vous faites une requête sur des enregistrements individuels. Dans

la mesure où cette section parle des différentes façons de trouver et charger les entities, vous pouvez lire préalablement la section *Entities* pour en savoir plus sur les entities.

Déboguer les Queries et les ResultSets

Étant donné que l'ORM retourne maintenant des Collections et des Entities, déboguer ces objets peut être plus compliqué qu'avec les versions précédentes de CakePHP. Il y a désormais plusieurs façons d'inspecter les données retournées par l'ORM.

- `debug($query)` Montre le SQL et les paramètres liés, ne montre pas les résultats.
- `debug($query->all())` Montre les propriétés de ResultSet (pas les résultats).
- `debug($query->toList())` Montre les résultats dans un tableau.
- `debug(iterator_to_array($query))` Montre les résultats de la requête formatés en tableau.
- `debug($query->toArray())` Une façon facile de montrer chacun des résultats.
- `debug(json_encode($query, JSON_PRETTY_PRINT))` Résultats plus lisibles.
- `debug($query->first())` Montre les propriétés d'une seule entity.
- `debug((string)$query->first())` Montre les propriétés d'une seule entity en JSON.

Récupérer une Entity Unique avec une Clé Primaire

`Cake\ORM\Table::get`(*$id*, *$options = []*)

Quand vous modifiez ou visualisez des entities et leurs données liées, il est souvent pratique de charger une entity unique à partir de la base de données. Vous pouvez faire ceci en utilisant `get()` :

```
// Dans un controller ou dans une méthode de table.

// Récupère un article unique
$article = $articles->get($id);

// Récupère un article unique, et les commentaires liés
$article = $articles->get($id, [
    'contain' => ['Comments']
]);
```

Si l'opération get ne trouve aucun résultat, une `Cake\Datasource\Exception\RecordNotFoundException` sera levée. Vous pouvez soit attraper cette exception vous-même, soit permettre à CakePHP de la convertir en une erreur 404.

Comme `find()`, `get()` a un cache intégré. Vous pouvez utiliser l'option `cache` quand vous appelez `get()` pour appliquer la mise en cache :

```
// Dans un controller ou dans une méthode de table.

// Utiliser une configuration de cache quelconque,
// ou une instance de CacheEngine & une clé générée
$article = $articles->get($id, [
    'cache' => 'cache_personnalise',
]);

// Utilise une configuration de cache quelconque,
// ou une instance de CacheEngine & une clé spécifique
$article = $articles->get($id, [
    'cache' => 'cache_personnalise', 'key' => 'ma_cle'
]);
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Désactive explicitement la mise en cache
$article = $articles->get($id, [
    'cache' => false
]);
```

Une autre possibilité est de faire un `get()` d'une entity en utilisant *Méthodes Finder Personnalisées*. Par exemple si vous souhaitez récupérer toutes les traductions pour une entity, vous pouvez le faire en utilisant l'option `finder` :

```
$article = $articles->get($id, [
    'finder' => 'traductions',
]);
```

Utiliser les Finders pour Charger les Données

`Cake\ORM\Table::find`(*$type*, *$options = []*)

Avant de travailler avec les entities, vous devrez les charger. La façon la plus facile de le faire est d'utiliser la méthode `find`. La méthode find est un moyen simple et extensible pour trouver les données qui vous intéressent :

```
// Dans un controller ou dans une méthode de table.

// Trouver tous les articles
$query = $articles->find('all');
```

La valeur retournée par une méthode `find` est toujours un objet `Cake\ORM\Query`. La classe Query vous permet de réaffiner une requête après l'avoir créée. Les objets Query sont évalués *lazily*, et ne s'exécutent qu'à partir du moment où vous commencez à récupérer des lignes, les convertissez en tableau, ou quand la méthode `all()` est appelée :

```
// Dans un controller ou dans une méthode de table.

// Trouver tous les articles.
// À ce niveau, la requête n'est pas lancée.
$query = $articles->find('all');

// L'itération va exécuter la requête.
foreach ($query->all() as $row) {
}

// Appeler all() va exécuter la requête
// et retourner l'ensemble des résultats.
$results = $query->all();

// Une fois le résultat obtenu, nous pouvons en récupérer toutes les lignes
$data = $results->toList();

// Convertir la requête en tableau associatif va l'exécuter.
$data = $query->toArray();
```

Note

Une fois que vous avez commencé une requête, vous pouvez utiliser l'interface *Query Builder* pour construire des requêtes plus complexes, ajouter des conditions supplémentaires, des limites, ou inclure des associations en utilisant

l'interface fluide.

```
// Dans un controller ou dans une méthode de table.
$query = $articles->find('all')
    ->where(['Articles.created >' => new DateTime('-10 days')])
    ->contain(['Comments', 'Authors'])
    ->limit(10);
```

Vous pouvez aussi fournir à `find()` plusieurs options couramment utilisées. Cela peut être utile pour les tests puisqu'il y a peu de méthodes à mocker :

```
// Dans un controller ou dans une méthode de table
$query = $articles->find('all', [
    'conditions' => ['Articles.created >' => new DateTime('-10 days')],
    'contain' => ['Authors', 'Comments'],
    'limit' => 10
]);
```

La liste des options supportées par find() est :

- `conditions` fournit des conditions pour la clause WHERE de la requête.
- `limit` définit le nombre de lignes que vous voulez.
- `offset` définit l'offset de la page que vous souhaitez. Vous pouvez aussi utiliser `page` pour faciliter le calcul.
- `contain` définit les associations à charger en eager.
- `fields` limite les champs chargés dans l'entity. Le fait de ne pas charger tous les champs peut cependant faire que les entities se comportent de manière inappropriée.
- `group` ajoute une clause GROUP BY à votre requête. C'est utile quand vous utilisez des fonctions d'agrégation.
- `having` ajoute une clause HAVING à votre requête.
- `join` définit les jointures personnalisées supplémentaires.
- `order` trie l'ensemble des résultats.

Les options qui ne sont pas dans cette liste seront passées aux écouteurs de beforeFind, où ils pourront être utilisés pour modifier l'objet requête. Vous pouvez utiliser la méthode `getOptions` sur un objet query pour récupérer les options utilisées. Bien que vous puissiez passer des objets requête à vos controllers, nous vous recommandons plutôt de les rassembler dans des *Méthodes Finder Personnalisées*. En utilisant des méthodes finder personnalisées, vous pourrez réutiliser vos requêtes et cela facilitera les tests.

Par défaut, les requêtes et les result sets renverront des objets *Entities*. Vous pouvez récupérer des tableaux basiques en désactivant l'hydratation :

```
$query->disableHydration();

// $data est le ResultSet qui contient le tableau de données.
$data = $query->all();
```

Récupérer les Premiers Résultats

La méthode `first()` vous permet de récupérer seulement la première ligne d'une requête. Si la requête n'a pas été exécutée, une clause `LIMIT 1` sera appliquée :

```
// Dans un controller ou dans une méthode de table.
$query = $articles->find('all', [
    'order' => ['Articles.created' => 'DESC']
]);
$row = $query->first();
```

Cette approche remplace le `find('first')` des versions précédentes de CakePHP. Vous pouvez aussi utiliser la méthode `get()` si vous recherchez les entities par leur clé primaire.

> **Note**
>
> La méthode `first()` renverra `null` si aucun résultat n'est trouvé.

Récupérer un Nombre de Résultats

Une fois que vous avez créé un objet query, vous pouvez utiliser la méthode `count()` pour récupérer un décompte des résultats de cette query :

```
// Dans un controller ou une méthode de table.
$query = $articles->find('all', [
    'conditions' => ['Articles.title LIKE' => '%Ovens%']
]);
$number = $query->count();
```

Consultez *Retourner le Nombre Total d'Enregistrements* pour d'autres utilisations de la méthode `count()`.

Trouver les Paires de Clé/Valeur

Cette fonctionnalité est pratique pour générer un tableau associatif de données à partir des données de votre application. C'est notamment très utile pour la création des elements *<select>*. CakePHP fournit une méthode simple à utiliser pour générer des listes de données :

```
// Dans un controller ou dans une méthode de table.
$query = $articles->find('list');
$data = $query->toArray();

// Les données ressemblent maintenant à ceci
$data = [
    1 => 'Premier post',
    2 => 'Mon deuxième article',
];
```

Sans autre option, les clés de `$data` correspondront à la clé primaire de votre table, tandis que les valeurs seront celles du champ désigné dans le paramètre "displayField" de la table. Le "displayField" par défaut est `title` ou `name`. Vous pouvez utiliser la méthode `setDisplayField()` de la table pour configurer le champ à afficher :

```
class ArticlesTable extends Table
{

    public function initialize(array $config): void
    {
        $this->setDisplayField('label');
    }
}
```

Au moment où vous appelez `list`, vous pouvez configurer les champs utilisés comme clé et comme valeur respectivement avec les options `keyField` et `valueField` :

```
// Dans un controller ou dans une méthode de table.
$query = $articles->find('list', [
```

(suite sur la page suivante)

(suite de la page précédente)

```
    'keyField' => 'slug',
    'valueField' => 'label'
]);
$data = $query->toArray();

// Les données ressemblent maintenant à
$data = [
    'premier-post' => 'Premier post',
    'mon-deuxieme-article' => 'Mon deuxième article',
];
```

Les résultats peuvent être regroupés. C'est utile quand vous voulez organiser valeurs en sous-groupes, ou que vous voulez construire des elements `<optgroup>` avec `FormHelper` :

```
// Dans un controller ou dans une méthode de table.
$query = $articles->find('list', [
    'keyField' => 'slug',
    'valueField' => 'label',
    'groupField' => 'author_id'
]);
$data = $query->toArray();

// Les données ressemblent maintenant à
$data = [
    1 => [
        'premier-post' => 'Premier post',
        'mon-deuxieme-article' => 'Mon deuxième article',
    ],
    2 => [
        // Les articles d'autres auteurs.
    ]
];
```

Vous pouvez aussi créer une liste de données à partir d'associations pouvant être réalisées avec une jointure :

```
$query = $articles->find('list', [
    'keyField' => 'id',
    'valueField' => 'author.name'
])->contain(['Authors']);
```

Les expressions `keyField`, `valueField`, et `groupField` font référence au chemin des attributs dans les entités, et non sur des colonnes de la base de données. Vous pouvez donc utiliser des champs virtuels dans les résultats de `find(list)`.

Personnaliser la Sortie Clé-Valeur

Pour finir, il est possible d'utiliser les closures pour accéder aux méthodes de mutation des entities dans vos finds list.

```
// Dans votre Entity Authors, créez un champ virtuel
// à utiliser en tant que champ à afficher:
protected function _getLibelle()
{
    return $this->_fields['first_name'] . ' ' . $this->_fields['last_name']
```

(suite sur la page suivante)

(suite de la page précédente)

```
        . ' / ' . __('User ID %s', $this->_fields['user_id']);
}
```

Cet exemple montre l'utilisation de la méthode accesseur `_getLabel()` dans l'entity Author.

```
// Dans vos finders/controller:
$query = $articles->find('list', [
        'keyField' => 'id',
        'valueField' => function ($article) {
            return $article->author->get('label');
        }
    ])
    ->contain('Authors');
```

Vous pouvez aussi récupérer le libellé directement dans la liste en utilisant.

```
// Dans AuthorsTable::initialize():
$this->setDisplayField('label'); // Va utiliser Author::_getLabel()
// Dans vos finders/controller:
$query = $authors->find('list'); // Va utiliser AuthorsTable::getDisplayField()
```

Trouver des Données Filées

Le finder `find('threaded')` retourne des entities imbriquées qui sont filées ensemble grâce à un champ servant de clé. Par défaut, ce champ est `parent_id`. Ce finder vous permet d'accéder aux données stockées dans une table de style "liste adjacente". Toutes les entities qui correspondent à un `parent_id` donné sont placées sous l'attribut `children` :

```
// Dans un controller ou dans une méthode table.
$query = $comments->find('threaded');

// Expanded les valeurs par défaut
$query = $comments->find('threaded', [
    'keyField' => $comments->primaryKey(),
    'parentField' => 'parent_id'
]);
$results = $query->toArray();

echo count($results[0]->children);
echo $results[0]->children[0]->comment;
```

Les clés `parentField` et `keyField` peuvent être utilisées pour définir les champs qui vont servir au filage.

> **Astuce**
>
> Si vous devez gérer des données en arbre plus compliquées, utiliser de préférence le *Tree*.

Méthodes Finder Personnalisées

Les exemples ci-dessus montrent comment utiliser les finders intégrés `all` et `list`. Cependant, il est possible et recommandé d'implémenter vos propres méthodes finder. Les méthodes finder sont idéales pour rassembler des requêtes utilisées couramment, ce qui vous permet de fournir une abstraction facile à utiliser pour de nombreux détails de la requête. Les méthodes finder sont définies en créant des méthodes nomméed par convention `findFoo`, où `Foo` est le

nom que vous souhaitez donner à votre finder. Par exemple, si nous voulons ajouter à notre table d'articles un finder servant à rechercher parmi les articles d'un certain auteur, nous ferions ceci :

```
use Cake\ORM\Query;
use Cake\ORM\Table;

class ArticlesTable extends Table
{

    public function findEcritPar(Query $query, array $options)
    {
        $user = $options['user'];

        return $query->where(['author_id' => $user->id]);
    }

}

$query = $articles->find('ecritPar', ['user' => $userEntity]);
```

Les méthodes finder peuvent modifier la requête comme il se doit, ou utiliser l'argument `$options` pour personnaliser l'opération finder selon la logique souhaitée. Vous pouvez aussi "empiler" les finders, ce qui permet d'exprimer sans effort des requêtes complexes. En supposant que vous ayez à la fois des finders "published" et "recent", vous pourriez très bien faire ceci :

```
$query = $articles->find('published')->find('recent');
```

Bien que tous les exemples que nous avons cités jusqu'ici montrent des finders qui s'appliquent sur des tables, il est aussi possible de définir des méthodes finder sur des *Behaviors (Comportements)*.

Si vous devez modifier les résultats après qu'ils ont été récupérés, vous pouvez utiliser une fonction *Modifier les Résultats avec Map/Reduce*. Les fonctionnalités de map reduce remplacent le callback "afterFind" présent dans les précédentes versions de CakePHP.

Finders Dynamiques

L'ORM de CakePHP fournit des finders construits dynamiquement, qui vous permettent d'exprimer des requêtes simples sans écrire de code particulier. Par exemple, admettons que vous recherchiez un utilisateur à partir de son *username*. Vous pourriez procéder ainsi :

```
// Dans un controller
// Les deux appels suivants sont équivalents.
$query = $this->Users->findByUsername('joebob');
$query = $this->Users->findAllByUsername('joebob');
```

Les finders dynamiques permettent même de filtrer sur plusieurs champs à la fois :

```
$query = $users->findAllByUsernameAndApproved('joebob', 1);
```

Vous pouvez aussi créer des conditions `OR` :

```
$query = $users->findAllByUsernameOrEmail('joebob', 'joe@example.com');
```

Vous pouvez utiliser des conditions OR ou AND, mais vous ne pouvez pas combiner les deux dans un même finder dynamique. Les autres options de requête comme `contain` ne sont pas non plus supportées par les finders dynamiques.

Vous devrez utiliser des *Méthodes Finder Personnalisées* pour encapsuler des requêtes plus complexes. Pour finir, vous pouvez aussi combiner les finders dynamiques avec des finders personnalisés :

```
$query = $users->findTrollsByUsername('bro');
```

Ce qui se traduirait ainsi :

```
$users->find('trolls', [
    'conditions' => ['username' => 'bro']
]);
```

Une fois que vous avez un objet query créé à partir d'un finder dynamique, vous devrez appeler `first()` si vous souhaitez récupérer le premier résultat.

> **Note**
>
> Bien que les finders dynamiques facilitent la gestion des requêtes, ils introduisent de petites contraintes. Vous ne pouvez pas appeler des méthodes `findBy` à partir d'un objet Query. Si vous voulez enchaîner des finders, le finder dynamique doit donc être appelé en premier.

Récupérer les Données Associées

Pour récupérer des données associées, ou filtrer selon les données associées, il y a deux façons de procéder :

- utiliser des fonctions de requêtage de l'ORM de CakePHP telles que `contain()` et `matching()`
- utiliser des fonctions de jointures telles que `innerJoin()`, `leftJoin()`, et `rightJoin()`.

Vous devriez utiliser `contain()` quand vous voulez charger le modèle primaire et ses données associées. Bien que `contain()` vous permette d'appliquer des conditions supplémentaires sur les associations chargées, vous ne pouvez pas filtrer le modèle primaire en fonction des données associées. Pour plus de détails sur `contain()`, consultez *Eager Loading des Associations Via Contain*.

Vous devriez utiliser `matching()` quand vous souhaitez filtrer le modèle primaire en fonction des données associées. Par exemple, quand vous voulez charger tous les articles auxquels est associé un tag spécifique. Pour plus de détails sur `matching()`, consultez *Filtrer par les Données Associées Via Matching et Joins*.

Si vous préférez utiliser les fonctions de jointure, vous pouvez consulter *Ajouter des Jointures* pour plus d'informations.

Eager Loading des Associations Via Contain

Par défaut, CakePHP ne charge **aucune** donnée associée lors de l'utilisation de `find()`. Vous devez faire un "contain" ou charger en eager chaque association que vous souhaitez voir figurer dans vos résultats.

L'eager loading aide à éviter la plupart des problèmes potentiels de performance qui entourent le lazy loading dans un ORM. Les requêtes générées par eager loading peuvent davantage tirer parti des jointures, ce qui permet de créer des requêtes plus efficaces. Dans CakePHP, vous utilisez la méthode "contain" pour indiquer quelles associations doivent être chargées en eager :

```
// Dans un controller ou une méthode de table.

// En option du find()
$query = $articles->find('all', ['contain' => ['Authors', 'Comments']]);

// En méthode sur un objet query
$query = $articles->find('all');
$query->contain(['Authors', 'Comments']);
```

Ceci va charger les auteurs et commentaires liés à chaque article du *result set*. Vous pouvez charger des associations imbriquées en utilisant les tableaux imbriqués pour définir les associations à charger :

```
$query = $articles->find()->contain([
    'Authors' => ['Addresses'], 'Comments' => ['Authors']
]);
```

Au choix, vous pouvez aussi exprimer des associations imbriquées en utilisant la notation par points :

```
$query = $articles->find()->contain([
    'Authors.Addresses',
    'Comments.Authors'
]);
```

Vous pouvez charger les associations en eager aussi profondément que vous le souhaitez :

```
$query = $produits->find()->contain([
    'Shops.Cities.Countries',
    'Shops.Managers'
]);
```

Vous pouvez sélectionner des champs de toutes les associations en utilisant plusieurs appels à `contain()` :

```
$query = $this->find()->select([
    'Realestates.id',
    'Realestates.title',
    'Realestates.description'
])
->contain([
    'RealestatesAttributes' => [
        'Attributes' => [
            'fields' => [
                // Les champs dépendant d'un alias doivent inclure le préfixe
                // du modèle dans contain() pour être mappés correctement.
                'Attributes__name' => 'attr_name'
            ]
        ]
    ]
])
->contain([
    'RealestatesAttributes' => [
        'fields' => [
            'RealestatesAttributes.realestate_id',
            'RealestatesAttributes.value'
        ]
    ]
])
->where($condition);
```

Si vous avez besoin de réinitialiser les *contain* sur une requête, vous pouvez définir le second argument à `true` :

```
$query = $articles->find();
$query->contain(['Authors', 'Comments'], true);
```

Note

Les noms d'association dans les appels à `contain()` doivent respecter la casse (majuscules/minuscules) avec lequelle votre association a été définie, et non pas selon le nom de la propriété utilisée pour accéder aux données associées depuis l'entity. Par exemple, si vous avez déclaré une association par `belongsTo('Users')`, alors vous devez utiliser `contain('Users')` et pas `contain('users')` ni `contain('user')`.

Passer des Conditions à Contain

Avec l'utilisation de `contain()`, vous pouvez restreindre les données retournées par les associations et les filtrer par conditions. Pour spécifier des conditions, passez une fonction anonyme qui reçoit en premier argument la query, de type `\Cake\ORM\Query` :

```
// Dans un controller ou une méthode de table.
$query = $articles->find()->contain('Comments', function (Query $q) {
    return $q
        ->select(['contenu', 'author_id'])
        ->where(['Comments.approved' => true]);
});
```

Cela fonctionne aussi pour la pagination au niveau du Controller :

```
$this->paginate['contain'] = [
    'Comments' => function (Query $query) {
        return $query->select(['body', 'author_id'])
        ->where(['Comments.approved' => true]);
    }
];
```

Avertissement

Si vous constatez qu'il manque des entités associées, vérifiez que les champs de clés étrangères sont bien sélectionnés dans la requête. Sans les clés étrangères, l'ORM ne peut pas retrouver les lignes correspondantes.

Il est aussi possible de restreindre les associations imbriquées en utilisant la notation par point :

```
$query = $articles->find()->contain([
    'Comments',
    'Authors.Profiles' => function (Query $q) {
        return $q->where(['Profiles.is_published' => true]);
    }
]);
```

Dans cet exemple, vous obtiendrez les auteurs même s'ils n'ont pas de profil publié. Pour ne récupérer que les auteurs avec un profil publié, utilisez *matching()*.

Si vous avez des finders personnalisés dans votre table associée, vous pouvez les utiliser à l'intérieur de `contain()` :

```
// Récupère tous les articles, mais récupère seulement les commentaires qui
// sont approuvés et populaires.
$query = $articles->find()->contain('Comments', function ($q) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
    return $q->find('approved')->find('popular');
});
```

> **Note**
>
> Pour les associations `BelongsTo` et `HasOne`, seules les clauses `where` et `select` sont utilisées lors du chargement par `contain()`. Avec `HasMany` et `BelongsToMany`, toutes les clauses sont valides, telles que `order()`.

Vous pouvez contrôler plus que les simples clauses utilisées par `contain()`. Si vous passez un tableau avec l'association, vous pouvez surcharger `foreignKey`, `joinType` et `strategy`. Reportez-vous à *Associations - Lier les Tables Ensemble* pour plus de détails sur la valeur par défaut et les options de chaque type d'association.

Vous pouvez passer `false` comme nouvelle valeur de `foreignKey` pour désactiver complètement les contraintes liées aux clés étrangères. Utilisez l'option `queryBuilder` pour personnaliser la requête quand vous passez un tableau :

```
$query = $articles->find()->contain([
    'Authors' => [
        'foreignKey' => false,
        'queryBuilder' => function (Query $q) {
            return $q->where(/* ... */); // Conditions complètes pour le filtrage
        }
    ]
]);
```

Si vous avez limité les champs que vous chargez avec `select()` mais que vous souhaitez aussi charger les champs des associations avec contain, vous pouvez passer l'objet association à `select()` :

```
// Sélectionne id & title de articles, mais aussi tous les champs de Users.
$query = $articles->find()
    ->select(['id', 'title'])
    ->select($articles->Users)
    ->contain(['Users']);
```

Autre possibilité, si vous avez des associations multiples, vous pouvez utiliser `enableAutoFields()` :

```
// Sélectionne id & title de articles, mais tous les champs de
// Users, Comments et Tags.
$query->select(['id', 'title'])
    ->contain(['Comments', 'Tags'])
    ->enableAutoFields(true)
    ->contain(['Users' => function(Query $q) {
        return $q->autoFields(true);
    }]);
```

Trier les Associations Contain

Quand vous chargez des associations HasMany et BelongsToMany, vous pouvez utiliser l'option `sort` pour trier les données dans ces associations :

```
$query->contain([
    'Comments' => [
```

(suite sur la page suivante)

(suite de la page précédente)

```
        'sort' => ['Comments.created' => 'DESC']
    ]
]);
```

Filtrer par les Données Associées Via Matching et Joins

Un cas de requête couramment utilisé avec les associations consiste à trouver les enregistrements qui correspondent à certaines données associées. Par exemple si vous avez une association "Articles belongsToMany Tags", vous voudrez probablement trouver les Articles qui portent le tag *CakePHP*. C'est extrêmement simple à faire avec l'ORM de CakePHP :

```
// Dans un controller ou une méthode de table.

$query = $articles->find();
$query->matching('Tags', function ($q) {
    return $q->where(['Tags.name' => 'CakePHP']);
});
```

Vous pouvez aussi appliquer cette stratégie aux associations HasMany. Par exemple si "Authors HasMany Articles", vous pouvez trouver tous les auteurs ayant publié un article récemment en écrivant ceci :

```
$query = $authors->find();
$query->matching('Articles', function ($q) {
    return $q->where(['Articles.created >=' => new DateTime('-10 days')]);
});
```

La syntaxe de `contain()`, qui doit déjà vous être familière, permet aussi de filtrer des associations imbriquées :

```
// Dans un controller ou une méthode de table.
$query = $produits->find()->matching(
    'Shops.Cities.Countries', function ($q) {
        return $q->where(['Countries.name' => 'Japon']);
    }
);

// Récupère les articles qui ont été commentés par 'markstory',
// en passant une variable.
// Utiliser la notation par points plutôt que des appels imbriqués à matching()
$username = 'markstory';
$query = $articles->find()->matching('Comments.Users', function ($q) use ($username) {
    return $q->where(['username' => $username]);
});
```

Note

Dans la mesure où cette fonction va créer un `INNER JOIN`, il serait judicieux d'utiliser `distinct` dans la requête. Sinon, vous risquez d'obtenir des doublons si les conditions posées ne l'excluent pas par principe. Dans notre exemple, cela peut être le cas si un utilisateur commente plusieurs fois le même article.

Les données des associations qui correspondent aux conditions (données *matchées*) seront disponibles dans l'attribut `_matchingData` des entities. Si vous utilisez à la fois `match` et `contain` sur la même association, vous pouvez vous attendre à avoir à la fois la propriété `_matchingData` et la propriété standard d'association dans vos résultats.

Utiliser innerJoinWith

Utiliser la fonction `matching()`, comme nous l'avons vu précédemment, va créer un `INNER JOIN` avec l'association spécifiée et va aussi charger les champs dans un ensemble de résultats.

Il peut arriver que vous veuillez utiliser `matching()` mais que vous n'êtes pas intéressé par le chargement des champs de l'association. Dans ce cas, vous pouvez utiliser `innerJoinWith()` :

```
$query = $articles->find();
$query->innerJoinWith('Tags', function ($q) {
    return $q->where(['Tags.name' => 'CakePHP']);
});
```

La méthode `innerJoinWith()` fonctionne de la même manière que `matching()`, ce qui signifie que vous pouvez utiliser la notation par points pour faire des jointures pour les associations imbriquées profondément :

```
$query = $products->find()->innerJoinWith(
    'Shops.Cities.Countries', function ($q) {
        return $q->where(['Countries.name' => 'Japon']);
    }
);
```

Si vous voulez à la fois poser des conditions sur certains champs de l'association et charger d'autres champs de cette même association, vous pouvez parfaitement combiner `innerJoinWith()` et `contain()`. L'exemple ci-dessous filtre les Articles qui ont des Tags spécifiques et charge ces Tags :

```
$filter = ['Tags.name' => 'CakePHP'];
$query = $articles->find()
    ->distinct($articles->getPrimaryKey())
    ->contain('Tags', function (Query $q) use ($filter) {
        return $q->where($filter);
    })
    ->innerJoinWith('Tags', function (Query $q) use ($filter) {
        return $q->where($filter);
    });
```

Note

Si vous utilisez `innerJoinWith()` et que vous voulez sélectionner des champs de cette association avec `select()`, vous devez utiliser un alias pour les noms des champs :

```
$query
    ->select(['country_name' => 'Countries.name'])
    ->innerJoinWith('Countries');
```

Sinon, vous verrez les données dans `_matchingData`, comme cela a été décrit ci-dessous à propos de `matching()`. C'est un angle mort de `matching()`, qui ne sait pas que vous avez sélectionné des champs.

Avertissement

Vous ne devez pas combiner `innerJoinWith()` and `matching()` pour la même association. Cela produirait de multiple requêtes `INNER JOIN` et ne réaliserait pas ce que vous en attendez.

Utiliser notMatching

L'opposé de `matching()` est `notMatching()`. Cette fonction va changer la requête pour qu'elle filtre les résultats qui n'ont pas de relation avec l'association spécifiée :

```
// Dans un controller ou une méthode de table.

$query = $articlesTable
    ->find()
    ->notMatching('Tags', function ($q) {
        return $q->where(['Tags.name' => 'ennuyeux']);
    });
```

L'exemple ci-dessus va trouver tous les articles qui n'ont pas été taggés avec le mot `ennuyeux`. Vous pouvez aussi utiliser cette méthode avec les associations HasMany. Vous pouvez, par exemple, trouver tous les auteurs qui n'ont publié aucun article dans les 10 derniers jours :

```
$query = $authorsTable
    ->find()
    ->notMatching('Articles', function ($q) {
        return $q->where(['Articles.created >=' => new \DateTime('-10 days')]);
    });
```

Il est aussi possible d'utiliser cette méthode pour filtrer les enregistrements qui ne matchent pas des associations imbriquées. Par exemple, vous pouvez trouver les articles qui n'ont pas été commentés par un utilisateur précis :

```
$query = $articlesTable
    ->find()
    ->notMatching('Comments.Users', function ($q) {
        return $q->where(['username' => 'jose']);
    });
```

Puisque les articles n'ayant absolument aucun commentaire satisfont aussi cette condition, vous aurez intérêt à combiner `matching()` et `notMatching()` dans cette requête. L'exemple suivant recherchera les articles ayant au moins un commentaire, mais non commentés par un utilisateur précis :

```
$query = $articlesTable
    ->find()
    ->notMatching('Comments.Users', function ($q) {
        return $q->where(['username' => 'jose']);
    })
    ->matching('Comments');
```

> **Note**
>
> Comme `notMatching()` va créer un `LEFT JOIN`, vous pouvez envisager d'appeler `distinct` sur la requête pour éviter d'obtenir des lignes dupliquées.

Gardez à l'esprit que le contraire de la fonction `matching()`, `notMatching()`, ne va pas ajouter de données à la propriété `_matchingData` dans les résultats.

Utiliser leftJoinWith

Dans certaines situations, vous aurez à calculer un résultat à partir d'une association, sans avoir à charger tous ses enregistrements. Par exemple, si vous voulez charger le nombre total de commentaires d'un article, en parallèle des données de l'article, vous pouvez utiliser la fonction `leftJoinWith()` :

```
$query = $articlesTable->find();
$query->select(['total_comments' => $query->func()->count('Comments.id')])
    ->leftJoinWith('Comments')
    ->group(['Articles.id'])
    ->enableAutoFields(true);
```

Le résultat de cette requête contiendra les données de l'article et la propriété `total_comments` pour chacun d'eux.

`leftJoinWith()` peut aussi être utilisée avec des associations imbriquées. C'est utile par exemple pour rechercher, pour chaque auteur, le nombre d'articles taggés avec un certain mot :

```
$query = $authorsTable
    ->find()
    ->select(['total_articles' => $query->func()->count('Articles.id')])
    ->leftJoinWith('Articles.Tags', function ($q) {
        return $q->where(['Tags.name' => 'redoutable']);
    })
    ->group(['Authors.id'])
    ->enableAutoFields(true);
```

Cette fonction ne va charger aucune colonne des associations spécifiées dans les résultats.

Changer les Stratégies de Récupération

Comme cela a été dit, vous pouvez personnaliser la stratégie (*strategy*) utilisée par une association dans un `contain()`.

Si vous observez les options des *associations* `BelongsTo` et `HasOne`, vous constaterez que la stratégie par défaut "join" et le type de jointure (`joinType`) "INNER" peuvent être remplacés par "select" :

```
$query = $articles->find()->contain([
    'Comments' => [
        'strategy' => 'select',
    ]
]);
```

Cela peut être utile lorsque vous avez besoin de conditions qui ne font pas bon ménage avec une jointure. Cela ouvre aussi la possibilité de requêter des tables entre lesquelles vous n'avez pas l'autorisation de faire des jointures, par exemple des tables se trouvant dans deux bases de données différentes.

Habituellement, vous définisser la stratégie d'une association quand vous la définissez, dans la méthode `Table::initialize()`, mais vous pouvez changer manuellement la stratégie de façon permanente :

```
$articles->Comments->setStrategy('select');
```

Récupération Avec la Stratégie de Sous-Requête

Quand vos tables grandissent en taille, la récupération des associations peut ralentir sensiblement, en particulier si vous faites de grandes requêtes en une fois. Un bon moyen d'optimiser le chargement des associations `hasMany` et `belongsToMany` est d'utiliser la stratégie `subquery` :

```
$query = $articles->find()->contain([
    'Comments' => [
            'strategy' => 'subquery',
            'queryBuilder' => function ($q) {
                return $q->where(['Comments.approved' => true]);
            }
    ]
]);
```

Le résultat va rester le même que pour la stratégie par défaut, mais ceci peut grandement améliorer la requête et son temps de récupération dans certaines bases de données, en particulier cela va permettre de récupérer des grandes portions de données en même temps dans les bases de données qui limitent le nombre de paramètres liés par requête, comme **Microsoft SQL Server**.

Lazy loading des Associations

CakePHP propose le chargement de vos associations en eager. Cependant il y a des cas où vous aurez besoin de charger les associations en lazy. Consultez les sections *Lazy Loading des Associations* et *Charger des Associations Supplémentaires* pour plus d'informations.

Travailler sur les Résultats (Result Sets)

Une fois qu'une requête est exécutée avec `all()`, vous récupérez une instance de `Cake\ORM\ResultSet`. Cet objet propose des outils puissants pour manipuler les données renvoyées par vos requêtes. Comme les objets Query, un ResultSet est une *Collection* et vous pouvez donc appeler sur celui-ci n'importe quelle méthode propre aux collections.

Les objets ResultSet vont charger les lignes paresseusement (*lazily*) à partir de la requête préparée sous-jacente. Par défaut, les résultats seront mis en mémoire tampon, vous permettant ainsi de parcourir les résultats plusieurs fois, ou de mettre les résultats en cache et d'itérer dessus. Si vous travaillez sur un ensemble de données trop large pour tenir en mémoire, vous pouvez désactiver la mise en mémoire tampon depuis la requête pour travailler sur les résultats à la volée :

```
$query->disableBufferedResults();
```

Stopper la mise en mémoire tampon appelle quelques mises en garde :

1. Vous ne pourrez itérer les résultats qu'une seule fois.
2. Vous ne pourrez pas non plus itérer et mettre en cache les résultats.
3. La mise en mémoire tampon ne peut pas être désactivée pour les requêtes qui chargent en eager les associations hasMany ou belongsToMany, puisque ces types d'association nécessitent le chargement en eager de tous les résultats afin de générer les requêtes dépendantes.

Avertissement

Traiter les résultats à la volée continuera d'allouer l'espace mémoire nécessaire pour la totalité des résultats lorsque vous utilisez PostgreSQL et SQL Server. Ceci est dû à des limitations dans PDO.

Les ResultSets vous permettent de mettre en cache, de sérialiser ou d'encoder en JSON les résultats :

```
// Dans un controller ou une méthode de table.
$results = $query->all();

// Serialisé
$serialized = serialize($results);
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Json
$json = json_encode($results);
```

La sérialisation des ResultSets et l'encodage en JSON fonctionnent comme vous pouvez vous y attendre. Les données sérialisées peuvent être désérialisées en un ResultSet fonctionnel. La conversion en JSON respecte la configuration des champs cachés et des champs virtuels dans tous les objets entity inclus dans le ResultSet.

Les ResultSets sont des objets "Collection" et supportent les mêmes méthodes que les *objets collection*. Par exemple, vous pouvez extraire une liste des tags uniques sur une collection d'articles en exécutant :

```
// Dans un controller ou une méthode de table.
$query = $articles->find()->contain(['Tags']);

$reducer = function ($output, $value) {
    if (!in_array($value, $output)) {
        $output[] = $value;
    }

    return $output;
};

$uniqueTags = $query->all()
    ->extract('tags.name')
    ->reduce($reducer, []);
```

Ci-dessous quelques autres exemples de méthodes de collection utilisées avec des ResultSets :

```
// Filtre les lignes sur une propriété calculée
$filtered = $results->filter(function ($row) {
    return $row->is_recent;
});

// Crée un tableau associatif depuis les propriétés du résultat
$results = $articles->find()->contain(['Authors'])->all();

$authorsList = $results->combine('id', 'author.name');
```

Le chapitre *Collections* comporte plus de détails sur ce qu'on peut faire avec les fonctionnalités des collections sur des ResultSets. La section *Ajouter des Champs Calculés* montre comment ajouter des champs calculés, ou remplacer le ResutSet.

Récupérer le Premier & Dernier enregistrement à partir d'un ResultSet

Vous pouvez utiliser les méthodes `first()` et `last()` pour récupérer respectivement le premier et le dernier enregistrement d'un ResultSet :

```
$result = $articles->find('all')->all();

// Récupère le premier et/ou le dernier résultat.
$row = $result->first();
$row = $result->last();
```

Récupérer une Ligne Spécifique d'un ResultSet

Vous pouvez utiliser `skip()` et `first()` pour récupérer un enregistrement arbitraire à partir d'un ensemble de résultats :

```
$result = $articles->find('all')->all();

// Récupère le 5ème enregistrement
$row = $result->skip(4)->first();
```

Vérifier si une Requête ou un ResultSet est vide

Vous pouvez utiliser la méthode `isEmpty()` sur un objet Query ou ResultSet pour savoir s'il contient au moins une ligne. Appeler `isEmpty()` sur un objet Query va exécuter la requête :

```
// Vérifie une requête.
$query->isEmpty();

// Vérifie les résultats.
$results = $query->all();
$results->isEmpty();
```

Charger des Associations Supplémentaires

Une fois que vous avez créé un ResultSet, vous pourriez vouloir charger en eager des associations supplémentaires. C'est le moment idéal pour charger paresseusement des données en eager. Vous pouvez charger des associations supplémentaires en utilisant `loadInto()` :

```
$articles = $this->Articles->find()->all();
$withMore = $this->Articles->loadInto($articles, ['Comments', 'Users']);
```

Vous pouvez charger en eager des données additionnelles dans une entity unique ou une collection d'entities.

Modifier les Résultats avec Map/Reduce

La plupart du temps, les opérations `find` nécessitent un traitement après-coup des données de la base de données. Les accesseurs (*getters*) des entities peuvent s'occuper de générer la plupart des propriétés virtuelles ou des formats de données particuliers ; néanmoins vous serez parfois amené à changer la structure des données de façon plus radicale.

Pour ces situations, l'objet `Query` propose la méthode `mapReduce()`, qui est une façon de traiter les résultats après qu'ils ont été récupérés dans la base de données.

Un exemple classique de changement de structure de données est le regroupement des résultats selon certaines conditions. Nous pouvons utiliser la fonction `mapReduce()` pour cette tâche. Nous avons besoin de deux fonctions appelées `$mapper` et `$reducer`. La fonction `$mapper` reçoit en premier argument un résultat de la base de données, en second argument la clé d'itération et en troisième elle reçoit une instance de la routine `MapReduce` en train d'être exécutée :

```
$mapper = function ($article, $key, $mapReduce) {
    $status = 'published';
    if ($article->isDraft() || $article->isInReview()) {
        $status = 'unpublished';
    }
    $mapReduce->emitIntermediate($article, $status);
};
```

Dans cet exemple, `$mapper` calcule le statut d'un article, soit publié soit non publié. Ensuite il appelle `emitIntermediate()` sur l'instance `MapReduce`. Cette méthode insère l'article dans la liste des articles soit sous l'étiquette "publié", soit sous l'étiquette "non publié".

La prochaine étape dans le processus de map-reduce est la consolidation des résultats finaux. La fonction `$reducer` sera appelée sur chaque statut créé dans le mapper, de manière à ce que vous puissiez faire des traitements supplémentaires. Cette fonction va recevoir la liste des articles d'un certain « tas » en premier paramètre, le nom du tas à traiter en second paramètre, et à nouveau, comme pour la fonction `mapper()`, l'instance de la routine `MapReduce` en troisième paramètre. Dans notre exemple, nous n'avons pas besoin de retraiter les listes d'articles une fois qu'elles sont constituées, donc nous nous contentons d'émettre (*emit*) les résultats finaux :

```
$reducer = function ($articles, $status, $mapReduce) {
    $mapReduce->emit($articles, $status);
};
```

Pour finir, nous pouvons passer ces deux fonctions pour exécuter le regroupement :

```
$articlesByStatus = $articles->find()
    ->where(['author_id' => 1])
    ->mapReduce($mapper, $reducer)
    ->all();

foreach ($articlesByStatus as $status => $articles) {
    echo sprintf("Il y a %d articles avec le statut %s", count($articles), $status);
}
```

Ce qui va afficher la sortie suivante :

```
Il y a 4 articles avec le statut published
Il y a 5 articles avec le statut unpublished
```

Bien sûr, ceci est un exemple simple qui pourrait être résolu d'une autre façon sans l'aide d'un traitement map-reduce. Maintenant, regardons un autre exemple dans lequel la fonction reducer sera nécessaire pour faire quelque chose de plus que d'émettre les résultats.

Pour calculer les mots mentionnés le plus souvent dans les articles contenant des informations sur CakePHP, comme d'habitude nous avons besoin d'une fonction mapper :

```
$mapper = function ($article, $key, $mapReduce) {
    if (stripos($article['body'], 'cakephp') === false) {
        return;
    }

    $words = array_map('strtolower', explode(' ', $article['body']));
    foreach ($words as $word) {
        $mapReduce->emitIntermediate($article['id'], $word);
    }
};
```

Elle vérifie d'abord si le mot « cakephp » est dans le corps de l'article, et ensuite coupe le corps en mots individuels. Chaque mot va créer son propre `tas`, où chaque id d'article sera stocké. Maintenant réduisons nos résultats pour extraire seulement le décompte du nombre de mots :

```
$reducer = function ($occurrences, $word, $mapReduce) {
    $mapReduce->emit(count($occurrences), $word);
}
```

Pour finir, nous mettons tout ensemble :

```
$nbMots = $articles->find()
    ->where(['published' => true])
    ->andWhere(['publication_date >=' => new DateTime('2014-01-01')])
    ->disableHydration()
    ->mapReduce($mapper, $reducer)
    ->all();
```

Ceci pourrait retourner un tableau très grand si nous ne purgeons pas les petits mots, mais cela pourrait ressembler à ceci :

```
[
    'cakephp' => 100,
    'génial' => 39,
    'impressionnant' => 57,
    'remarquable' => 10,
    'hallucinant' => 83
]
```

Un dernier exemple et vous serez un expert de map-reduce. Imaginez que vous ayez une table `amis` et que vous souhaitiez trouver les « faux amis » dans notre base de données ou, autrement dit, des gens qui ne se suivent pas mutuellement. Commençons avec notre fonction `mapper()` :

```
$mapper = function ($rel, $key, $mr) {
    $mr->emitIntermediate($rel['target_user_id'], $rel['source_user_id']);
    $mr->emitIntermediate(-$rel['source_user_id'], $rel['target_user_id']);
};
```

Le tableau intermédiaire ressemblera à ceci :

```
[
    1 => [2, 3, 4, 5, -3, -5],
    2 => [-1],
    3 => [-1, 1, 6],
    4 => [-1],
    5 => [-1, 1],
    6 => [-3],
    ...
]
```

La clé de premier niveau étant un utilisateur, les nombres positifs indiquent que l'utilisateur suit d'autres utilisateurs et les nombres négatifs qu'il est suivi par d'autres utilisateurs.

Maintenant, il est temps de la réduire. Pour chaque appel au reducer, il va recevoir une liste de followers par utilisateur :

```
$reducer = function ($friends, $user, $mr) {
    $fakeFriends = [];

    foreach ($friends as $friend) {
        if ($friend > 0 && !in_array(-$friend, $friends)) {
            $fakeFriends[] = $friend;
        }
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    if ($fakeFriends) {
        $mr->emit($fakeFriends, $user);
    }
};
```

Et nous fournissons nos fonctions à la requête :

```
$fakeFriends = $friends->find()
    ->disableHydration()
    ->mapReduce($mapper, $reducer)
    ->all();
```

Ceci retournerait un tableau ressemblant à :

```
[
    1 => [2, 4],
    3 => [6]
    ...
]
```

Ce tableau final signifie, par exemple, que l'utilisateur avec l'id 1 suit les utilisateurs 2 et 4, mais ceux-ci ne suivent pas 1 de leur côté.

Empiler Plusieurs Opérations

L'utilisation de *mapReduce* dans une requête ne va pas l'exécuter immédiatement. L'opération va être enregistrée pour être lancée dès que l'on tentera de récupérer le premier résultat. Ceci vous permet de continuer à chainer les méthodes et les filtres sur la requête même après avoir ajouté une routine map-reduce :

```
$query = $articles->find()
    ->where(['published' => true])
    ->mapReduce($mapper, $reducer);

// Plus loin dans votre app:
$query->where(['created >=' => new DateTime('1 day ago')]);
```

C'est particulièrement utile pour construire des méthodes finder personnalisées comme décrit dans la section *Méthodes Finder Personnalisées* :

```
public function findPublished(Query $query, array $options)
{
    return $query->where(['published' => true]);
}

public function findRecent(Query $query, array $options)
{
    return $query->where(['created >=' => new DateTime('1 day ago')]);
}

public function findCommonWords(Query $query, array $options)
{
    // Comme dans l'exemple précédent sur la fréquence des mots
    $mapper = ...;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    $reducer = ...;

    return $query->mapReduce($mapper, $reducer);
}

$motsCourant = $articles
    ->find('commonWords')
    ->find('published')
    ->find('recent');
```

De plus, il est aussi possible d'empiler plusieurs opérations `mapReduce` pour une même requête. Par exemple, si nous souhaitons avoir les mots les plus couramment utilisés pour les articles, mais ensuite les filtrer pour retourner uniquement les mots qui étaient mentionnés plus de 20 fois tout au long des articles :

```
$mapper = function ($count, $word, $mr) {
    if ($count > 20) {
        $mr->emit($count, $word);
    }
};

$articles->find('commonWords')->mapReduce($mapper)->all();
```

Retirer Toutes les Opérations Map-reduce Empilées

Dans certaines circonstances vous pourriez vouloir modifier un objet `Query` pour que les opérations `mapReduce` prévues ne soient pas exécutées du tout. Vous pouvez le faire en appelant la méthode avec les deux paramètres à null et le troisième paramètre (overwrite) à `true` :

```
$query->mapReduce(null, null, true);
```

Valider des Données

Avant que vous ne *sauvegardiez des données* vous voudrez probablement vous assurer que les données sont correctes et cohérentes. Dans CakePHP, nous avons deux étapes de validation :

1. Avant que les données de requête ne soient converties en entities, il est possible d'appliquer des règles de validation concernant le type de données et leur format.
2. Avant que les données ne soient sauvegardées, il est possible d'appliquer des règles du domaine ou de l'application. Ces règles aident à garantir la cohérence des données de votre application.

Valider les Données Avant de Construire les Entities

Vous pouvez valider les données lors de leur transformation en entities. Cette validation vous permet de vérifier le type, la forme et la taille des données. Par défaut, les données de la requête seront validées avant d'être converties en entities. Si une ou plusieurs règles de validation échouent, l'entity retournée contiendra des erreurs. Les champs comportant des erreurs ne figureront pas dans l'entity renvoyée :

```
$article = $articles->newEntity($this->request->getData());
if ($article->getErrors()) {
    // La validation de l'entity a échoué.
}
```

Voici ce qui se passe quand vous construisez une entity et que la validation est activée :

1. L'objet validator est créé.
2. Les fournisseurs de validation (*providers*) `table` et `default` sont attachés.
3. La méthode de validation désignée est appelée. Par exemple `validationDefault()`.
4. L'événement `Model.buildValidator` est déclenché.
5. Les données de la requête sont validées.
6. Les données de la requête sont castées en types correspondant aux types des colonnes.
7. Les erreurs sont définies dans l'entity.
8. Les données valides sont définies dans l'entity, tandis que les champs qui ont échoué à la validation sont laissés de côté.

Si vous voulez désactiver la validation lors de la conversion des données de la requête, définissez l'option `validate` à false :

```
$article = $articles->newEntity(
    $this->request->getData(),
    ['validate' => false]
);
```

La méthode `patchEntity()` fonctionne de façon identique :

```
$article = $articles->patchEntity($article, $newData, [
    'validate' => false
]);
```

Créer un Ensemble de Validation par Défaut

Les règles de validation sont définies dans la classe Table par commodité. Cela définit quelles données doivent être validées en conjonction avec l'endroit où elles seront sauvegardées.

Pour créer un objet de validation par défaut dans votre table, créez la fonction `validationDefault()` :

```
use Cake\ORM\Table;
use Cake\Validation\Validator;

class ArticlesTable extends Table
{
    public function validationDefault(Validator $validator): Validator
    {
        $validator
            ->requirePresence('title', 'create')
            ->notEmptyString('title');

        $validator
            ->allowEmptyString('link')
            ->add('link', 'valid-url', ['rule' => 'url']);

        ...

        return $validator;
    }
}
```

Les méthodes et règles de validation disponibles proviennent de la classe `Validator` et sont documentées dans la section *Créer les Validators*.

> **Note**
>
> Les objets de validation sont principalement destinés à valider les données provenant de l'utilisateur, par exemple les formulaires ou d'autres données postées dans la requête.

Utiliser un Ensemble de Validation Différent

Outre la possibilité de désactiver la validation, vous pouvez choisir quel ensemble de règles de validation que vous souhaitez appliquer :

```
$article = $articles->newEntity(
    $this->request->getData(),
    ['validate' => 'update']
);
```

Ceci appellerait la méthode `validationUpdate()` sur l'instance de table pour construire les règles requises. Par défaut, c'est la méthode `validationDefault()` qui sera utilisée. Voici un exemple de validator pour notre table `articles` :

```
class ArticlesTable extends Table
{
    public function validationUpdate($validator)
    {
        $validator
            ->notEmptyString('title', __('Vous devez indiquer un titre'))
            ->notEmptyString('body', __('Un contenu est nécessaire'));

        return $validator;
    }
}
```

Vous pouvez avoir autant d'ensembles de validation que vous le souhaitez. Consultez le *chapitre sur la validation* pour plus d'informations sur la construction des ensembles de règles de validation.

Utiliser un Ensemble de Validation Différent pour les Associations

Les ensembles de validation peuvent également être définis pour chaque association. Lorsque vous utilisez les méthodes `newEntity()` ou `patchEntity()`, vous pouvez passer des options supplémentaires à chaque association qui doit être convertie :

```
$data = [
     'title' => 'Mon titre',
     'body' => 'Le texte',
     'user_id' => 1,
     'user' => [
         'username' => 'marc'
     ],
     'comments' => [
         ['body' => 'Premier commentaire'],
         ['body' => 'Second commentaire'],
     ]
 ];
```

(suite sur la page suivante)

(suite de la page précédente)

```
$article = $articles->patchEntity($article, $data, [
    'validate' => 'update',
    'associated' => [
        'Users' => ['validate' => 'signup'],
        'Comments' => ['validate' => 'custom']
    ]
]);
```

Combiner les Validators

Grâce à la manière dont les objets validator sont construits, vous pouvez diviser leur process de construction en petites étapes réutilisables :

```
// UsersTable.php

public function validationDefault(Validator $validator): Validator
{
    $validator->notEmptyString('username');
    $validator->notEmptyString('password');
    $validator->add('email', 'valid-email', ['rule' => 'email']);
    ...

    return $validator;
}

public function validationHardened(Validator $validator): Validator
{
    $validator = $this->validationDefault($validator);

    $validator->add('password', 'length', ['rule' => ['lengthBetween', 8, 100]]);

    return $validator;
}
```

Avec cette configuration, lors de l'utilisation de l'ensemble de validation `hardened`, vous aurez aussi les règles de l'ensemble `default`.

Validation Providers

Les règles de validation peuvent utiliser des fonctions définies par n'importe quel provider connu. Par défaut, CakePHP définit quelques providers :

1. Les méthodes sur la classe de table, ou ses behaviors, sont disponibles dans le provider `table`.
2. La classe du cœur `Validation` est configurée en tant que provider `default`.

En créant une règle de validation, vous pouvez désigner le provider de cette règle. Par exemple, si votre table a une méthode `isValidRole`, vous pouvez l'utiliser comme une règle de validation :

```
use Cake\ORM\Table;
use Cake\Validation\Validator;

class UsersTable extends Table
{
```

(suite sur la page suivante)

(suite de la page précédente)

```
    public function validationDefault(Validator $validator): Validator
    {
        $validator
            ->add('role', 'validRole', [
                'rule' => 'isValidRole',
                'message' => __('Vous devez fournir un rôle valide'),
                'provider' => 'table',
            ]);

        return $validator;
    }

    public function isValidRole($value, array $context): bool
    {
        return in_array($value, ['admin', 'editor', 'author'], true);
    }

}
```

Vous pouvez également utiliser des closures en tant que règles de validation :

```
$validator->add('name', 'myRule', [
    'rule' => function ($value, array $context) {
        if ($value > 1) {
            return true;
        }

        return 'Valeur incorrecte.';
    }
]);
```

Les méthodes de validation peuvent renvoyer des messages d'erreur lorsqu'elles échouent. C'est un moyen simple de créer des messages d'erreur dynamiques basés en fonction de la valeur fournie.

Récupérer des Validators depuis les Tables

Une fois que vous avez créé quelques ensembles de validation dans votre classe de table, vous pouvez récupérer l'objet résultant par son nom :

```
$defaultValidator = $usersTable->getValidator('default');

$hardenedValidator = $usersTable->getValidator('hardened');
```

Classe Validator par Défaut

Comme mentionné ci-dessus, par défaut les méthodes de validation reçoivent une instance de `Cake\Validation\Validator`. Si vous souhaitez utiliser une instance d'un validator personnalisé, vous pouvez utiliser l'attribut `$_validatorClass` de la table :

```
// Dans votre classe de table
public function initialize(array $config): void
{
```

(suite sur la page suivante)

(suite de la page précédente)

```
    $this->_validatorClass = \FullyNamespaced\Custom\Validator::class;
}
```

Appliquer des Règles d'Application

Au-delà de la validation basique des données qui est lancée quand *les données de la requête sont converties en entities*, de nombreuses applications ont des validations plus complexes qui doivent être appliquées après la validation basique.

Là où la validation s'assure que la forme ou la syntaxe de vos données sont correctes, les règles s'attellent à comparer les données avec l'état existant de votre application et/ou du réseau.

Ces types de règles sont souvent appelées "règles de domaine" ou "règles d'application". CakePHP utilise ce concept avec les "RulesCheckers" qui sont appliquées avant que les entities ne soient sauvegardées. Voici quelques exemples de règles de domaine :

- S'assurer qu'un email est unique.
- Transition d'états ou étapes de flux de travail, par exemple pour mettre à jour le statut d'une facture.
- Eviter la modification d'articles ayant fait l'objet d'une suppression logique.
- Appliquer des limites d'usage, que ce soit en nombre d'appels total ou en nombre d'appels sur une période donnée.

Les règles de domaine sont vérifiées lors de l'appel aux méthodes `save()` et `delete()` de Table.

Créer un Vérificateur de Règles

Les classes de vérificateur de règles (*rules checkers*) sont généralement définies par la méthode `buildRules()` dans votre classe de table. Les behaviors et les autres souscripteurs d'events peuvent utiliser l'événement `Model.buildRules` pour ajouter des règles au vérificateur pour une classe Table donnée :

```
use Cake\ORM\RulesChecker;

// Dans une classe de table
public function buildRules(RulesChecker $rules): RulesChecker
{
    // Ajoute une règle qui est appliquée pour les opérations de création et de mise à␣
↪jour
    $rules->add(function ($entity, $options) {
        // Retourne un booléen pour indiquer si succès/échec
    }, 'ruleName');

    // Ajoute une règle pour la création.
    $rules->addCreate(function ($entity, $options) {
        // Retourne un booléen pour indiquer si succès/échec
    }, 'ruleName');

    // Ajoute une règle pour la mise à jour.
    $rules->addUpdate(function ($entity, $options) {
        // Retourne un booléen pour indiquer si succès/échec
    }, 'ruleName');

    // Ajoute une règle pour la suppression.
    $rules->addDelete(function ($entity, $options) {
        // Retourne un booléen pour indiquer si succès/échec
    }, 'ruleName');
```

(suite sur la page suivante)

(suite de la page précédente)

```
    return $rules;
}
```

Vos fonctions de règles ont pour paramètres l'Entity à vérifier et un tableau d'options. Le tableau d'options contiendra `errorField`, `message` et `repository`. L'option `repository` contiendra la classe de table à laquelle les règles sont attachées. Comme les règles acceptent n'importe quel `callable`, vous pouvez aussi utiliser des fonctions d'instance :

```
$rules->addCreate([$this, 'uniqueEmail'], 'uniqueEmail');
```

ou des classes callable :

```
$rules->addCreate(new IsUnique(['email']), 'uniqueEmail');
```

Lors de l'ajout de règles, vous pouvez définir en options le champ correspondant à la règle et le message d'erreur :

```
$rules->add([$this, 'isValidState'], 'validState', [
    'errorField' => 'status',
    'message' => 'Cette facture ne peut pas être déplacée vers ce statut.'
]);
```

L'erreur sera visible lors de l'appel à la méthode `getErrors()` dans l'entity :

```
$entity->getErrors(); // Contient les messages d'erreur des règles de domaine
```

Créer des Règles d'Unicité de Champ

Les règles d'unicité étant couramment utilisées, CakePHP inclut une classe simple qui vous permet de définir des ensembles de champs uniques :

```
use Cake\ORM\Rule\IsUnique;

// Un seul champ.
$rules->add($rules->isUnique(['email']));

// Une liste de champs
$rules->add($rules->isUnique(
    ['username', 'account_id'],
    'Cette combinaison `username` & `account_id` est déjà utilisée.'
));
```

Quand vous définissez des règles sur des champs de clé étrangère, il est important de se rappeler que seuls les champs listés sont utilisés dans la règle. L'ensemble des règles d'unicité sera trouvé avec `find('all')`. Cela signifie que la règle ci-dessus ne sera pas déclenchée en définissant `$user->account->id`.

De nombreux moteurs de bases de données autorisent plusieurs valeurs NULL dans les index UNIQUE. Pour simuler ce comportement, définissez l'option `allowMultipleNulls` à `true` :

```
$rules->add($rules->isUnique(
    ['username', 'account_id'],
    ['allowMultipleNulls' => true]
));
```

Règles de Clés Etrangères

Bien que vous puissiez vous reposer sur les erreurs de la base de données pour imposer des contraintes, le fait d'utiliser des règles vous permet de fournir une expérience utilisateur plus sympathique. C'est pour cela que CakePHP inclut une classe de règle `ExistsIn` :

```
// Un champ unique.
$rules->add($rules->existsIn('article_id', 'Articles'));

// Plusieurs clés, utile pour des clés primaires composites.
$rules->add($rules->existsIn(['site_id', 'article_id'], 'Articles'));
```

Les champs dont vous demandez à vérifier l'existence dans la table correspondante doivent faire partie de la clé primaire.

Vous pouvez forcer `existsIn` à accepter qu'une partie de votre clé étrangère composite soit null :

```
// Exemple: NodesTable contient une clé primaire composite (parent_id, site_id).
// Un "Node" peut faire référence à un Node parent mais ce n'est pas
// obligatoire. Dans ce dernier cas, parent_id est null.
// Nous autorisons la validation de cette règle même si les champs qui sont
// nullables, comme parent_id, sont null:
$rules->add($rules->existsIn(
    ['parent_id', 'site_id'], // Schema: parent_id NULL, site_id NOT NULL
    'ParentNodes',
    ['allowNullableNulls' => true]
));

// Un Node doit cependant toujours avoir une référence à un Site.
$rules->add($rules->existsIn(['site_id'], 'Sites'));
```

Dans la majorité des bases de données SQL, les index `UNIQUE` sur plusieurs colonnes autorisent plusieurs valeurs null car `NULL` n'est pas égal à lui même. Même si CakePHP autorise par défaut plusieurs valeurs null, vous pouvez inclure les nulls dans vos vérifications d'unicité en utilisant `allowMultipleNulls` :

```
// Une seule valeur null peut exister dans `parent_id` et `site_id`
$rules->add($rules->existsIn(
    ['parent_id', 'site_id'],
    'ParentNodes',
    ['allowMultipleNulls' => false]
));
```

Règles sur le Nombre de Valeurs d'une Association

Si vous avez besoin de valider qu'une propriété ou une association contient un certain nombre de valeurs, vous pouvez utiliser la règle `validCount()` :

```
// Dans le fichier ArticlesTable.php
// Pas plus de 5 tags sur un article.
$rules->add($rules->validCount('tags', 5, '<=', 'Vous ne pouvez avoir que 5 tags'));
```

Quand vous définissez des règles qui basées sur un nombre de valeurs, le troisième paramètre vous permet de définir l'opérateur de comparaison à utiliser. Les opérateurs acceptés sont ==, >=, <=, >, <, et !=. Pour vérifier que le décompte d'une propriété est entre certaines valeurs, utilisez deux règles :

```
// Dans le fichier ArticlesTable.php
// Entre 3 et 5 tags
$rules->add($rules->validCount('tags', 3, '>=', 'Vous devez avoir au moins 3 tags'));
$rules->add($rules->validCount('tags', 5, '<=', 'Vous devez avoir au plus 5 tags'));
```

Notez que `validCount` retourne `false` si la propriété ne peut pas être comptée ou n'existe pas :

```
// La sauvegarde échouera si tags est null
$rules->add($rules->validCount('tags', 0, '<=', 'Vous ne devez pas avoir de tags'));
```

Règles de Contraintes d'Association

La règle `LinkConstraint` vous permet d'émuler des contraintes SQL dans les bases de données qui ne les supportent pas, ou quand vous voulez fournir des messages d'erreur plus sympathiques quand la contrainte échoue. Cette règle vous permet de vérifier si une association a ou non des enregistrements liés en fonction du mode utilisé :

```
// S'assure que chaque commentaire est lié à un Article lors de sa mise à jour.
$rules->addUpdate($rules->isLinkedTo(
    'Articles',
    'article',
    'Spécifiez un article'
));

// S'assure qu'un article n'a pas de commentaire au moment de sa suppression.
$rules->addDelete($rules->isNotLinkedTo(
    'Comments',
    'comments',
    'Impossible de supprimer un article qui contient des commentaires.'
));
```

Ajouté dans la version 4.0.0.

Utiliser les Méthodes d'Entity en tant que Règles

Vous pouvez utiliser les méthodes d'une entity en tant que règles de domaine :

```
$rules->add(function ($entity, $options) {
    return $entity->isOkLooking();
}, 'ruleName');
```

Utiliser des Règles Conditionnelles

Vous pouvez appliquer des règles conditionnelles en fonction des données de l'entity :

```
$rules->add(function ($entity, $options) use($rules) {
    if ($entity->role == 'admin') {
        $rule = $rules->existsIn('user_id', 'Admins');

        return $rule($entity, $options);
    }
    if ($entity->role == 'user') {
        $rule = $rules->existsIn('user_id', 'Users');
```

(suite sur la page suivante)

(suite de la page précédente)

```
        return $rule($entity, $options);
    }

    return false;
}, 'userExists');
```

Messages d'Erreur Dynamiques/Conditionnels

Les règles, qu'elles soient *des callables personnalisés* ou *des objets Rule*, peuvent soit retourner un booléen indiquant si elles ont réussi, soit retourner une chaîne qui signifie que la règle a échoué et que la chaîne doit être utilisée comme message d'erreur.

Les messages d'erreur possibles définis par l'option `message` seront écrasés par ceux retournés par la règle :

```
$rules->add(
    function ($entity, $options) {
        if (!$entity->length) {
            return false;
        }

        if ($entity->length < 10) {
            return "Message d'erreur quand la valeur est inférieure à 10";
        }

        if ($entity->length > 20) {
            return "Message d'erreur quand la valeur est supérieure à 20";
        }

        return true;
    },
    'ruleName',
    [
        'errorField' => 'length',
        'message' => "Message d'erreur générique utilisé quand la règle retourne `false`.
↪"
    ]
);
```

Note

Notez que pour que le message retourné puisse être utilisé, vous *devez* aussi définir l'option `errorField`, sinon la règle va se contenter d'échouer silencieusement, c'est-à-dire sans insérer le message d'erreur dans l'entity !

Créer des Règles Personnalisées Réutilisables

Vous pouvez vouloir réutiliser des règles de domaine personnalisées. Vous pouvez le faire en créant votre propre règle invokable :

```
use App\ORM\Rule\IsUniqueWithNulls;
// ...
public function buildRules(RulesChecker $rules): RulesChecker
```

(suite sur la page suivante)

(suite de la page précédente)

```
{
    $rules->add(new IsUniqueWithNulls(['parent_id', 'instance_id', 'name']),
↪'uniqueNamePerParent', [
        'errorField' => 'name',
        'message' => 'Doit être unique pour chaque parent.'
    ]);

    return $rules;
}
```

Regardez les règles du cœur pour avoir des exemples sur la façon de créer de telles règles.

Créer des Objets de Règles Personnalisées

Si votre application a des règles qui sont souvent réutilisées, il peut être utile de packager ces règles dans des classes réutilisables :

```
// Dans src/Model/Rule/CustomRule.php
namespace App\Model\Rule;

use Cake\Datasource\EntityInterface;

class CustomRule
{
    public function __invoke(EntityInterface $entity, array $options)
    {
        // Faire le boulot ici
        return false;
    }
}

// Ajouter la règle personnalisée
use App\Model\Rule\CustomRule;

$rules->add(new CustomRule(/* ... */), 'ruleName');
```

En ajoutant des classes de règles personnalisées, vous pouvez garder votre code DRY et tester vos règles de domaine isolément.

Désactiver les Règles

Quand vous sauvegardez une entity, vous pouvez désactiver les règles si c'est nécessaire :

```
$articles->save($article, ['checkRules' => false]);
```

Validation vs. Règles d'Application

L'ORM de CakePHP est unique dans le sens où il utilise une approche à deux couches pour la validation.

La première couche est la validation. Les règles de validation ont pour objet d'opérer sans état (*stateless*). Elles servent à s'assurer que la forme, les types de données et le format des données sont corrects.

La seconde couche est celle des règles d'application. Les règles d'application sont plus appropriés pour vérifier l'état des propriétés de vos entities. Par exemple, les règles de validation peuvent permettre de s'assurer qu'une adresse email

est valide, tandis qu'une règle d'application permet de s'assurer que l'adresse email est unique.

Comme vous avez pu le voir, la première couche est réalisée par l'objet `Validator` lors de l'appel à `newEntity()` ou `patchEntity()` :

```
$validatedEntity = $articlesTable->newEntity(
    $donneesDouteuses,
    ['validate' => 'maRegle']
);
$validatedEntity = $articlesTable->patchEntity(
    $entity,
    $donneesDouteuses,
    ['validate' => 'maRegle']
);
```

Dans l'exemple ci-dessus, nous allons utiliser un validateur "maRegle", qui est défini en utilisant la méthode `validationMaRegle()` :

```
public function validationMaRegle($validator)
{
    $validator->add(
        // ...
    );

    return $validator;
}
```

La validation présuppose que les arguments passés sont des chaînes de caractères ou des tableaux, puisque c'est ce qui est passé dans n'importe quelle requête :

```
// Dans src/Model/Table/UsersTable.php
public function validatePasswords($validator)
{
    $validator->add('confirm_password', 'no-misspelling', [
        'rule' => ['compareWith', 'password'],
        'message' => 'Les mots de passe ne sont pas identiques',
    ]);

    // ...

    return $validator;
}
```

La validation **n'est pas** déclenchée lorsque vous définissez directement une propriété sur vos entities :

```
$userEntity->email = 'pas un email!!';
$usersTable->save($userEntity);
```

Dans l'exemple ci-dessus, l'entity sera sauvegardée car la validation n'est déclenchée que par les méthodes `newEntity()` et `patchEntity()`. Le second niveau de validation est conçu pour gérer cette situation.

Les règles d'application, comme expliqué précédement, seront vérifiées à chaque appel de `save()` ou `delete()` :

```
// Dans src/Model/Table/UsersTable.php
public function buildRules(RulesChecker $rules): RulesChecker
```

(suite sur la page suivante)

(suite de la page précédente)

```
{
    $rules->add($rules->isUnique(['email']));

    return $rules;
}

// Autre part dans le code de votre application
$userEntity->email = 'a@email.en.doublon';
$usersTable->save($userEntity); // Retourne false
```

La validation est conçue pour les données provenant directement des utilisateurs, tandis que les règles d'application sont conçues spécifiquement pour les transitions de données générées à l'intérieur de l'application :

```
// Dans src/Model/Table/CommandesTable.php
public function buildRules(RulesChecker $rules): RulesChecker
{
    $check = function($commande) {
        if ($commande->livraison !== 'gratuit') {
            return true;
        }

        return $commande->prix >= 100;
    };
    $rules->add($check, [
        'errorField' => 'livraison',
        'message' => 'Pas de frais de port gratuits pour une commande de moins de 100!'
    ]);

    return $rules;
}

// Autre part dans le code de l'application
$commande->prix = 50;
$commande->livraison = 'gratuit';
$commandesTable->save($commande); // Retourne false
```

Utiliser la Validation en tant que Règle d'Application

Dans certaines situations, vous voudrez peut-être lancer les mêmes routines à la fois pour des données générées par un utilisateur et pour l'intérieur de votre application. Cela peut se produire lorsque vous exécutez un script CLI qui définit des propriétés directement dans des entities :

```
// Dans src/Model/Table/UsersTable.php
public function validationDefault(Validator $validator): Validator
{
    $validator->add('email', 'email_valide', [
        'rule' => 'email',
        'message' => 'Email invalide'
    ]);

    // ...
```

(suite sur la page suivante)

(suite de la page précédente)

```
    return $validator;
}

public function buildRules(RulesChecker $rules): RulesChecker
{
    // Ajoute des règles de validation
    $rules->add(function($entity) {
        $data = $entity->extract($this->getSchema()->columns(), true);
        if (!$entity->isNew() && !empty($data)) {
            $data += $entity->extract((array)$this->getPrimaryKey());
        }
        $validator = $this->getValidator('default');
        $errors = $validator->validate($data, $entity->isNew());
        $entity->setErrors($errors);

        return empty($errors);
    });

    // ...

    return $rules;
}
```

Lors de l'exécution du code suivant, la sauvegarde échouera grâce à la nouvelle règle d'application qui a été ajoutée :

```
$userEntity->email = 'Pas un email!!!';
$usersTable->save($userEntity);
$userEntity->getErrors('email'); // Email invalide
```

Le même résultat est attendu lors de l'utilisation de `newEntity()` ou `patchEntity()` :

```
$userEntity = $usersTable->newEntity(['email' => 'Pas un email!!']);
$userEntity->getErrors('email'); // Email invalide
```

Sauvegarder les Données

class `Cake\ORM\Table`

Après avoir *chargé vos données* vous voudrez probablement mettre à jour et sauvegarder les changements.

Coup d'Oeil sur l'Enregistrement des Données

Les applications ont habituellement deux façons d'enregistrer les données. La première est évidemment d'utiliser des formulaires web et l'autre consiste à générer ou modifier directement les données dans le code pour l'envoyer à la base de données.

Insérer des Données

Le moyen le plus simple d'insérer des données dans une base de données est de créer une nouvelle entity et de la passer à la méthode `save()` de la classe `Table` :

```
use Cake\ORM\Locator\LocatorAwareTrait;
```

(suite sur la page suivante)

(suite de la page précédente)

```
$articlesTable = $this->getTableLocator()->get('Articles');
$article = $articlesTable->newEmptyEntity();

$article->title = 'Un nouvel Article';
$article->body = 'Ceci est le contenu de cet article';

if ($articlesTable->save($article)) {
    // L'entity $article contient maintenant l'id
    $id = $article->id;
}
```

Mettre à jour des Données

La méthode `save()` sert également à la mise à jour des données :

```
use Cake\ORM\Locator\LocatorAwareTrait;

$articlesTable = $this->getTableLocator()->get('Articles');
$article = $articlesTable->get(12); // Retourne l'article avec l'id 12

$article->title = 'Un nouveau titre pour cet article';
$articlesTable->save($article);
```

CakePHP saura s'il doit faire une insertion ou une mise à jour d'après le résultat de la méthode `isNew()`. Les entities qui sont récupérées via `get()` ou `find()` renverront toujours `false` lorsqu'on appelle leur méthode `isNew()`.

Enregistrer avec des Associations

Par défaut, la méthode `save()` sauvegardera aussi un seul niveau d'association :

```
$articlesTable = $this->getTableLocator()->get('Articles');
$author = $articlesTable->Authors->findByUserName('mark')->first();

$article = $articlesTable->newEmptyEntity();
$article->title = 'Un article par mark';
$article->author = $author;

if ($articlesTable->save($article)) {
    // La valeur de la clé étrangère a été ajoutée automatiquement.
    echo $article->author_id;
}
```

La méthode `save()` est également capable de créer de nouveaux enregistrements pour les associations :

```
$firstComment = $articlesTable->Comments->newEmptyEntity();
$firstComment->body = 'Un super article';

$secondComment = $articlesTable->Comments->newEmptyEntity();
$secondComment->body = 'J aime lire ceci!';

$tag1 = $articlesTable->Tags->findByName('cakephp')->first();
$tag2 = $articlesTable->Tags->newEmptyEntity();
```

(suite sur la page suivante)

(suite de la page précédente)

```
$tag2->name = 'Génial';

$article = $articlesTable->get(12);
$article->comments = [$firstComment, $secondComment];
$article->tags = [$tag1, $tag2];

$articlesTable->save($article);
```

Associer des Enregistrements Many to Many

Dans le code ci-dessus il y a déjà un exemple de liaison d'un article vers deux tags. Il y a un autre moyen de faire la même chose en utilisant la méthode `link()` dans l'association :

```
$tag1 = $articlesTable->Tags->findByName('cakephp')->first();
$tag2 = $articlesTable->Tags->newEmptyEntity();
$tag2->name = 'Génial';

$articlesTable->Tags->link($article, [$tag1, $tag2]);
```

Dissocier des Enregistrements Many To Many

Pour dissocier des enregistrements many-to-many, utilisez la méthode `unlink()` :

```
$tags = $articlesTable
    ->Tags
    ->find()
    ->where(['name IN' => ['cakephp', 'awesome']])
    ->toList();

$articlesTable->Tags->unlink($article, $tags);
```

Lorsque vous modifiez des enregistrements en définissant ou modifiant directement leurs propriétés, il n'y aura pas de validation, ce qui est problématique s'il s'agit d'accepter les données d'un formulaire. La section suivante explique comment convertir efficacement les données de formulaire en entities afin qu'elles puissent être validées et sauvegardées.

Convertir les Données Requêtées en Entities

Avant de modifier et de réenregistrer les données dans la base de données, vous devrez convertir les données d'un format de tableau (figurant dans la requête) en entities utilisées par l'ORM. La classe Table peut convertir efficacement une ou plusieurs entities à partir des données de la requête. Vous pouvez convertir une entity unique en utilisant :

```
// Dans un controller.

$articles = $this->getTableLocator()->get('Articles');

// Valide et convertit en un objet Entity
$entity = $articles->newEntity($this->request->getData());
```

Note

Si vous utilisez newEntity() et qu'il manque tout ou partie des nouvelles données dans les entities créées, vérifiez que les colonnes que vous voulez modifier sont listées dans la propriété `$_accessible` de votre entity. Cf. *Assignement de Masse*.

Les données de la requête doivent suivre la structure de vos entities. Par exemple si vous avez un article, que celui-ci appartient à un utilisateur, (*belongs to*), et a plusieurs commentaires (*has many*), les données de votre requête devraient ressembler à :

```
$data = [
    'title' => 'Mon titre',
    'body' => 'Le texte',
    'user_id' => 1,
    'user' => [
        'username' => 'mark'
    ],
    'comments' => [
        ['body' => 'Premier commentaire'],
        ['body' => 'Deuxième commentaire'],
    ]
];
```

Par défaut, la méthode `newEntity()` valide les données qui lui sont passées, comme expliqué dans la section *Valider les Données Avant de Construire les Entities*. Si vous voulez empêcher la validation des données, passez l'option `'validate' => false` :

```
$entity = $articles->newEntity($data, ['validate' => false]);
```

Lors de la construction de formulaires qui sauvegardent des associations imbriquées, vous devez définir quelles associations doivent être converties :

```
// Dans un controller

$articles = $this->getTableLocator()->get('Articles');

// Nouvelle entity avec des associations imbriquées
$entity = $articles->newEntity($this->request->getData(), [
    'associated' => [
        'Tags', 'Comments' => ['associated' => ['Users']]
    ]
]);
```

Ceci indique que les Tags, les commentaires et les utilisateurs associés aux commentaires doivent être convertis. Au choix, vous pouvez aussi utiliser la notation par points, qui est plus concise :

```
// Dans un controller

$articles = $this->getTableLocator()->get('Articles');

// Nouvelle entity avec des associations imbriquées en utilisant
// la notation par points
$entity = $articles->newEntity($this->request->getData(), [
    'associated' => ['Tags', 'Comments.Users']
]);
```

Vous pouvez aussi désactiver la conversion d'associations imbriquées comme ceci :

```
$entity = $articles->newEntity($data, ['associated' => []]);
// ou...
$entity = $articles->patchEntity($entity, $data, ['associated' => []]);
```

Les données associées sont aussi validées par défaut, à moins de spécifier le contraire. Vous pouvez également changer l'ensemble de validation utilisé pour chaque association :

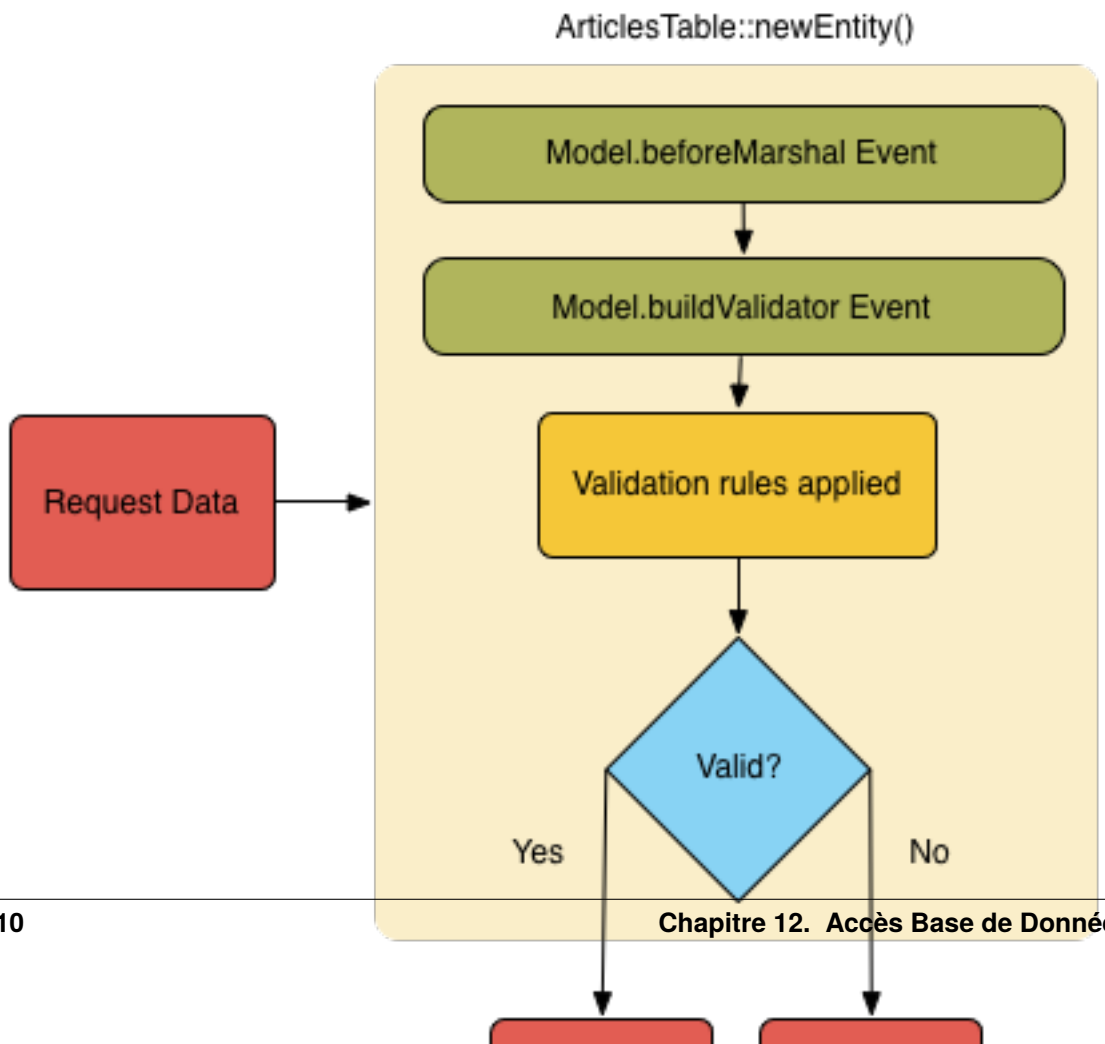
```
// Dans un controller

$articles = $this->getTableLocator()->get('Articles');

// Ne fait pas de validation pour l'association Tags et
// appelle l'ensemble de validation 'signup' pour Comments.Users
$entity = $articles->newEntity($this->request->getData(), [
    'associated' => [
        'Tags' => ['validate' => false],
        'Comments.Users' => ['validate' => 'signup']
    ]
]);
```

Le chapitre *Utiliser un Ensemble de Validation Différent pour les Associations* contient davantage d'informations sur la façon d'utiliser les différents validateurs pour la sauvegarde d'associations.

Le diagramme suivant donne un aperçu de ce qui se passe dans la méthode `newEntity()` ou `patchEntity()` :



La méthode `newEmptyEntity()` vous renverra toujours une entity. Si la validation échoue, votre entité contiendra des erreurs et

les champs invalides ne seront pas remplis dans l'entity créée.

Convertir des Données BelongsToMany

Si vous sauvegardez des associations belongsToMany, vous pouvez utiliser soit une liste de données d'entity, soit une liste d'ids. Quand vous utilisez une liste de données d'entity, les données de votre requête devraient ressembler à ceci :

```
$data = [
    'title' => 'Mon titre',
    'body' => 'Le texte',
    'user_id' => 1,
    'tags' => [
        ['name' => 'CakePHP'],
        ['name' => 'Internet'],
    ],
];
```

Ce code créera 2 nouveaux tags. Si vous voulez créer un lien entre un article et des tags existants, vous pouvez utiliser une liste d'ids. Les données de votre requête doivent ressembler à ceci :

```
$data = [
    'title' => 'Mon titre',
    'body' => 'Le texte',
    'user_id' => 1,
    'tags' => [
        '_ids' => [1, 2, 3, 4],
    ],
];
```

Si vous souhaitez lier des entrées belongsToMany existantes et en créer de nouvelles en même temps, vous pouvez utiliser la forme étendue :

```
$data = [
    'title' => 'Mon titre',
    'body' => 'Le texte',
    'user_id' => 1,
    'tags' => [
        ['name' => 'Un nouveau tag'],
        ['name' => 'Un autre nouveau tag'],
        ['id' => 5],
        ['id' => 21],
    ],
```

(suite sur la page suivante)

(suite de la page précédente)

```
];
```

Quand ces données seront converties en entities, il y aura 4 tags. Les deux premiers seront de nouveaux objets, et les deux autres seront des références à des tags existants.

Quand vous convertissez des données de belongsToMany, vous pouvez désactiver la création d'une nouvelle entity en utilisant l'option `onlyIds` :

```
$result = $articles->patchEntity($entity, $data, [
    'associated' => ['Tags' => ['onlyIds' => true]],
]);
```

Quand elle est activée, cette option restreint la conversion des données de belongsToMany pour utiliser uniquement la clé `_ids`.

Convertir des Données HasMany

Si vous souhaitez mettre à jour des associations hasMany existantes et mettre à jour leurs propriétés, vous devez d'abord vous assurer que votre entity est chargée avec les données hasMany associées. Vous pouvez ensuite utiliser une requête avec des données structurées de la façon suivante :

```
$data = [
    'title' => 'Mon titre',
    'body' => 'Le texte',
    'comments' => [
        ['id' => 1, 'comment' => 'Mettre à jour le premier commentaire'],
        ['id' => 2, 'comment' => 'Mettre à jour le deuxième commentaire'],
        ['comment' => 'Créer un nouveau commentaire'],
    ],
];
```

Si vous sauvegardez des associations hasMany et que voulez lier des enregistrements existants à un nouveau parent, vous pouvez utiliser le format `_ids` :

```
$data = [
    'title' => 'Mon nouvel article',
    'body' => 'Le texte',
    'user_id' => 1,
    'comments' => [
        '_ids' => [1, 2, 3, 4],
    ],
];
```

Quand vous convertissez des données de hasMany, vous pouvez désactiver la création d'une nouvelle entity en utilisant l'option `onlyIds`. Quand elle est activée, cette option restreint la conversion des données hasMany pour utiliser uniquement la clé `_ids` et ignorer toutes les autres données.

Convertir des Enregistrements Multiples

Lorsque vous créez des formulaires de création/mise à jour de plusiseurs enregistrements en une seule opération, vous pouvez utiliser `newEntities()` :

```
// Dans un controller.

$articles = $this->getTableLocator()->get('Articles');
$entities = $articles->newEntities($this->request->getData());
```

Dans cette situation, les données de la requête pour plusieurs articles doivent ressembler à ceci :

```
$data = [
    [
        'title' => 'Premier post',
        'published' => 1,
    ],
    [
        'title' => 'Second post',
        'published' => 1,
    ],
];
```

Une fois que vous avez converti les données de la requête en entities, vous pouvez sauvegarder :

```
// Dans un controller.
foreach ($entities as $entity) {
    // Sauvegarder l'entity
    $articles->save($entity);
}
```

Ce code va lancer séparément une transaction pour chaque entity sauvegardée. Si vous voulez traiter toutes les entities en transaction unique, vous pouvez utiliser `saveMany()` ou `saveManyOrFail()` :

```
// Renvoie un booléen pour indiquer si l'opération a réussi
$articles->saveMany($entities);

// Lève une PersistenceFailedException si l'un des enregistrements échoue
$articles->saveManyOrFail($entities);
```

Changer les Champs Accessibles

Il est également possible d'autoriser `newEntity()` à écrire dans des champs non accessibles. Par exemple, `id` est généralement absent de la propriété `_accessible`. Dans un tel cas, vous pouvez utiliser l'option `accessibleFields`. Il pourrait être utile de conserver les ids des entities associées :

```
// Dans un controller.

$articles = $this->getTableLocator()->get('Articles');
$entity = $articles->newEntity($this->request->getData(), [
    'associated' => [
        'Tags', 'Comments' => [
            'associated' => [
                'Users' => [
                    'accessibleFields' => ['id' => true],
                ],
            ],
        ],
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ],
]);
```

Le code ci-dessus permet de conserver l'association entre Comments et Users pour l'entity concernée.

> **Note**
>
> Si vous utilisez newEntity() et qu'il manque dans l'entity tout ou partie des données transmises, vérifiez à deux fois que les colonnes que vous souhaitez définir sont listées dans la propriété `$_accessible` de votre entity. Cf. *Assignement de Masse*.

Fusionner les Données de la Requête dans les Entities

Pour mettre à jour des entities, vous pouvez choisir d'appliquer les données de la requête directement sur une entity existante. Cela a l'avantage que seuls les champs réellement modifiés seront sauvegardés, au lieu d'envoyer tous les champs à la base de données, même ceux qui sont inchangés. Vous pouvez fusionner un tableau de données brutes dans une entity existante en utilisant la méthode `patchEntity()` :

```
// Dans un controller.

$articles = $this->getTableLocator()->get('Articles');
$article = $articles->get(1);
$articles->patchEntity($article, $this->request->getData());
$articles->save($article);
```

Validation et patchEntity

De la même façon que `newEntity()`, la méthode `patchEntity` validera les données avant de les copier dans l'entity. Ce mécanisme est expliqué dans la section *Valider les Données Avant de Construire les Entities*. Si vous souhaitez désactiver la validation lors du patch d'une entity, passez l'option `validate` comme ceci :

```
// Dans un controller.

$articles = $this->getTableLocator()->get('Articles');
$article = $articles->get(1);
$articles->patchEntity($article, $data, ['validate' => false]);
```

Vous pouvez également changer l'ensemble de validation utilisé pour l'entity ou pour n'importe qu'elle association :

```
$articles->patchEntity($article, $this->request->getData(), [
    'validate' => 'custom',
    'associated' => ['Tags', 'Comments.Users' => ['validate' => 'signup']]
]);
```

Patcher des HasMany et BelongsToMany

Comme expliqué dans la section précédente, les données de la requête doivent suivre la structure de votre entity. La méthode `patchEntity()` est également capable de fusionner les associations. Par défaut seul les premiers niveaux d'associations sont fusionnés mais si vous voulez contrôler la liste des associations à fusionner, ou fusionner des niveaux de plus en plus profonds, vous pouvez utiliser le troisième paramètre de la méthode :

```
// Dans un controller.
$associated = ['Tags', 'Comments.Users'];
$article = $articles->get(1, ['contain' => $associated]);
$articles->patchEntity($article, $this->request->getData(), [
    'associated' => $associated,
]);
$articles->save($article);
```

Les associations sont fusionnées en faisant correspondre la clé primaire des entities source avec les champs correspondants dans le tableau de données fourni. Si aucune entity cible n'est trouvée, les associations construiront de nouvelles entities.

Pa exemple, prenons une requête contenant les données suivantes :

```
$data = [
    'title' => 'Mon titre',
    'user' => [
        'username' => 'mark',
    ],
];
```

Si vous essayez de patcher une entity ne contenant pas d'entity associée dans la propriété user, une nouvelle entity sera créée pour *user* :

```
// Dans un controller.
$entity = $articles->patchEntity(new Article, $data);
echo $entity->user->username; // Affiche 'mark'
```

Cela fonctionne de la même manière pour les associations hasMany et belongsToMany, avec cependant un point d'attention important :

Note

Pour les associations belongsToMany, vérifiez que les entities associées ont bien une propriété accessible pour l'entité associée.

Si Product belongsToMany Tag :

```
// Dans l'entity Product
protected array $_accessible = [
    // .. autres propriétés
   'tags' => true,
];
```

Note

Pour les associations hasMany et belongsToMany, s'il y avait des entities dont la clé primaire ne correspondait à aucun enregistrement dans le tableau de données, alors ces enregistrements seraient écartés de l'entity résultante.

Rappelez-vous que l'utilisation de `patchEntity()` ou de `patchEntities()` ne fait pas persister les données, elle ne fait que modifier (ou créer) les entities données. Pour sauvegarder l'entity, vous devrez appeler la méthode `save()` de la table.

Par exemple, considérons le cas suivant :

```
$data = [
    'title' => 'Mon titre',
    'body' => 'Le text',
    'comments' => [
        ['body' => 'Premier commentaire', 'id' => 1],
        ['body' => 'Second commentaire', 'id' => 2],
    ],
];
$entity = $articles->newEntity($data);
$articles->save($entity);

$newData = [
    'comments' => [
        ['body' => 'Commentaire modifié', 'id' => 1],
        ['body' => 'Un nouveau commentaire'],
    ],
];
$articles->patchEntity($entity, $newData);
$articles->save($entity);
```

Au final, si l'entity est à nouveau convertie en tableau, vous obtiendrez le résultat suivant :

```
[
    'title' => 'Mon titre',
    'body' => 'Le text',
    'comments' => [
        ['body' => 'Commentaire modifié', 'id' => 1],
        ['body' => 'Un nouveau commentaire'],
    ]
];
```

Comme vous pouvez le constater, le commentaire avec l'id 2 a disparu, puisqu'il ne correspondait à aucun élément du tableau `$newData`. Cela se passe ainsi parce CakePHP calque les données de l'entity sur le nouvel état que décrit par les données de la requête.

Un autre avantage à cette approche est qu'elle réduit le nombre d'opérations à exécuter lorsque l'entity est à nouveau persistée.

Notez bien que cela ne veut pas dire que le commentaire avec l'id 2 a été supprimé de la base de données. Si vous souhaitez supprimer les commentaires de cet article qui ne sont pas présents dans l'entity, vous pouvez récupérer les clés primaires et exécuter une suppression batch pour celles qui ne sont pas dans la liste :

```
// Dans un controller.
use Cake\Collection\Collection;

$comments = $this->getTableLocator()->get('Comments');
$present = (new Collection($entity->comments))->extract('id')->filter()->toList();
$comments->deleteAll([
    'article_id' => $article->id,
    'id NOT IN' => $present,
]);
```

Comme vous voyez, cela permet aussi de créer des solutions dans lesquelles une association a besoin d'être implémentée comme un ensemble unique.

Vous pouvez aussi faire un patch de plusieurs entities à la fois. Ce que nous avons vu pour les associations hasMany et belongsToMany s'applique aussi pour patcher plusieurs entities : les correspondances se font d'après la valeur de la clé primaire et celles qui sont absentes dans le tableau des entities d'origine seront retirées et absentes des résultats :

```
// Dans un controller.

$articles = $this->getTableLocator()->get('Articles');
$list = $articles->find('popular')->toList();
$patched = $articles->patchEntities($list, $this->request->getData());
foreach ($patched as $entity) {
    $articles->save($entity);
}
```

De la même façon qu'avec `patchEntity()`, vous pouvez utiliser le troisième argument pour contrôler les associations qui seront fusionnées dans chacune des entities du tableau :

```
// Dans un controller.
$patched = $articles->patchEntities(
    $list,
    $this->request->getData(),
    ['associated' => ['Tags', 'Comments.Users']]
);
```

Modifier les Données de la Requête Avant de Construire les Entities

Si vous devez modifier les données de la requête avant de les convertir en entities, vous pouvez utiliser l'event `Model.beforeMarshal`. Cet event vous permet de manipuler les données de la requête juste avant la création des entities :

```
// Ajoutez les instructions use au début de votre fichier.
use Cake\Event\EventInterface;
use ArrayObject;

// Dans une classe de table ou un behavior
public function beforeMarshal(EventInterface $event, ArrayObject $data, ArrayObject
↪$options)
{
    if (isset($data['username'])) {
        $data['username'] = mb_strtolower($data['username']);
    }
}
```

Le paramètre `$data` est une instance `ArrayObject`, donc vous n'avez pas besoin de la renvoyer pour changer les données qui seront utilisées pour créer les entities.

Le but principal de `beforeMarshal` est d'aider les utilisateurs à passer le process de validation lorsque des erreurs simples peuvent être résolues automatiquement, ou lorsque les données doivent être restructurées pour être placées dans les bons champs.

L'event `Model.beforeMarshal` est lancé juste au début du process de validation. Une des raisons à cela est que `beforeMarshal` est autorisé à modifier les règles de validation et les options d'enregistrement, telles que la liste blanche des champs. La validation est lancée juste après la fin de l'exécution de cet événement. Un exemple classique de modification des données avant leur validation est la suppression des espaces superflus dans tous les champs avant leur enregistrement :

```
// Ajoutez les instructions use au début de votre fichier.
use Cake\Event\EventInterface;
use ArrayObject;

// Dans une table ou un behavior
public function beforeMarshal(EventInterface $event, ArrayObject $data, ArrayObject
↪$options)
{
    foreach ($data as $key => $value) {
        if (is_string($value)) {
            $data[$key] = trim($value);
        }
    }
}
```

Du fait du mode de fonctionnement du marshalling, si un champ ne passe pas la validation il sera automatiquement supprimé du tableau de données et ne sera pas copié dans l'entity. Cela évite d'avoir des données incohérentes dans l'objet entity.

De plus, les données fournies à la méthode `beforeMarshal` sont une copie des données passées. La raison à cela est qu'il est important de préserver les données saisies à l'origine par l'utilisateur, car elles sont susceptibles de servir autre part.

Modifier les Entités Après leur Mise À Jour À Partir de la Requête

L'événement `Model.afterMarshal` vous permet de modifier les entités après qu'elles auront été créées ou modifiées à partir des données de la requête. Cela peut vous servir à appliquer une logique de validation supplémentaire qui ne peut pas être exprimée de manière simple à travers les méthodes du Validator :

```
// Ajoutez les instructions use au début de votre fichier.
use Cake\Event\EventInterface;
use Cake\ORM\EntityInterface;
use ArrayObject;

// Dans une classe de table ou de behavior
public function afterMarshal(
    EventInterface $event,
    EntityInterface $entity,
    ArrayObject $data,
    ArrayObject $options
) {
    // Ne pas accepter les personnes dont le nom commence par J le 20 de
    // chaque mois.
    if (mb_substr($entity->name, 1) === 'J' && (int)date('d') === 20) {
        $entity->setError('name', 'Pas de noms en J aujourd\'hui. Désolé.');
    }
}
```

Ajouté dans la version 4.1.0.

Valider les Données Avant de Construire les Entities

Le chapitre *Valider des Données* vous fournira plus d'informations sur les fonctionnalités de validation de CakePHP pour garantir l'exactitude et la cohérence de vos données.

Éviter les Attaques d'Assignement en Masse de Propriétés

Lors de la création ou la fusion des entities à partir de données de la requête, vous devez être attentif aux ajouts ou modifications que vous permettez à vos utilisateurs dans les entities. Par exemple, en envoyant dans la requpete un tableau contenant `user_id`, un pirate pourrait changer le propriétaire d'un article, ce qui entraînerait des effets indésirables :

```
// Contient ['user_id' => 100, 'title' => 'Piraté !'];
$data = $this->request->getData();
$entity = $this->patchEntity($entity, $data);
$this->save($entity);
```

Il y a deux façons de se protéger contre ce problème. La première est de définir les colonnes par défaut qui peuvent être définies en toute sécurité à partir d'une requête en utilisant la fonctionnalité d'*Assignement de Masse* dans les entities.

La deuxième façon est d'utiliser l'option `fields` lors de la création ou la fusion de données dans une entity :

```
// Contient ['user_id' => 100, 'title' => 'Piraté !'];
$data = $this->request->getData();

// Permet seulement de changer le title
$entity = $this->patchEntity($entity, $data, [
    'fields' => ['title']
]);
$this->save($entity);
```

Vous pouvez aussi contrôler les propriétés qui peuvent être assignées pour les associations :

```
// Permet seulement de modifier le titre et les tags
// et le nom du tag est la seule colonne qui puisse être définie
$entity = $this->patchEntity($entity, $data, [
    'fields' => ['title', 'tags'],
    'associated' => ['Tags' => ['fields' => ['name']]]
]);
$this->save($entity);
```

Cette fonctionnalité est pratique quand vos utilisateurs ont accès plusieurs fonctions et que vous voulez leur permettre de modifier différentes données en fonction de leurs privilèges.

Sauvegarder les Entities

`Cake\ORM\Table::save`(*Entity $entity*, *array $options = []*)

Quand vous sauvegardez les données de la requête dans votre base de données, vous devez d'abord hydrater une nouvelle entity en utilisant `newEntity()`, que vous pourrez ensuite passer à `save()`. Par exemple :

```
// Dans un controller

$articles = $this->getTableLocator()->get('Articles');
$article = $articles->newEntity($this->request->getData());
```

(suite sur la page suivante)

(suite de la page précédente)

```
if ($articles->save($article)) {
    // ...
}
```

L'ORM utilise la méthode `isNew()` sur une entity pour déterminer s'il faut réaliser une insertion ou une mise à jour. Si la méthode `isNew()` renvoie `true` et que l'entity a une clé primaire, l'ORM va d'abord lancer une requête "exists". Cette requête "exists" peut être supprimée en passant `'checkExisting' => false` dans l'argument `$options` :

```
$articles->save($article, ['checkExisting' => false]);
```

Une fois que vous aurez chargé quelques entities, vous voudrez probablement les modifier et mettre à jour la base de données. C'est une manipulation simple dans CakePHP :

```
$articles = $this->getTableLocator()->get('Articles');
$article = $articles->find('all')->where(['id' => 2])->first();

$article->title = 'Mon nouveau titre';
$articles->save($article);
```

Lors de la sauvegarde, CakePHP va *appliquer vos règles de validation*, et inclure l'opération de sauvegarde dans une transaction de la base de données. Cela mettra à jour uniquement les propriétés qui ont changé. L'appel à `save()` ci-dessus va générer un code SQL de ce type :

```
UPDATE articles SET title = 'Mon nouveau titre' WHERE id = 2;
```

Si vous aviez une nouvelle entity, cela générerait le code SQL suivant :

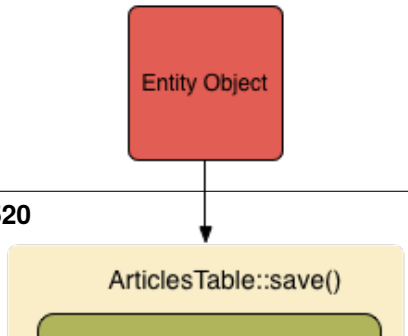
```
INSERT INTO articles (title) VALUES ('Mon nouveau titre');
```

Quand une entity est sauvegardée, voici ce qui se passe :

1. La vérification des règles commencera si elle n'est pas désactivée.
2. La vérification des règles va déclencher l'event `Model.beforeRules`. Si l'event est stoppé, l'opération de sauvegarde échouera et retournera `false`.
3. Les règles seront vérifiées. Si l'entity est en train d'être créée, les règles `create` seront utilisées. Si l'entity est en train d'être mise à jour, les règles `update` seront utilisées.
4. L'event `Model.afterRules` sera déclenché.
5. L'event `Model.beforeSave` est dispatché. S'il est stoppé, la sauvegarde sera annulée, et save() va retourner `false`.
6. Les associations parentes sont sauvegardées. Par exemple, toute association belongsTo listée sera sauvegardée.
7. Les champs modifiés sur l'entity seront sauvegardés.
8. Les associations enfants sont sauvegardées. Par exemple, toute association hasMany, hasOne, ou belongsToMany listée sera sauvegardée.
9. L'event `Model.afterSave` sera dispatché.
10. L'event `Model.afterSaveCommit` sera dispatché.

Le diagramme suivant illustre ce procédé :

Consultez la section *Appliquer des Règles d'Application* pour plus d'informations sur la création et l'utilisation des règles.



Avertissement

Si l'entity n'a subi aucun c
ment de sa sauvegarde, les
pas être déclenchés car au
sauvegarde n'est effectuée.

La méthode `save()` va retourner l'entity modifiée en cas de succès, et `false` en cas d'échec. Vous pouvez désactiver les règles et/ou les transactions en utilisant l'argument `$options` lors de la sauvegarde :

```
// Dans un controller␣
↪ou une méthode de table.
$articles-
↪>save($article, ['checkRules
↪' => false, 'atomic' => false]);
```

Sauvegarder les Associations

Quand vous sauvegardez une entity, vous pouvez aussi choisir de sauvegarder tout ou partie des entities associées. Par défaut, toutes les entities de premier niveau seront sauvegardées. Par exemple, sauvegarder un Article va aussi mettre à jour automatiquement toute entity modifiée directement liée à la table des articles.

Vous pouvez accéder à un réglage plus fin des associations qui sont sauvegardées en utilisant l'option `associated` :

```
// Dans un controller.

// Sauvegarde␣
↪seulement l'association␣
↪avec les commentaires
$articles->save($entity,␣
↪['associated' => ['Comments']]);
```

Vous pouvez définir une sauvegarde d'associations imbriquées sur plusieurs niveaux en utilisant la notation par point :

```
// Sauvegarde␣
↪la société, les employés et les␣
↪adresses liées à chacun d'eux.
$companies-
↪>save($entity, ['associated
↪' => ['Employees.Addresses']]);
```

De plus, vous pouvez combiner la notation par point pour les associations avec le tableau d'options :

```
$companies->save($entity, [
  'associated' => [
    'Employees',
    'Employees.Addresses'
  ]
]);
```

Vos entities doivent être structurées de la même façon qu'elles l'étaient quand elles ont été chargées à partir de la base de données. Consultez la documentation du helper Form pour savoir comment *Créer des Inputs pour les Données Associées*.

Si vous construisez ou modifiez des données associées après avoir construit vos entities, vous devrez marquer la propriété d'association comme étant modifiée en utilisant `setDirty()` :

```
$company->author->name = 'Master Chef';
$company->setDirty('author', true);
```

Sauvegarder les Associations BelongsTo

Lors de la sauvegarde des associations belongsTo, l'ORM attend une entity imbriquée unique avec le nom de l'association au singulier et *des underscores*. Par exemple :

```
// Dans un controller.
$data = [
    'title' => 'Premier Post',
    'user' => [
        'id' => 1,
        'username' => 'mark'
    ]
];

$articles = $this->getTableLocator()->get('Articles');
$article = $articles->newEntity($data, [
    'associated' => ['Users']
]);

$articles->save($article);
```

Sauvegarder les Associations HasOne

Lors de la sauvegarde d'associations hasOne, l'ORM attend une entity imbriquée unique avec le nom de l'association au singulier et *des underscores underscore*. Par exemple :

```
// Dans un controller.
$data = [
    'id' => 1,
    'username' => 'cakephp',
    'profile' => [
        'twitter' => '@cakephp'
    ]
```

(suite sur la page suivante)

(suite de la page précédente)

```
];

$users = $this->getTableLocator()->get('Users');
$user = $users->newEntity($data, [
    'associated' => ['Profiles']
]);
$users->save($user);
```

Sauvegarder les Associations HasMany

Lors de la sauvegarde d'associations hasMany, l'ORM attend un tableau d'entities avec le nom de l'association au pluriel et *des underscores*. Par exemple :

```
// Dans un controller.
$data = [
    'title' => 'Premier Post',
    'comments' => [
        ['body' => 'Le meilleur post de ma vie'],
        ['body' => 'Celui-là, je l\'aime vraiment bien.']
    ]
];

$articles = $this->getTableLocator()->get('Articles');
$article = $articles->newEntity($data, [
    'associated' => ['Comments']
]);
$articles->save($article);
```

Lors de la sauvegarde d'associations hasMany, les enregistrements associés seront soit mis à jour, soit insérés. Dans les cas où l'enregistrement a déjà des enregistrements associés dans la base de données, vous avez le choix entre deux stratégies de sauvegarde :

append
: Les enregistrements associés sont mis à jour dans la base de données ou, s'ils ne correspondent à aucun enregistrement existant, sont insérés.

replace
: Tout enregistrement existant qui ne correspond pas aux enregistrements fournis sera supprimé de la base de données. Seuls les enregistrements fournis resteront (ou seront insérés).

Par défaut, l'ORM utilise la stratégie de sauvegarde `append`. Consultez *Associations HasMany* pour plus de détails sur la définition de `saveStrategy`.

Quel que soit le moment où vous ajoutez de nouveaux enregistrements dans une association existante, vous devez toujours marquer la propriété de l'association comme "dirty". Cela fait savoir à l'ORM que la propriété de cette association doit être persistée :

```
$article->comments[] = $comment;
$article->setDirty('comments', true);
```

Sans l'appel à `setDirty()`, les commentaires mis à jour ne seront pas sauvegardés.

Si vous créez une entité et voulez y attacher des enregistrements existants dans une association hasMany ou belongsToMany, vous devez d'abord initialiser la propriété de l'association :

```
$article->comments = [];
```

Sans l'initialisation, l'appel à `$article->comments[] = $comment;` sera sans effet.

Sauvegarder les Associations BelongsToMany

Lors de la sauvegarde d'associations hasMany, l'ORM attend un tableau d'entities avec le nom de l'association au pluriel et *des underscores*. Par exemple :

```
// Dans un controller.
$data = [
    'title' => 'Premier Post',
    'tags' => [
        ['tag' => 'CakePHP'],
        ['tag' => 'Framework']
    ]
];

$articles = $this->getTableLocator()->get('Articles');
$article = $articles->newEntity($data, [
    'associated' => ['Tags']
]);
$articles->save($article);
```

Quand vous convertissez les données de la requête en entities, les méthodes `newEntity()` et `newEntities()` traiteront les deux tableaux de propriétés, ainsi qu'une liste d'ids sous la clé `_ids`. L'utilisation de la clé `_ids` permet de construire des contrôles de formulaire basés sur une liste déroulante ou une liste de cases à cocher pour les associations belongsToMany. Consultez la section *Convertir les Données Requêtées en Entities* pour plus d'informations.

Lors de la sauvegarde des associations belongsToMany, vous avez le choix entre deux stratégies de sauvegarde :

append
: Seuls les nouveaux liens seront créés de chaque côté de cette association. Cette stratégie détruira pas les liens existants même s'ils sont absents du tableau d'entities à sauvegarder.

replace
: Lors de la sauvegarde, les liens existants seront supprimés et les nouveaux liens seront créés dans la table de jointure. S'il y a des liens existants dans la base de données vers certaines des entities que l'on souhaite sauvegarder, ces liens seront mis à jour, et non supprimés et re-sauvegardés.

Consultez *Associations BelongsToMany* pour plus de détails sur la façon de définir `saveStrategy`.

Par défaut, l'ORM utilise la stratégie `replace`. Si vous ajoutez à quelque moment que ce soit de nouveaux enregistrements dans une association existante, vous devez toujours marquer la propriété de l'association comme "dirty". Cela fait savoir à l'ORM que la propriété de l'association doit être persistée :

```
$article->tags[] = $tag;
$article->setDirty('tags', true);
```

Sans appel à `setDirty()`, les tags modifiés ne seront pas sauvegardés.

Vous vous retrouverez souvent à vouloir créer une association entre deux entities existantes, par exemple un utilisateur co-auteur d'un article. Pour cela, utilisez la méthode `link()` :

```
$article = $this->Articles->get($articleId);
$user = $this->Users->get($userId);

$this->Articles->Users->link($article, [$user]);
```

Lors de la sauvegarde d'associations belongsToMany, il peut être pertinent de sauvegarder des données supplémentaires dans la table de jointure. Dans l'exemple précédent des tags, on pourrait imaginer le type de vote `vote_type` de la personne qui a voté sur cet article. Le `vote_type` peut être soit `upvote`, soit `downvote`, et il est représenté par une chaîne de caractères. La relation est entre Users et Articles.

La sauvegarde de cette association et du `vote_type` est réalisée en ajoutant tout d'abord des données à `_joinData` et en sauvegardant ensuite l'association avec `link()`, par exemple :

```
$article = $this->Articles->get($articleId);
$user = $this->Users->get($userId);

$user->_joinData = new Entity(['vote_type' => $voteType], ['markNew' => true]);
$this->Articles->Users->link($article, [$user]);
```

Sauvegarder des Données Supplémentaires dans la Table de Jointure

Dans certaines situations, vous aurez des colonnes supplémentaires dans la table de jointure de l'association BelongsToMany. Avec CakePHP, il est facile d'y sauvegarder des propriétés. Chaque entity d'une association belongsToMany a une propriété `_joinData` qui contient les colonnes supplémentaires de la table de jointure. Ces données peuvent être soit un tableau, soit une instance Entity. Par exemple, si les Students BelongsToMany Courses, nous pourrions avoir une table de jointure qui ressemble à ceci :

```
id | student_id | course_id | days_attended | grade
```

Lors de la sauvegarde de données, vous pouvez remplir les colonnes supplémentaires de la table de jointure en définissant les données dans la propriété `_joinData` :

```
$student->courses[0]->_joinData->grade = 80.12;
$student->courses[0]->_joinData->days_attended = 30;

$studentsTable->save($student);
```

La propriété `_joinData` peut être soit une entity, soit un tableau de données si vous sauvegardez des entities construites à partir de données de la requête. Lorsque vous sauvegardez des données de la table de jointure à partir de données de la requête, vos données POST doivent ressembler à ceci :

```
$data = [
    'first_name' => 'Sally',
    'last_name' => 'Parker',
    'courses' => [
        [
            'id' => 10,
            '_joinData' => [
                'grade' => 80.12,
                'days_attended' => 30
            ]
        ],
        // d'autres cours (courses).
    ]
];
$student = $this->Students->newEntity($data, [
    'associated' => ['Courses._joinData']
]);
```

Consultez le chapitre sur les *inputs pour les données associées* pour savoir comment construire correctement des inputs avec le `FormHelper`.

Sauvegarder les Types Complexes

Les tables peuvent stocker des données représentées dans des types basiques, comme des chaînes, integers, floats, booleans, etc… Mais elles peuvent aussi être étendues pour accepter des types plus complexes comme des tableaux ou des objets et sérialiser ces données en types plus simples qui peuvent être sauvegardés dans la base de données.

Pour cela, vous devez utiliser le système de types personnalisés. Consulter la section *Ajouter des Types Personnalisés* pour savoir comment construire des Types de colonnes personnalisés :

```
use Cake\Database\TypeFactory;

TypeFactory::map('json', 'Cake\Database\Type\JsonType');

// Dans src/Model/Table/UsersTable.php

class UsersTable extends Table
{

    public function initialize(): void
    {
        $this->getSchema()->setColumnType('preferences', 'json');
    }

}
```

Le code ci-dessus fait correspondre la colonne `preferences` au type personnalisé `json`. Cela signifie que lorsque vous récupérez des données de cette colonne, les chaînes JSON de la base de données seront désérialisées et insérées dans une entity sous forme de tableau.

De même, lors de la sauvegarde, le tableau sera à nouveau transformé au format JSON :

```
$user = new User([
    'preferences' => [
        'sports' => ['football', 'baseball'],
        'books' => ['Maîtriser le PHP', 'Hamlet']
    ]
]);
$usersTable->save($user);
```

Quand vous utilisez des types complexes, il est important de vérifier que les données que vous recevez de l'utilisateur final correspondent au bon type. Sinon, en ne traitant pas correctement les données complexes, vous vous exposez à ce que des utilisateurs malveillants puissent stocker des données qu'ils n'auraient normalement pas le droit de stocker.

Strict Saving

`Cake\ORM\Table::saveOrFail`(*EntityInterface $entity*, *array $options = []*)

L'appel à cette méthode lancera une `Cake\ORM\Exception\PersistenceFailedException` si :
- les règles de validation ont échoué
- l'entity contient des erreurs
- la sauvegarde a été annulée par un _callback_.

Cette méthode peut être utile pour effectuer des opérations complexes sur la base de données sans surveillance humaine, comme lors de l'utilisation de script via des _tasks_ Shell.

> **Note**
>
> Si vous utilisez cette méthode dans un Controller, assurez-vous de capturer la `PersistenceFailedException` qui pourrait être levée.

Si vous voulez trouver l'entity qui n'a pas pu être sauvegardée, vous pouvez utiliser la méthode `Cake\ORMException\PersistenceFailedException::getEntity()` :

```
try {
    $table->saveOrFail($entity);
} catch (\Cake\ORM\Exception\PersistenceFailedException $e) {
    echo $e->getEntity();
}
```

Dans la mesure où cette méthode utilise la méthode *`Cake\ORM\Table::save()`*, tous les événements de `save` seront déclenchés.

Trouver Ou Créer une Entity

`Cake\ORM\Table::`**`findOrCreate`**(*$search*, *$callback = null*, *$options = []*)

Trouve un enregistrement d'après les critères de `$search` ou crée un nouvel enregistrement en utilisant les propriétés de `$search` et en appelant la méthode optionnelle `$callback`. Cette méthode est idéale dans les scénarios où vous avez besoin réduire les risque de doublons :

```
$record = $table->findOrCreate(
    ['email' => 'bobbi@example.com'],
    function ($entity) use ($autresDonnees) {
        // Appelé seulement en cas de création d'un nouvel enregistrement
        $entity->name = $autresDonnees['name'];
    }
);
```

Si vos conditions pour `find` nécessitent un tri, des associations ou des conditions personnalisés, alors le paramètre `$search` peut être un _callable_ ou un objet `Query`. Si vous utilisez un _callable_, il est censé prendre un objet `Query` comme argument.

L'entité renvoyée aura été sauvegardée s'il s'agit d'un nouvel enregistrement. Cette méthode supporte les options suivantes :

- `atomic` Si les opérations `find` et `save` sont censées être effectuées à l'intérieur d'une transaction.
- `defaults` Défini à `false` pour ne pas définir les propriétés `$search` dans l'entity créée.

Créer Avec une Clé Primaire

Quand vous traitez des clés primaires UUID, vous avez souvent besoin de fournir une valeur générée ailleurs, au lieu d'un identifiant autogénéré pour vous. Dans ce cas, assurez-vous de ne pas passer la clé primaire au milieu des autres données. Au lieu de cela, assignez la clé primaire puis patchez les autres données dans l'entity :

```
$record = $table->newEmptyEntity();
$record->id = $existingUuid;
$record = $table->patchEntity($record, $existingData);
$table->saveOrFail($record);
```

Sauvegarder Plusieurs Entities

`Cake\ORM\Table::saveMany(iterable $entities, array $options = [])`

Cette méthode vous permet de sauvegarder plusieurs entities de façon atomique. `$entities` peut être un tableau d'entities créées avec `newEntities()` / `patchEntities()`. `$options` peut avoir les mêmes options que celles acceptées par `save()` :

```
$data = [
    [
        'title' => 'Premier post',
        'published' => 1
    ],
    [
        'title' => 'Second post',
        'published' => 1
    ],
];

$articles = $this->getTableLocator()->get('Articles');
$entities = $articles->newEntities($data);
$result = $articles->saveMany($entities);
```

La méthode renvoie les entities mises à jour en cas de succès, ou `false` en cas d'échec.

Mises à Jour en Masse

`Cake\ORM\Table::updateAll($fields, $conditions)`

Il y a des cas où la mise à jour de lignes individuelles n'est pas efficace ni nécessaire. Dans ce cas, il est préférable d'utiliser une mise à jour en masse pour modifier plusieurs lignes en une fois, en assignant les nouvelles valeurs des champs et les conditions de mise à jour :

```
// Publie tous les articles non publiés.
function publishAllUnpublished()
{
    $this->updateAll(
        [  // champs
            'published' => true,
            'publish_date' => FrozenTime::now()
        ],
        [  // conditions
            'published' => false
        ]
    );
}
```

Si vous devez faire des mises à jour en masse et utiliser des expressions SQL, vous devrez utiliser un objet expression puisque `updateAll()` utilise des requêtes préparées sous le capot :

```
use Cake\Database\Expression\QueryExpression;

...

function incrementCounters()
```

(suite sur la page suivante)

(suite de la page précédente)

```
{
    $expression = new QueryExpression('view_count = view_count + 1');
    $this->updateAll([$expression], ['published' => true]);
}
```

Une mise à jour en masse sera considérée comme réussie si une ou plusieurs lignes sont mises à jour.

Avertissement

updateAll *ne* va *pas* déclencher d'events beforeSave/afterSave. Si vous avez besoin de ceux-ci, chargez d'abord une collection d'enregistrements et mettez les à jour.

`updateAll()` est seulement une fonction de commodité. Vous pouvez également utiliser cette interface plus flexible :

```
// Publication de tous les articles non publiés.
function publishAllUnpublished()
{
    $this->query()
        ->update()
        ->set(['published' => true])
        ->where(['published' => false])
        ->execute();
}
```

Reportez-vous à la section *Mettre à Jour les Données*.

Supprimer des Données

class Cake\ORM**Table**

Cake\ORM\Table::**delete**(*EntityInterface $entity*, *array $options = []*)

Une fois que vous avez chargé une entity, vous pouvez la supprimer en appelant la méthode delete de la table d'origine :

```
// Dans un controller.
$entity = $this->Articles->get(2);
$result = $this->Articles->delete($entity);
```

Quand vous supprimez des entities, quelques actions se passent :

1. Les *règles de suppression* seront appliquées. Si les règles échouent, la suppression sera empêchée.
2. L'évènement `Model.beforeDelete` est déclenché. Si cet évènement est arrêté, la suppression sera abandonnée et les résultats de l'évènement seront retournés.
3. L'entity sera supprimée.
4. Toutes les associations dépendantes seront supprimées. Si les associations sont supprimées en tant qu'entities, des événements supplémentaires seront dispatchés.
5. Tout enregistrement de table jointe pour les associations BelongsToMany sera retirées.
6. L'évènement `Model.afterDelete` sera déclenché.

Par défaut, toutes les suppressions se passent dans une transaction. Vous pouvez désactiver la transaction avec l'option atomic :

```
$result = $this->Articles->delete($entity, ['atomic' => false]);
```

Suppression en Cascade

Quand les entities sont supprimées, les données associées peuvent aussi être supprimées. Si vos associations HasOne et HasMany sont configurées avec `dependent`, les opérations de suppression se feront aussi en "cascade" sur leurs entitites. Par défaut, les entities dans les tables associées sont retirées en utilisant `Cake\ORM\Table::deleteAll()`. Vous pouvez choisir que l'ORM charge les entities liées et les supprime individuellement en configurant l'option `cascadeCallbacks` à `true`. Un exemple d'association HasMany avec ces deux options activées serait :

```
// Dans une méthode initialize de Table.
$this->hasMany('Comments', [
    'dependent' => true,
    'cascadeCallbacks' => true,
]);
```

> **Note**
>
> Définir `cascadeCallbacks` à `true`, entrainera des lenteurs supplémentaires des suppressions par rapport aux suppressions de masse. L'option cascadeCallbacks doit seulement être activée quand votre application a un travail important de gestion des écouteurs d'événements.

Suppressions en Masse

`Cake\ORM\Table::deleteAll(`*$conditions*`)`

Il peut arriver des fois où la suppression de lignes une par une n'est pas efficace ou utile. Dans ces cas, il est plus performant d'utiliser une suppression en masse pour retirer plusieurs lignes en une fois :

```
// Supprime tous les spams
public function destroySpam()
{
    return $this->deleteAll(['is_spam' => true]);
}
```

Une suppression en masse va être considérée comme réussie si une ou plusieurs lignes ont été supprimées.

> **Avertissement**
>
> deleteAll *ne* va *pas* déclencher les événements beforeDelete/afterDelete. Si vous avez besoin d'eux, chargez d'abord une collection d'enregistrements et supprimez les.

Suppressions strictes

`Cake\ORM\Table::deleteOrFail(`*EntityInterface $entity*, *array $options = []*`)`

Utiliser cette méthode lancera une `Cake\ORM\Exception\PersistenceFailedException` si :

- l'entity est _new_ (si elle n'a jamais été persistée)
- l'entity n'a pas de valeur pour sa clé primaire
- les règles de validation ont échoué
- la suppression a été annulée via un _callback_.

Si vous voulez trouver l'entity qui n'a pas pu être sauvegardée, vous pouvez utiliser la méthode `Cake\ORMException\PersistenceFailedException::getEntity()` :

```
try {
    $table->deleteOrFail($entity);
} catch (\Cake\ORM\Exception\PersistenceFailedException $e) {
    echo $e->getEntity();
}
```

Puisque cette méthode utilise la méthode *Cake\ORM\Table::delete()*, tous les événements de `delete` seront déclenchés.

Associations - Lier les Tables Ensemble

Définir les relations entre les différents objets dans votre application sera un processus naturel. Par exemple, un article peut avoir plusieurs commentaires, et appartenir à un auteur. Les Auteurs peuvent avoir plusieurs articles et plusieurs commentaires. Les quatre types d'association dans CakePHP sont : hasOne, hasMany, belongsTo, et belongsToMany.

Relation	Type d'Association	Exemple
one to one	hasOne	Un user a un profile.
one to many	hasMany	Un user peut avoir plusieurs articles.
many to one	belongsTo	Plusieurs articles appartiennent à un user.
many to many	belongsToMany	Les Tags appartiennent aux articles.

Les Associations sont définies durant la méthode `initialize()` de votre objet table. Les méthodes ayant pour nom le type d'association vous permettent de définir les associations dans votre application. Par exemple, si nous souhaitions définir une association belongsTo dans notre ArticlesTable :

```
namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{
    public function initialize(array $config): void
    {
        $this->belongsTo('Authors');
    }
}
```

La forme la plus simple de toute configuration d'association prend l'alias de la table avec laquelle vous souhaitez l'associer. Par défaut, tous les détails d'une association vont utiliser les conventions de CakePHP. Si vous souhaitez personnaliser la façon dont sont gérées vos associations, vous pouvez les modifier avec les setters :

```
class ArticlesTable extends Table
{
    public function initialize(array $config): void
    {
        $this->belongsTo('Authors', [
                'className' => 'Publishing.Authors'
            ])
            ->setForeignKey('authorid')
            ->setProperty('person');
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }
}
```

Vous pouvez également configurer votre association à l'aide d'un tableau de paramètres :

```
$this->belongsTo('Authors', [
   'className' => 'Publishing.Authors',
   'foreignKey' => 'authorid',
   'propertyName' => 'person'
]);
```

La même table peut être utilisée plusieurs fois pour définir différents types d'associations. Par exemple considérons le cas où vous voulez séparer les commentaires approuvés et ceux qui n'ont pas encore été modérés :

```
class ArticlesTable extends Table
{
    public function initialize(array $config): void
    {
        $this->hasMany('Comments')
            ->setConditions(['approved' => true]);

        $this->hasMany('UnapprovedComments', [
                'className' => 'Comments'
            ])
            ->setConditions(['approved' => false])
            ->setProperty('unapproved_comments');
    }
}
```

Comme vous pouvez le voir, en spécifiant la clé `className`, il est possible d'utiliser la même table avec des associations différentes pour la même table. Vous pouvez même créer les tables associées avec elles-même pour créer des relations parent-enfant :

```
class CategoriesTable extends Table
{
    public function initialize(array $config): void
    {
        $this->hasMany('SubCategories', [
            'className' => 'Categories'
        ]);

        $this->belongsTo('ParentCategories', [
            'className' => 'Categories'
        ]);
    }
}
```

Vous pouvez aussi définir les associations en masse via un appel unique à la méthode `Table::addAssociations()` qui accepte en paramètre un tableau contenant les noms de tables indexés par association :

```
class PostsTable extends Table
{
    public function initialize(array $config): void
```

(suite sur la page suivante)

(suite de la page précédente)

```
    {
        $this->addAssociations([
            'belongsTo' => [
                'Users' => ['className' => 'App\Model\Table\UsersTable']
            ],
            'hasMany' => ['Comments'],
            'belongsToMany' => ['Tags']
        ]);
    }
}
```

Chaque type d'association accepte plusieurs associations où les clés sont les alias et les valeurs sont les données de configuration de l'association. Si une clé numérique est utilisée, la valeur sera traitée en tant qu'alias.

Associations HasOne

Mettons en place une Table Users avec une relation de type hasOne (a une seule) Table Addresses.

Tout d'abord, les tables de votre base de données doivent être saisies correctement. Pour qu'une relation de type hasOne fonctionne, une table doit contenir une clé étrangère qui pointe vers un enregistrement de l'autre. Dans notre cas, la table addresses contiendra un champ nommé `user_id`. Le motif de base est :

hasOne : l'"*autre* model contient la clé étrangère.

Relation	Schema
Users hasOne Addresses	addresses.user_id
Doctors hasOne Mentors	mentors.doctor_id

> **Note**
>
> Il n'est pas obligatoire de suivre les conventions de CakePHP, vous pouvez outrepasser l'utilisation de toute clé étrangère dans les définitions de vos associations. Néanmoins, coller aux conventions donnera un code moins répétitif, plus facile à lire et à maintenir.

Si nous avions les classes `UsersTable` et `AddressesTable`, nous pourrions faire l'association avec le code suivant :

```
class UsersTable extends Table
{
    public function initialize(array $config): void
    {
        $this->hasOne('Addresses');
    }
}
```

Si vous avez besoin de plus de contrôle, vous pouvez définir vos associations en utilisant les setters. Par exemple, vous voudrez peut-être limiter l'association pour inclure seulement certains enregistrements :

```
class UsersTable extends Table
{
    public function initialize(array $config): void
    {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        $this->hasOne('Addresses')
            ->setName('Addresses')
            ->setConditions(['Addresses.primary' => '1'])
            ->setDependent(true);
    }
}
```

Les clés possibles pour une association hasOne sont :

- **className** : le nom de la classe de la table que l'on souhaite associer au model actuel. Si l'on souhaite définir la relation "User a une Address", la valeur associée à la clé "className" devra être "Addresses".
- **foreignKey** : le nom de la clé étrangère que l'on trouve dans l'autre table. Ceci sera particulièrement pratique si vous avez besoin de définir des relations hasOne multiples. La valeur par défaut de cette clé est le nom du model actuel (avec des underscores) suffixé avec "_id". Dans l'exemple ci-dessus la valeur par défaut aurait été "user_id".
- **bindingKey** : le nom de la colonne dans la table courante, qui sera utilisée pour correspondre à la `foreignKey`. S'il n'est pas spécifié, la clé primaire (par exemple la colonne id de la table `Users`) sera utilisée.
- **conditions** : un tableau des conditions compatibles avec find() ou un fragment de code SQL tel que `['Addresses.primary' => true]`.
- **joinType** : le type de join à utiliser dans la requête SQL, par défaut à LEFT. Vous voulez peut-être utiliser INNER si votre association hasOne est requis.
- **dependent** : Quand la clé dependent est définie à `true`, et qu'une entity est supprimée, les enregistrements du model associé sont aussi supprimés. Dans ce cas, nous le définissons à `true` pour que la suppression d'un User supprime aussi son Address associée.
- **cascadeCallbacks** : Quand ceci et **dependent** sont à `true`, les suppressions en cascade vont charger et supprimer les entities pour que les callbacks soient lancés correctement. Quand il est à `false`. `deleteAll()` est utilisée pour retirer les données associées et que aucun callback ne soit lancé.
- **propertyName** : Le nom de la propriété qui doit être rempli avec les données d'une table associée dans les résultats d'une table source. Par défaut, c'est un nom en underscore et singulier de l'association, donc `address` dans notre exemple.
- **strategy** : La stratégie de requête utilisée pour charger les données correspondantes de l'autre table. Les valeurs acceptées sont `'join'` et `'select'`. L'utilisation de `'select'` générera une requête séparée, ce qui peut être utile quand l'autre table se trouve dans une base de données différente. Par défaut à `'join'`.
- **finder** : La méthode finder à utiliser lors du chargement des enregistrements associés.

Une fois que cette association a été définie, les opérations find sur la table Users peuvent contenir l'enregistrement Address, s'il existe :

```
// Dans un controller ou dans une méthode table.
$query = $users->find('all')->contain(['Addresses']);
foreach ($query as $user) {
    echo $user->address->street;
}
```

Ce qui est au-dessus génèrera une commande SQL similaire à :

```
SELECT * FROM users INNER JOIN addresses ON addresses.user_id = users.id;
```

Associations BelongsTo

Maintenant que nous avons un accès des données Address à partir de la table User, définissons une association belongsTo dans la table Addresses afin d'avoir un accès aux données liées de l'User. L'association belongsTo est un complément naturel aux associations hasOne et hasMany, permettant de voir les données associées dans l'autre sens.

Lorsque vous remplissez les clés des tables de votre base de données pour une relation belongsTo, suivez cette convention :

belongsTo : le model *courant* contient la clé étrangère.

Relation	Schema
Addresses belongsTo Users	addresses.user_id
Mentors belongsTo Doctors	mentors.doctor_id

Astuce

Si une Table contient une clé étrangère, elle appartient à (belongsTo) l'autre Table.

Nous pouvons définir l'association belongsTo dans notre table Addresses comme ce qui suit :

```
class AddressesTable extends Table
{

    public function initialize(array $config): void
    {
        $this->belongsTo('Users');
    }
}
```

Nous pouvons aussi définir une relation plus spécifique en utilisant les setters :

```
class AddressesTable extends Table
{

    public function initialize(array $config): void
    {
        $this->belongsTo('Users')
            ->setForeignKey('user_id') // Avant la version CakePHP 3.4, utilisez␣
↪foreignKey() au lieu de setForeignKey()
            ->setJoinType('INNER');
    }
}
```

Les clés possibles pour les tableaux d'association belongsTo sont :

- **className** : le nom de classe du model associé au model courant. Si vous définissez une relation "Profile belongsTo User", la clé className devra être "Users".
- **foreignKey** : le nom de la clé étrangère trouvée dans la table courante. C'est particulièrement pratique si vous avez besoin de définir plusieurs relations belongsTo au même model. La valeur par défaut pour cette clé est le nom au singulier de l'autre model avec des underscores, suffixé avec `_id`.
- **bindingKey** : le nom de la colonne dans l'autre table, qui sera utilisée pour correspondre à la `foreignKey`. S'il n'est pas spécifié, la clé primaire (par exemple la colonne id de la table `Users`) sera utilisée.
- **conditions** : un tableau de conditions compatibles find() ou de chaînes SQL comme `['Users.active' => true]`
- **joinType** : le type de join à utiliser dans la requête SQL, par défaut LEFT ce qui peut ne pas correspondre à vos besoins dans toutes les situations, INNER peut être utile quand vous voulez tout de votre model principal ainsi que de vos models associés !
- **propertyName** : Le nom de la propriété qui devra être remplie avec les données de la table associée dans les résultats de la table source. Par défaut il s'agit du nom singulier avec des underscores de l'association donc `user` dans notre exemple.

— **strategy** : La stratégie de requête utilisée pour charger les données correspondantes de l'autre table. Les valeurs acceptées sont `'join'` et `'select'`. L'utilisation de `'select'` générera une requête séparée, ce qui peut être utile quand l'autre table se trouve dans une base de données différente. Par défaut à `'join'`.
— **finder** : La méthode finder à utiliser lors du chargement des enregistrements associés.

Une fois que cette association a été définie, les opérations find sur la table Addresses peuvent contenir l'enregistrement User s'il existe :

```
// Dans un controller ou dans une méthode table.
$query = $addresses->find('all')->contain(['Users']);
foreach ($query as $address) {
    echo $address->user->username;
}
```

Ce qui est au-dessus génèrera une commande SQL similaire à :

```
SELECT * FROM addresses LEFT JOIN users ON addresses.user_id = users.id;
```

Associations HasMany

Un exemple d'association hasMany est « Article hasMany Comments » (Un Article a plusieurs Commentaires). Définir cette association va nous permettre de récupérer les commentaires d'un article quand l'article est chargé.

Lors de la création des tables de votre base de données pour une relation hasMany, suivez cette convention :

hasMany : l'*autre* model contient la clé étrangère.

Relation	Schema
Article hasMany Comment	Comment.article_id
Product hasMany Option	Option.product_id
Doctor hasMany Patient	Patient.doctor_id

Nous pouvons définir l'association hasMany dans notre model Articles comme suit :

```
class ArticlesTable extends Table
{

    public function initialize(array $config): void
    {
        $this->hasMany('Comments');
    }
}
```

Nous pouvons également définir une relation plus spécifique en utilisant les setters :

```
class ArticlesTable extends Table
{

    public function initialize(array $config): void
    {
        $this->hasMany('Comments')
            ->setForeignKey('article_id')
            ->setDependent(true);
    }
}
```

Parfois vous voudrez configurer les clés composites dans vos associations :

```
// Dans l'appel ArticlesTable::initialize()
$this->hasMany('Reviews')
    ->setForeignKey([
        'article_id',
        'article_hash'
    ]);
```

En se référant à l'exemple du dessus, nous avons passé un tableau contenant les clés composites dans `setForeignKey()`. Par défaut `bindingKey` serait automatiquement défini respectivement avec `id` et `hash`, mais imaginons que vous souhaitiez spécifier avec des champs de liaisons différents de ceux par défault, vous pouvez les configurer manuellement via `setForeignKey()` :

```
// Dans un appel de ArticlesTable::initialize()
$this->hasMany('Reviews')
    ->setForeignKey([
        'article_id',
        'article_hash'
    ])
    ->setBindingKey([
        'whatever_id',
        'whatever_hash'
    ]);
```

Il est important de noter que les valeurs de `foreignKey` font référence à la table **reviews** et les valeurs de `bindingKey` font référence à la table **articles**.

Les clés possibles pour les tableaux d'association hasMany sont :

- **className** : le nom de la classe du model que l'on souhaite associer au model actuel. Si l'on souhaite définir la relation "User hasMany Comment" (l'User a plusieurs Commentaires), la valeur associée à la clef "className" devra être "Comments".
- **foreignKey** : le nom de la clé étrangère que l'on trouve dans l'autre table. Ceci sera particulièrement pratique si vous avez besoin de définir plusieurs relations hasMany. La valeur par défaut de cette clé est le nom du model actuel (avec des underscores) suffixé avec "_id".
- **bindingKey** : le nom de la colonne dans la table courante, qui sera utilisée pour correspondre à la `foreignKey`. S'il n'est pas spécifié, la clé primaire (par exemple la colonne id de la table `Users`) sera utilisée.
- **conditions** : un tableau de conditions compatibles avec find() ou des chaînes SQL comme `['Comments.visible' => true]`.
- **sort** : un tableau compatible avec les clauses order de find() ou les chaînes SQL comme `['Comments.created' => 'ASC']`.
- **dependent** : Lorsque dependent vaut `true`, une suppression récursive du model est possible. Dans cet exemple, les enregistrements Comment seront supprimés lorsque leur Article associé l'aura été.
- **cascadeCallbacks** : Quand ceci et **dependent** sont à `true`, les suppressions en cascade chargeront les entities supprimés pour que les callbacks soient correctement lancés. Si à `false`. `deleteAll()` est utilisée pour retirer les données associées et aucun callback ne sera lancé.
- **propertyName** : Le nom de la propriété qui doit être rempli avec les données des Table associées dans les résultats de la table source. Par défaut, celui-ci est le nom au pluriel et avec des underscores de l'association donc `comments` dans notre exemple.
- **strategy** : Définit la stratégie de requête à utiliser. Par défaut à "select". L'autre valeur valide est "subquery", qui remplace la liste `IN` avec une sous-requête équivalente.
- **saveStrategy** : Soit "append" ou bien "replace". Par défaut à "append". Quand "append" est choisi, les enregistrements existants sont ajoutés aux enregistrements de la base de données. Quand "replace" est choisi, les enregistrements associés qui ne sont pas dans l'ensemble actuel seront retirés. Si la clé étrangère est une colonne qui peut être null ou si `dependent` est à true, les enregistrements seront orphelins.

— **finder** : La méthode finder à utiliser lors du chargement des enregistrements associés.

Une fois que cette association a été définie, les opérations de recherche sur la table Articles récupèreront également les Comments liés s'ils existent :

```
// Dans un controller ou dans une méthode de table.
$query = $articles->find('all')->contain(['Comments']);
foreach ($query as $article) {
    echo $article->comments[0]->text;
}
```

Ce qui est au-dessus génèrera une commande SQL similaire à :

```
SELECT * FROM articles;
SELECT * FROM comments WHERE article_id IN (1, 2, 3, 4, 5);
```

Quand la stratégie de sous-requête est utilisée, une commande SQL similaire à ce qui suit sera générée :

```
SELECT * FROM articles;
SELECT * FROM comments WHERE article_id IN (SELECT id FROM articles);
```

Vous voudrez peut-être mettre en cache les compteurs de vos associations hasMany. C'est utile quand vous avez souvent besoin de montrer le nombre d'enregistrements associés, mais que vous ne souhaitez pas charger tous les articles juste pour les compter. Par exemple, le compteur de comment sur n'importe quel article donné est souvent mis en cache pour rendre la génération des lists d'article plus efficace. Vous pouvez utiliser *CounterCacheBehavior* pour mettre en cache les compteurs des enregistrements associés.

Assurez-vous que vos tables de base de données ne contiennent pas de colonnes du même nom que les attributs d'association. Si par exemple vous avez un champs counter en collision avec une propriété d'association, vous devez soit renommer l'association ou le nom de la colonne.

Associations BelongsToMany

Un exemple d'association BelongsToMany est « Article BelongsToMany Tags », où les tags d'un article sont partagés avec d'autres articles. BelongsToMany fait souvent référence au « has and belongs to many », et est une association classique « many to many ».

La principale différence entre hasMany et BelongsToMany est que le lien entre les modèles dans une association BelongsToMany n'est pas exclusif. Par exemple nous joignons notre table Articles avec la table Tags. Utiliser "funny" comme un Tag pour mon Article, n"« épuise » pas le tag. Je peux toujours l'utiliser pour le prochain article que j'écris.

Trois tables de la base de données sont nécessaires pour une association BelongsToMany. Dans l'exemple du dessus, nous aurons besoin des tables pour `articles`, `tags` et `articles_tags`. La table `articles_tags` contient les données qui font le lien entre les tags et les articles. La table de jointure est nommée à partir des deux tables impliquées, séparée par un underscore par convention. Dans sa forme la plus simple, cette table se résume à `article_id` et `tag_id`.

belongsToMany nécessite une table de jointure séparée qui inclut deux noms de *model*.

Relation	Champs de la table de jointure
Article belongsToMany Tag	articles_tags.id, articles_tags.tag_id, articles_tags.article_id
Patient belongsToMany Doctor	doctors_patients.id, doctors_patients.doctor_id, doctors_patients.patient_id.

Nous pouvons définir l'association belongsToMany dans nos deux models comme suit :

```
// Dans src/Model/Table/ArticlesTable.php
class ArticlesTable extends Table
```

(suite sur la page suivante)

(suite de la page précédente)

```
{

    public function initialize(array $config): void
    {
        $this->belongsToMany('Tags');
    }
}

// Dans src/Model/Table/TagsTable.php
class TagsTable extends Table
{

    public function initialize(array $config): void
    {
        $this->belongsToMany('Articles');
    }
}
```

Nous pouvons aussi définir une relation plus spécifique en passant un tableau de configuration :

```
// In src/Model/Table/TagsTable.php
class TagsTable extends Table
{

    public function initialize(array $config): void
    {
        $this->belongsToMany('Articles', [
            'joinTable' => 'articles_tags',
        ]);
    }
}
```

Les clés possibles pour un tableau définissant une association belongsToMany sont :

- **className** : Le nom de la classe du model que l'on souhaite associer au model actuel. Si l'on souhaite définir la relation "Article belongsToMany Tag", la valeur associée à la clef "className" devra être "Tags".
- **joinTable** : Le nom de la table de jointure utilisée dans cette association (si la table ne colle pas à la convention de nommage des tables de jointure belongsToMany). Par défaut, le nom de la table sera utilisé pour charger l'instance Table pour la table de jointure/pivot.
- **foreignKey** : le nom de la clé étrangère dans la table de jointure et qui fait référence au model actuel ou la liste en cas de clés étrangères composites. Ceci est particulièrement pratique si vous avez besoin de définir plusieurs relations belongsToMany. La valeur par défaut de cette clé est le nom du model actuel (avec des underscores) avec le suffixe "_id".
- **bindingKey** : le nom de la colonne dans l'autre table, qui sera utilisée pour correspondre à la `foreignKey`. S'il n'est pas spécifié, la clé primaire (par exemple la colonne id de la table `Users`) sera utilisée.
- **targetForeignKey** : le nom de la clé étrangère dans la table de jointure pour le model cible ou la liste en cas de clés étrangères composites. La valeur par défaut pour cette clé est le model cible, au singulier et en underscore, avec le suffixe "_id".
- **conditions** : un tableau de conditions compatibles avec `find()`. Si vous avez des conditions sur une table associée, vous devriez utiliser un model "through" et lui définir les associations belongsTo nécessaires.
- **sort** : un tableau de clauses order compatible avec find().
- **dependent** : Quand la clé dependent est définie à `false` et qu'une entity est supprimée, les enregistrements de la table de jointure ne seront pas supprimés.
- **through** : Vous permet de fournir soit le nom de l'instance de la Table que vous voulez utiliser, soit l'instance

elle-même. Cela rend possible la personnalisation des clés de la table de jointure, et vous permet de personnaliser le comportement de la table pivot.
— **cascadeCallbacks** : Quand définie à `true`, les suppressions en cascade vont charger et supprimer les entities ainsi les callbacks sont correctement lancés sur les enregistrements de la table de jointure. Quand définie à `false`. `deleteAll()` est utilisée pour retirer les données associées et aucun callback n'est lancé. Ceci est par défaut à `false` pour réduire la charge.
— **propertyName** : Le nom de la propriété qui doit être remplie avec les données de la table associée dans les résultats de la table source. Par défaut c'est le nom au pluriel, avec des underscores de l'association, donc `tags` dans notre exemple.
— **strategy** : Définit la stratégie de requête à utiliser. Par défaut à "select". L'autre valeur valide est "subquery", qui remplace la liste `IN` avec une sous-requête équivalente.
— **saveStrategy** : Soit "append" ou bien "replace". Par défaut à "replace". Indique le mode à utiliser pour sauvegarder les entities associées. Le premier va seulement créer des nouveaux liens entre les deux côtés de la relation et le deuxième va effacer et remplacer pour créer les liens entre les entities passées lors de la sauvegarde.
— **finder** : La méthode finder à utiliser lors du chargement des enregistrements associés.

Une fois que cette association a été définie, les opérations find sur la table Articles peuvent contenir les enregistrements de Tag s'ils existent :

```
// Dans un controller ou dans une méthode table.
$query = $articles->find('all')->contain(['Tags']);
foreach ($query as $article) {
    echo $article->tags[0]->text;
}
```

Ce qui est au-dessus génèrera une requête SQL similaire à :

```
SELECT * FROM articles;
SELECT * FROM tags
INNER JOIN articles_tags ON (
  tags.id = article_tags.tag_id
  AND article_id IN (1, 2, 3, 4, 5)
);
```

Quand la stratégie de sous-requête est utilisée, un SQL similaire à ce qui suit sera générée :

```
SELECT * FROM articles;
SELECT * FROM tags
INNER JOIN articles_tags ON (
  tags.id = article_tags.tag_id
  AND article_id IN (SELECT id FROM articles)
);
```

Utiliser l'Option "through"

Si vous souhaitez ajouter des informations supplémentaires à la table join/pivot, ou si vous avez besoin d'utiliser les colonnes jointes en dehors des conventions, vous devrez définir l'option `through`. L'option `through` vous fournit un contrôle total sur la façon dont l'association belongsToMany sera créée.

Il est parfois souhaitable de stocker des données supplémentaires avec une association many to many. Considérez ce qui suit :

```
Student BelongsToMany Course
Course BelongsToMany Student
```

Un Etudiant (Student) peut prendre plusieurs Cours (many Courses) et un Cours (Course) peut être pris par plusieurs Etudiants (many Students). C'est une simple association many to many. La table suivante suffira :

```
id | student_id | course_id
```

Maintenant si nous souhaitons stocker le nombre de jours qui sont attendus par l'étudiant sur le cours et leur note finale ? La table que nous souhaiterions serait :

```
id | student_id | course_id | days_attended | grade
```

La façon d'intégrer notre besoin est d'utiliser un **model join**, autrement connu comme une association **hasMany through**. Ceci étant, l'association est un model lui-même. Donc, nous pouvons créer un nouveau model CoursesMemberships. Regardez les models suivants :

```
class StudentsTable extends Table
{
    public function initialize(array $config): void
    {
        $this->belongsToMany('Courses', [
            'through' => 'CoursesMemberships',
        ]);
    }
}

class CoursesTable extends Table
{
    public function initialize(array $config): void
    {
        $this->belongsToMany('Students', [
            'through' => 'CoursesMemberships',
        ]);
    }
}

class CoursesMembershipsTable extends Table
{
    public function initialize(array $config): void
    {
        $this->belongsTo('Students');
        $this->belongsTo('Courses');
    }
}
```

La table de jointure CoursesMemberships identifie de façon unique une participation donnée d'un Etudiant à un Cours en plus des meta-informations supplémentaires.

Conditions d'Association par Défaut

L'option `finder` vous permet d'utiliser un *finder personnalisé* pour charger les données associées. Ceci permet de mieux encapsuler vos requêtes et de garder votre code plus DRY. Il y a quelques limitations lors de l'utilisation de finders pour charger les données dans les associations qui sont chargées en utilisant les jointures (belongsTo/hasOne). Les seuls aspects de la requête qui seront appliqués à la requête racine sont les suivants :

- WHERE conditions.
- Additional joins.
- Contained associations.

Les autres aspects de la requête, comme les colonnes sélectionnées, l'order, le group by, having et les autres sous-instructions, ne seront pas appliqués à la requête racine. Les associations qui *ne* sont *pas* chargées avec les jointures (hasMany/belongsToMany), n'ont pas les restrictions ci-dessus et peuvent aussi utiliser les formateurs de résultats ou les fonctions map/reduce.

Charger les Associations

Une fois que vous avez défini vos associations, vous pouvez *charger en eager les associations* quand vous récupérez les résultats.

Behaviors (Comportements)

Les behaviors (comportements) sont une manière d'organiser et de réutiliser la logique de la couche Model. Conceptuellement, ils sont similaires aux traits. Cependant, les behaviors sont implémentés en classes séparées. Ceci leur permet de s'insérer dans le cycle de vie des callbacks que les models émettent, tout en fournissant des fonctionnalités de type trait.

Les Behaviors fournissent une façon pratique de packager un behavior qui est commun à plusieurs models. Par exemple, CakePHP intègre un `TimestampBehavior`. Plusieurs models voudront des champs timestamp, et la logique pour gérer ces champs n'est pas spécifique à un seul model. C'est dans ce genre de scénario que les behaviors sont utiles.

Utiliser les Behaviors

Les Behaviors fournissent un moyen de créer des parties de logique réutilisables horizontalement liées aux classes table. Vous vous demandez peut-être pourquoi les behaviors sont des classes classiques et non des traits. La raison principale tient aux écouteurs d'event. Alors que les traits permettent de réutiliser des parties de logique, ils compliqueraient la liaison des events.

Pour ajouter un behavior à votre table, vous pouvez appeler la méthode `addBehavior()`. Généralement, le meilleur endroit pour le faire est dans la méthode `initialize()` :

```
namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{
    public function initialize(array $config): void
    {
        $this->addBehavior('Timestamp');
    }
}
```

Comme pour les associations, vous pouvez utiliser la *syntaxe de plugin* et fournir des options de configuration supplémentaires :

```
namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{
    public function initialize(array $config): void
    {
        $this->addBehavior('Timestamp', [
```

(suite sur la page suivante)

(suite de la page précédente)

```
            'events' => [
                'Model.beforeSave' => [
                    'created_at' => 'new',
                    'modified_at' => 'always'
                ]
            ]
        ]);
    }
}
```

Behaviors du Cœur

CounterCache

class Cake\ORM\Behavior**CounterCacheBehavior**

Souvent les applications web doivent afficher le nombre d'objets liés. Par exemple, quand vous montrez une liste d'articles, vous voulez peut-être afficher combien de commentaires ils ont. Ou quand vous montrez un utilisateur, vous voulez montrer le nombre d'amis/de followers qu'il a. Le behavior CounterCache est présent pour ces situations. CounterCache va mettre à jour un champ dans les models associés assignés dans les options quand il est invoqué. Les champs doivent exister dans la base de données et être de type INT.

Usage Basique

Vous activez le behavior CounterCache comme tous les autres behaviors, mais il ne fera rien jusqu'à ce que vous configuriez quelques relations et le nombre de champs qui doivent être stockés sur chacun d'eux. Utiliser notre exemple ci-dessous, nous pourrions mettre en cache le nombre de commentaires pour chaque article avec ce qui suit :

```
class CommentsTable extends Table
{
    public function initialize(array $config): void
    {
        $this->addBehavior('CounterCache', [
            'Articles' => ['comment_count']
        ]);
    }
}
```

> **Note**
>
> Il doit y avoir une colonne `comment_count` dans la table `articles`.

La configuration de CounterCache doit être composée de noms de relations et de la configuration spécifique pour cette relation.

Comme vous le voyez, il faut ajouter le behavior « de l'autre côté » de l'association dans laquelle vous voulez que le champ soit mis à jour. Dans cet exemple, le behavior est ajouté dans la table `CommentsTable` bien qu'il actualise le champ `comment_count` dans la table `ArticlesTable`.

La valeur du compteur sera mise à jour à chaque fois qu'une entity est sauvegardée ou supprimée. Le compteur **ne va pas** être mis à jour lorsque vous

- sauvegardez une entity sans changer ses données,
- ou utilisez `updateAll()`,

— ou utilisez `deleteAll()`,
— ou exécutez du SQL que vous avez écrit vous-même.

Usage Avancé

Si vous avez besoin de garder un compteur mis en cache pour moins que tous les enregistrements liés, vous pouvez fournir des conditions supplémentaires ou des méthodes finder pour générer une valeur du compteur :

```
// Utilise une méthode find spécifique.
// Dans ce cas find(published)
$this->addBehavior('CounterCache', [
    'Articles' => [
        'comment_count' => [
            'finder' => 'published'
        ]
    ]
]);
```

Si vous n'avez pas de méthode de finder personnalisé, vous pouvez fournir un tableau de conditions pour trouver les enregistrements à la place :

```
$this->addBehavior('CounterCache', [
    'Articles' => [
        'comment_count' => [
            'conditions' => ['Comments.spam' => false]
        ]
    ]
]);
```

Si vous voulez que CounterCache mette à jour plusieurs champs, par exemple deux champs qui montrent un compte conditionnel et un compte basique, vous pouvez ajouter ces champs dans le tableau :

```
$this->addBehavior('CounterCache', [
    'Articles' => ['comment_count',
        'published_comment_count' => [
            'finder' => 'published'
        ]
    ]
]);
```

Si vous souhaitez calculer la valeur du champ de `CounterCache` par vous-même, vous pouvez définir l'option `ignoreDirty` à `true`. Cela empêchera le champ d'être recalculé automatiquement si vous l'avez définit `dirty` avant :

```
$this->addBehavior('CounterCache', [
    'Articles' => [
        'comment_count' => [
            'ignoreDirty' => true
        ]
    ]
]);
```

Enfin, si un finder personnalisé et les conditions ne sont pas réunies, vous pouvez fournir une méthode de callback. Cette méthode retourne la valeur du compteur à stocker :

```
$this->addBehavior('CounterCache', [
    'Articles' => [
        'rating_avg' => function ($event, $entity, $table) {
            return 4.5;
        }
    ]
]);
```

> **Note**
>
> Le comportement CounterCache fonctionne uniquement pour les associations `belongsTo`. Par exemple pour « Comments belongsTo Articles », vous devez ajouter le behavior CounterCache à la `CommentsTable` pour pouvoir générer `comment_count` pour la table Articles.

Utilisation avec Belongs To Many

Il est possible d'utiliser le behavior CounterCache dans une association `belongsToMany`. Avant tout, vous devez ajouter les options `through` et `cascadeCallbacks` à l'association `belongsToMany` :

```
'through'          => 'CommentsArticles',
'cascadeCallbacks' => true
```

Consultez également *Utiliser l'Option "through"* pour savoir comment configurer une jointure de table personnalisée. `CommentsArticles` est le nom de la classe de la table de jointure. Si vous ne l'avez pas, vous pouvez la créer avec l'outil bake en ligne de commande.

Dans ce `src/Model/Table/CommentsArticlesTable.php`, ajoutez ensuite le behavior avec le code décrit plus haut. :

```
$this->addBehavior('CounterCache', [
    'Articles' => [ 'comments_count' ],
]);
```

Pour finir, supprimez tous les caches avec `bin/cake cache clear_all` et faites l'essai.

Timestamp

class `Cake\ORM\Behavior\`**`TimestampBehavior`**

Le behavior timestamp permet à vos objets de table de mettre à jour un ou plusieurs timestamps sur chaque évènement de model. C'est principalement utilisé pour remplir les données dans les champs `created` et `modified`. Cependant, avec quelques configurations supplémentaires, vous pouvez mettre à jour la colonne timestamp/datetime sur chaque évènement qu'une table publie.

Utilisation Basique

Vous activez le behavior timestamp comme tout autre behavior :

```
class ArticlesTable extends Table
{
    public function initialize(array $config): void
    {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        $this->addBehavior('Timestamp');
    }
}
```

La configuration par défaut va faire ce qui suit :

- Quand une nouvelle entity est sauvegardée, les champs `created` et `modified` seront définis avec le time courant.
- Quand une entity est mise à jour, le champ `modified` est défini au time courant.

Utiliser et Configurer le Behavior

Si vous avez besoin de modifier les champs avec des noms différents, ou si vous souhaitez mettre à jour le timestamp supplémentaire sur des événements personnalisés, vous pouvez utiliser quelques configurations supplémentaires :

```
class OrdersTable extends Table
{
    public function initialize(array $config): void
    {
        $this->addBehavior('Timestamp', [
            'events' => [
                'Model.beforeSave' => [
                    'created_at' => 'new',
                    'updated_at' => 'always',
                ],
                'Orders.completed' => [
                    'completed_at' => 'always'
                ]
            ]
        ]);
    }
}
```

Comme vous pouvez le voir au-dessus, en plus de l'évènement standard `Model.beforeSave`, nous mettons aussi à jour la colonne `completed_at` quand les ordres sont complétés.

Mettre à jour les Timestamps sur les Entities

Parfois, vous souhaitez mettre à jour uniquement les timestamps sur une entity sans changer aucune autre propriété. On fait parfois référence au "touching" d'un enregistrement. Dans CakePHP, vous pouvez utiliser la méthode `touch()` pour faire exactement ceci :

```
// Touch basé sur l'évènement Model.beforeSave.
$articles->touch($article);

// Touch basé sur un évènement spécifique.
$orders->touch($order, 'Orders.completed');
```

Après avoir sauvegardé l'entity, le champ est mis à jour.

Toucher les enregistrements peut être utile quand vous souhaitez signaler qu'une ressource parente a changé quand une ressource enfante est créée/mise à jour. Par exemple : mettre à jour un article quand un nouveau commentaire est ajouté.

Sauvegardez les Mises à Jour sans Modifier les Timestamps

Pour désactiver la modification automatique de la colonne timestamp `updated` quand vous sauvegardez une entity, vous pouvez marquer l'attribut avec "dirty" :

```
// Marquer la colonne modified avec dirty
// la valeur actuelle à définir lors de la mise à jour.
$order->dirty('modified', true);
```

Translate

class Cake\ORM\Behavior**TranslateBehavior**

Le behavior Translate vous permet de créer et de récupérer les copies traduites de vos entities en plusieurs langues. Il le fait en utilisant une table `i18n` séparée où il stocke la traduction pour chacun des champs de tout objet Table donné auquel il est lié.

> **Avertissement**
>
> TranslateBehavior ne supporte pas les clés primaires composite pour l'instant.

Un Rapide Aperçu

Après avoir créé la table `i18n` dans votre base de données, attachez le behavior à l'objet Table que vous souhaitez rendre traduisible :

```
class ArticlesTable extends Table
{
    public function initialize(array $config)
    {
        $this->addBehavior('Translate', ['fields' => ['title']]);
    }
}
```

Maintenant, sélectionnez une langue à utiliser pour récupérer les entities :

```
// Dans un controller. Change la locale
I18n::setLocale('es');
$this->loadModel('Articles');
```

Ensuite, récupérez une entity existante :

```
$article = $this->Articles->get(12);
echo $article->title; // Affiche 'A title', pas encore traduit
```

Ensuite, traduisez votre entity :

```
$article->title = 'Un Artículo';
$this->Articles->save($article);
```

Vous pouvez maintenant essayer de récupérer à nouveau votre entity :

```
$article = $this->Articles->get(12);
echo $article->title; // Affiche 'Un Artículo', ouah facile!
```

Travailler avec plusieurs traductions peut être fait en utilisant un trait spécial dans votre classe Entity :

```
use Cake\ORM\Behavior\Translate\TranslateTrait;
use Cake\ORM\Entity;

class Article extends Entity
{
    use TranslateTrait;
}
```

Maintenant, vous pouvez trouver toutes les traductions pour une seule entity :

```
$article = $this->Articles->>find('translations')->first();
echo $article->translation('es')->title; // 'Un Artículo'

echo $article->translation('en')->title; // 'An Article';
```

Il est également facile de sauvegarder plusieurs traductions en une fois :

```
$article->translation('es')->title = 'Otro Título';
$article->translation('fr')->title = 'Un autre Titre';
$this->Articles->save($article);
```

Si vous voulez aller plus en profondeur sur la façon dont il fonctionne ou pour affiner le behavior à vos besoins, continuez de lire le reste de ce chapitre.

Initialiser la Table i18n de la Base de Données

Afin d'utiliser le behavior, vous avez besoin de créer une table `i18n` avec le bon schéma. Habituellement, la seule façon de charger la table `i18n` est en lançant manuellement le script SQL suivant dans votre base de données :

```
CREATE TABLE i18n (
    id int NOT NULL auto_increment,
    locale varchar(6) NOT NULL,
    model varchar(255) NOT NULL,
    foreign_key int(10) NOT NULL,
    field varchar(255) NOT NULL,
    content text,
    PRIMARY KEY     (id),
    UNIQUE INDEX I18N_LOCALE_FIELD(locale, model, foreign_key, field),
    INDEX I18N_FIELD(model, foreign_key, field)
);
```

Le schéma est aussi disponible sous la forme d'un fichier sql dans **/config/schema/i18n.sql**.

Une remarque sur les abréviations des langues : Le behavior Translate n'impose aucune restriction sur l'identifieur de langues, les valeurs possibles sont seulement restreintes par le type/la taille de la colonne `locale`. `locale` est définie avec `varchar(6)` dans le cas où vous souhaitez utiliser les abréviations comme `es-419` (Espagnol pour l'Amérique Latine, l'abréviation des langues avec le code de zone UN M.49 [135]).

Astuce

135. https://en.wikipedia.org/wiki/UN_M.49

> Il est sage d'utiliser les mêmes abréviations de langue que celles requises pour *l'Internationalisation et la Localisation*. Ainsi vous êtes cohérent et le changement de langue fonctionne de la même manière à la fois pour le `Translate Behaviour` et `l'Internationalisation et la Localisation`.

Il est donc recommandé d'utiliser soit le code ISO à 2 lettres de la langue, comme `en`, `fr`, `de`, soit le nom de la locale complète comme `fr_FR`, `es_AR`, `da_DK` qui contient à la fois la langue et le pays où elle est parlée.

Attacher le Behavior Translate à Vos Tables

Attacher le behavior peut être fait dans la méthode `initialize()` de votre classe Table :

```
class ArticlesTable extends Table
{

    public function initialize(array $config): void
    {
        $this->addBehavior('Translate', ['fields' => ['title', 'body']]);
    }
}
```

La première chose à noter est que vous devez passer la clé `fields` dans le tableau de configuration. La liste des champs est souhaitée pour dire au behavior les colonnes qui pourront stocker les traductions.

Utiliser une Table de Traductions Séparée

Si vous souhaitez utiliser une table autre que `i18n` pour la traduction d'un dépôt particulier, vous pouvez le spécifier dans la configuration du behavior. C'est le cas quand vous avez plusieurs tables à traduire et que vous souhaitez une séparation propre des données qui sont stockées pour chaque table spécifiquement :

```
class Articles extends Table
{

    public function initialize(array $config): void
    {
        $this->addBehavior('Translate', [
            'fields' => ['title', 'body'],
            'translationTable' => 'ArticlesI18n'
        ]);
    }
}
```

Vous avez besoin de vous assurer que toute table personnalisée que vous utilisez a les colonnes `field`, `foreign_key`, `locale` et `model`.

Lire du Contenu Traduit

Comme montré ci-dessus, vous pouvez utiliser la méthode `locale` pour choisir la traduction active pour les entities qui sont chargées :

```
use Cake\I18n\I18n;

// Change la langue dans votre action
I18n::setLocale('es');
```

(suite sur la page suivante)

(suite de la page précédente)

```
$this->loadModel('Articles');

// Toutes les entities dans les résultats vont contenir la traduction espagnol
$results = $this->Articles->find()->all();
```

Cette méthode fonctionne avec n'importe quel finder se trouvant dans vos tables. Par exemple, vous pouvez utiliser TranslateBehavior avec `find('list')` :

```
I18n::setLocale('es');
$data = $this->Articles->find('list')->toArray();

// Data va contenir
[1 => 'Mi primer artículo', 2 => 'El segundo artículo', 15 => 'Otro articulo' ...]
```

Récupérer Toutes les Traductions Pour Une Entity

Lorsque vous construisez des interfaces pour la mise à jour de contenu traduite, il est souvent utile de montrer une ou plusieurs traduction(s) au même moment. Vous pouvez utiliser le finder `translations` pour ceci :

```
// Récupère le premier article avec toutes les traductions correspondantes
$article = $this->Articles->find('translations')->first();
```

Dans l'exemple ci-dessus, vous obtiendrez une liste d'entities en retour qui a une propriété `_translations` définie. Cette propriété va contenir une liste d'entities de données traduites. Par exemple, les propriétés suivantes seront accessibles :

```
// Affiche 'en'
echo $article->_translations['en']->locale;

// Affiche 'title'
echo $article->_translations['en']->field;

// Affiche 'My awesome post!'
echo $article->_translations['en']->body;
```

Une façon plus élégante pour gérer les données est d'ajouter un trait pour la classe entity qui est utilisé pour votre table :

```
use Cake\ORM\Behavior\Translate\TranslateTrait;
use Cake\ORM\Entity;

class Article extends Entity
{
    use TranslateTrait;
}
```

Ce trait contient une méthode unique appelée `translation`, ce qui vous laisse accéder ou créer à la volée des entities pour de nouvelles traductions :

```
// Affiche 'title'
echo $article->translation('en')->title;

// Ajoute une nouvelle donnée de traduction de l'entity à l'article
$article->translation('deu')->title = 'Wunderbar';
```

Limiter les Traductions à Récupérer

Vous pouvez limiter les langues que vous récupérez à partir de la base de données pour un ensemble particulier d'enregistrements :

```
$results = $this->Articles->find('translations', [
    'locales' => ['en', 'es']
]);
$article = $results->first();
$spanishTranslation = $article->translation('es');
$englishTranslation = $article->translation('en');
```

Eviter la Récupération de Traductions Vides

Les enregistrements traduits peuvent contenir tout type de chaîne, si un enregistrement a été traduit et stocké comme étant une chaîne vide ("") le behavior translate va prendre et utiliser ceci pour écraser la valeur du champ originel.

Si ce n'est pas désiré, vous pouvez ignorer les traductions qui sont vides en utilisant la clé de config `allowEmptyTranslations` :

```
class ArticlesTable extends Table
{

    public function initialize(array $config): void
    {
        $this->addBehavior('Translate', [
            'fields' => ['title', 'body'],
            'allowEmptyTranslations' => false
        ]);
    }
}
```

Ce qui est au-dessus va seulement charger les données traduites qui ont du contenu.

Récupérer Toutes les Traductions pour des Associations

Il est aussi possible de trouver des traductions pour toute association dans une unique opération de find :

```
$article = $this->Articles->find('translations')->contain([
    'Categories' => function ($query) {
        return $query->find('translations');
    }
])->first();

// Affiche 'Programación'
echo $article->categories[0]->translation('es')->name;
```

Ceci implique que `Categories` a le TranslateBehavior attaché à celui-ci. Il utilise simplement la fonction de construction de requête pour la clause `contain` d'utiliser les `translations` du finder personnalisé dans l'association.

Récupérer une Langue sans Utiliser I18n : :setLocale

Appeler `I18n::setLocale('es');` change la locale par défaut pour tous les finds traduits, il peut y avoir des fois où vous souhaitez récupérer du contenu traduit sans modification de l'état de l'application. Pour ces scenarios, utilisez la méthode `setLocale()` du behavior :

```
I18n::setLocale('en'); // réinitialisation pour l'exemple

$this->loadModel('Articles');
// locale spécifique. Avant CakePHP 3.6 utilisez locale().
$this->Articles->setLocale('es');

$article = $this->Articles->get(12);
echo $article->title; // Echoes 'Un Artículo', yay piece of cake!
```

Notez que ceci va seulement changer la locale de la table Articles, cela ne changera pas la langue des données associées. Pour utiliser cette technique pour changer les données associées, il est nécessaire d'appeler la locale pour chaque table par exemple :

```
I18n::setLocale('en'); // reset for illustration

$this->loadModel('Articles');
// Avant CakePHP 3.6 utilisez locale().
$this->Articles->setLocale('es');
$this->Articles->Categories->setLocale('es');

$data = $this->Articles->find('all', ['contain' => ['Categories']]);
```

Cet exemple suppose que `Categories` a le TranslateBehavior attaché.

Faire une requête sur un champ traduit

Par défaut, le `TranslateBehavior` ne remplace rien dans les conditions des `find`. Vous devez utiliser la méthode `translationField()` pour composer des `find` basés sur des champs traduits :

```
// Avant CakePHP 3.6 utilisez locale().
$this->Articles->setLocale('es');
$data = $this->Articles->find()->where([
    $this->Articles->translationField('title') => 'Otro Título'
]);
```

Sauvegarder dans une Autre Langue

La philosophie derrière le TranslateBehavior est que vous avez une entity représentant la langue par défaut, et plusieurs traductions qui peuvent surcharger certains champs dans de telles entities. Garder ceci à l'esprit, vous pouvez sauvegarder de façon intuitive les traductions pour une entity donnée. Par exemple, étant donné la configuration suivante :

```
// dans src/Model/Table/ArticlesTable.php
class ArticlesTable extends Table
{
    public function initialize(array $config): void
    {
        $this->addBehavior('Translate', ['fields' => ['title', 'body']]);
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
}

// dans src/Model/Entity/Article.php
class Article extends Entity
{
    use TranslateTrait;
}

// Dans un controller
$this->loadModel('Articles');
$article = new Article([
    'title' => 'My First Article',
    'body' => 'This is the content',
    'footnote' => 'Some afterwords'
]);

$this->Articles->save($article);
```

Donc, après avoir sauvegardé votre premier article, vous pouvez maintenant sauvegarder une traduction pour celui-ci. Il y a quelques façons de le faire. La première est de configurer la langue directement dans une entity :

```
$article->_locale = 'es';
$article->title = 'Mi primer Artículo';

$this->Articles->save($article);
```

Après que l'entity a été sauvegardé, le champ traduit va aussi être persistent, une chose à noter est que les valeurs qui étaient par défaut surchargées à partir de la langue, seront préservées :

```
// Affiche 'This is the content'
echo $article->body;

// Affiche 'Mi primer Artículo'
echo $article->title;
```

Une fois que vous surchargez la valeur, la traduction pour ce champ sera sauvegardée et récupérée comme d'habitude :

```
$article->body = 'El contendio';
$this->Articles->save($article);
```

La deuxième manière de l'utiliser pour sauvegarder les entities dans une autre langue est de définir par défaut la langue directement à la table :

```
$article->title = 'Mi Primer Artículo';

// Avant CakePHP 3.6 utilisez locale().
$this->Articles->setLocale('es');
$this->Articles->save($article);
```

Configurer la langue directement dans la table est utile quand vous avez besoin à la fois de récupérer et de sauvegarder les entities pour la même langue ou quand vous avez besoin de sauvegarder plusieurs entities en une fois.

Sauvegarder Plusieurs Traductions

C'est un prérequis habituel d'être capable d'ajouter ou de modifier plusieurs traductions à l'enregistrement de la base de données au même moment. Ceci peut être fait en utilisant `TranslateTrait` :

```
use Cake\ORM\Behavior\Translate\TranslateTrait;
use Cake\ORM\Entity;

class Article extends Entity
{
    use TranslateTrait;
}
```

Maintenant vous pouvez ajouter les translations avant de les sauvegarder :

```
$translations = [
    'fr' => ['title' => "Un article"],
    'es' => ['title' => 'Un artículo']
];

foreach ($translations as $lang => $data) {
    $article->translation($lang)->set($data, ['guard' => false]);
}

$this->Articles->save($article);
```

Et créer des inputs de formulaire pour vos champs traduits :

```
// Dans un template de vue.
<?= $this->Form->create($article); ?>
<fieldset>
    <legend>French</legend>
    <?= $this->Form->control('_translations.fr.title'); ?>
    <?= $this->Form->control('_translations.fr.body'); ?>
</fieldset>
<fieldset>
    <legend>Spanish</legend>
    <?= $this->Form->control('_translations.es.title'); ?>
    <?= $this->Form->control('_translations.es.body'); ?>
</fieldset>
```

Dans votre controller, vous pouvez marshal les données comme d'habitude, mais avec l'option `translations` activée :

```
$article = $this->Articles->newEntity($this->request->data, [
    'translations' => true
]);
$this->Articles->save($article);
```

Ceci va faire que votre article, les traductions françaises et espagnoles vont tous persister. Vous devrez aussi vous souvenir d'ajouter `_translations` dans les champs accessibles `$_accessible` de votre entity.

Valider les Entities Traduites

Quand vous attachez `TranslateBehavior` à un model, vous pouvez définir le validateur qui doit être utilisé quand les enregistrements de traduction sont créés/mis à jours par le behavior pendant `newEntity()` ou `patchEntity()` :

```
class ArticlesTable extends Table
{
    public function initialize(array $config): void
    {
        $this->addBehavior('Translate', [
            'fields' => ['title'],
            'validator' => 'translated'
        ]);
    }
}
```

Ce qui est au-dessus va utiliser le validateur créé par les entities traduites validées `validationTranslated`.

Tree

class `Cake\ORM\Behavior\`**`TreeBehavior`**

Il est courant de vouloir stocker des données hiérarchisées dans une table de base de données. Des exemples de ce type de données pourrait être des catégories sans limite de sous-catégories, les données liées à un système de menu multi-niveau ou une représentation littérale de la hiérarchie comme un département dans une entreprise.

Les bases de données relationnelles ne sont couramment pas utilisées pour le stockage et la récupération de ce type de données, mais il y a quelques techniques connues qui les rendent possible pour fonctionner avec une information multi-niveau.

Le TreeBehavior vous aide à maintenir une structure de données hiérarchisée dans la base de données qui peut être requêtée facilement et aide à reconstruire les données en arbre pour trouver et afficher les processus.

Pré-Requis

Ce behavior nécessite que les colonnes suivantes soient présentes dans votre table :

- `parent_id` (nullable) La colonne contenant l'ID de la ligne parente
- `lft` (integer, signed) Utilisé pour maintenir la structure en arbre
- `rght` (integer, signed) Utilisé pour maintenir la structure en arbre

Vous pouvez configurer le nom de ces champs. Plus d'informations sur la signification des champs et comment ils sont utilisés peuvent être trouvées dans cet article décrivant la MPTT logic [136]

> **Avertissement**
>
> TreeBehavior ne supporte pas les clés primaires composites pour le moment.

Un Aperçu Rapide

Vous activez le behavior Tree en l'ajoutant à la Table où vous voulez stocker les données hiérarchisées dans :

```
class CategoriesTable extends Table
{
```

(suite sur la page suivante)

136. https://www.sitepoint.com/hierarchical-data-database-2/

(suite de la page précédente)

```
    public function initialize(array $config): void
    {
        $this->addBehavior('Tree');
    }
}
```

Une fois ajoutées, vous pouvez laisser CakePHP construire la structure interne si la table contient déjà quelques lignes :

```
// Prior to 3.6 use TableRegistry::get('Categories')
$categories = TableRegistry::getTableLocator()->get('Categories');
$categories->recover();
```

Vous pouvez vérifier que cela fonctionne en récupérant toute ligne de la table et en demandant le nombre de descendants qu'il a :

```
$node = $categories->get(1);
echo $categories->childCount($node);
```

Pour obtenir une liste à plat des descendants pour un nœud :

```
$descendants = $categories->find('children', ['for' => 1]);

foreach ($descendants as $category) {
    echo $category->name . "\n";
}
```

Si vous souhaitez uniquement les enfants directs du niveau en dessous

```
$directDescendants = $categories->find('children', ['for' => 1, 'direct' => true]);

foreach ($directDescendants as $category) {
    echo $category->name . "\n";
}
```

vous n'obtiendrez ainsi que les enfants du niveau n-1 et pas ceux des niveaux n-2,n-3 … etc …

Si vous avez besoin de passer des conditions, vous pouvez le faire comme avec n'importe quelle requête :

```
$descendants = $categories
    ->find('children', ['for' => 1])
    ->where(['name LIKE' => '%Foo%']);

foreach ($descendants as $category) {
    echo $category->name . "\n";
}
```

Si à la place, vous avez besoin d'une liste liée, où les enfants pour chaque nœud sont imbriqués dans une hiérarchie, vous pouvez utiliser le finder "threaded" :

```
$children = $categories
    ->find('children', ['for' => 1])
    ->find('threaded')
    ->toArray();
```

(suite sur la page suivante)

(suite de la page précédente)

```
foreach ($children as $child) {
    echo "{$child->name} has " . count($child->children) . " direct children";
}
```

Traverser les résultats threaded nécessitent habituellement des fonctions récursives, mais si vous avez besoin seulement d'un ensemble de résultats contenant un champ unique à partir de chaque niveau pour afficher une liste, dans un select HTML par exemple, il est préférable d'utiliser le finder "treeList" :

```
$list = $categories->find('treeList');

// Dans un fichier template de CakePHP:
echo $this->Form->control('categories', ['options' => $list]);

// Ou vous pouvez l'afficher en texte, par exemple dans un script de CLI
foreach ($list as $categoryName) {
    echo $categoryName . "\n";
}
```

La sortie sera similaire à ceci :

```
My Categories
_Fun
__Sport
___Surfing
___Skating
_Trips
__National
__International
```

Le finder `treeList` accepte un certain nombre d'options :

- `keyPath` : Le chemin séparé par des points pour récupérer le champ à utiliser en clé de tableau, ou une closure qui retourne la clé de la ligne fournie.
- `valuePath` : Le chemin séparé par des points pour récupérer le champ à utiliser en valeur de tableau, ou une closure qui retourne la valeur de la ligne fournie.
- `spacer` : Une chaîne de caractères utilisée en tant que préfixe pour désigner la profondeur dans l'arbre pour chaque item.

Un exemple d'utilisation de toutes les options serait :

```
$query = $categories->find('treeList', [
    'keyPath' => 'url',
    'valuePath' => 'id',
    'spacer' => ' '
]);
```

Une tâche classique est de trouver le chemin de l'arbre à partir d'un nœud en particulier vers la racine de l'arbre. C'est utile, par exemple, pour ajouter la liste des breadcrumbs pour une structure de menu :

```
$nodeId = 5;
$crumbs = $categories->find('path', ['for' => $nodeId]);

foreach ($crumbs as $crumb) {
    echo $crumb->name . ' > ';
}
```

Les arbres construits avec TreeBehavior ne peuvent pas être triés avec d'autres colonnes que `lft`, ceci parce que la représentation interne de l'arbre dépend de ce tri. Heureusement, vous pouvez réorganiser les nœuds à l'intérieur du même niveau dans avoir à changer leur parent :

```
$node = $categories->get(5);

// Déplace le nœud pour qu'il monte d'une position quand on liste les enfants.
$categories->moveUp($node);

// Déplace le nœud vers le haut de la liste dans le même niveau.
$categories->moveUp($node, true);

// Déplace le nœud vers le bas.
$categories->moveDown($node, true);
```

Configuration

Si les noms de colonne par défaut qui sont utilisés par ce behavior ne correspondent pas à votre schéma, vous pouvez leur fournir des alias :

```
public function initialize(array $config): void
{
    $this->addBehavior('Tree', [
        'parent' => 'ancestor_id', // Utilise ceci plutôt que parent_id,
        'left' => 'tree_left', // Utilise ceci plutôt que lft
        'right' => 'tree_right' // Utilise ceci plutôt que rght
    ]);
}
```

Niveau des Nœuds (profondeur)

Connaître la profondeur d'une structure arbre peut être utile lorsque vous voulez récupérer des nœuds jusqu'à un certain niveau uniquement par exemple lorsque pour générer un menu. Vous pouvez utiliser l'option `level` pour spécifier les champs qui sauvegarderont la profondeur de chaque nœud :

```
$this->addBehavior('Tree', [
    'level' => 'level', // Defaults to null, i.e. no level saving
]);
```

Si vous ne souhaitez pas mettre en cache le niveau en utilisant un champ de base de données, vous pouvez utiliser la méthode `TreeBehavior::getLevel()` pour connaître le niveau d'un nœuds.

Scoping et Arbres Multiples

Parfois vous voulez avoir plus d'une structure d'arbre dans la même table, vous pouvez arriver à faire ceci en utilisant la configuration "scope". Par exemple, dans une table locations vous voudrez créer un arbre par pays :

```
class LocationsTable extends Table
{

    public function initialize(array $config): void
    {
        $this->addBehavior('Tree', [
```

(suite sur la page suivante)

(suite de la page précédente)

```
            'scope' => ['country_name' => 'Brazil']
        ]);
    }

}
```

Dans l'exemple précédent, toutes les opérations sur l'arbre seront scoped seulement pour les lignes ayant la colonne `country_name` défini à "Brazil". Vous pouvez changer le scoping à la volée en utilisant la fonction "config" :

```
$this->behaviors()->Tree->config('scope', ['country_name' => 'France']);
```

En option, vous pouvez avoir un contrôle plus fin du scope en passant une closure au scope :

```
$this->behaviors()->Tree->config('scope', function ($query) {
    $country = $this->getConfigureContry(); // A made-up function
    return $query->where(['country_name' => $country]);
});
```

Récupération avec un Tri Personnalisé du Champ

Par défaut, recover() trie les items en utilisant la clé primaire. Ceci fonctionne bien s'il s'agit d'une colonne numérique (avec incrémentation auto), mais cela peut entraîner des résultats étranges si vous utilisez les UUIDs.

Si vous avez besoin de tri personnalisé pour la récupération, vous pouvez définir une clause order personnalisée dans votre config :

```
$this->addBehavior('Tree', [
    'recoverOrder' => ['country_name' => 'DESC'],
]);
```

Sauvegarder les Données Hiérarchisées

Quand vous utilisez le behavior Tree, vous n'avez habituellement pas besoin de vous soucier de la représentation interne de la structure hiérarchisée. Les positions où les nœuds sont placés dans l'arbre se déduisent de la colonne "parent_id" dans chacune de vos entities :

```
$aCategory = $categoriesTable->get(10);
$aCategory->parent_id = 5;
$categoriesTable->save($aCategory);
```

Fournir des ids de parent non existant lors de la sauvegarde ou tenter de créer une boucle dans l'arbre (faire un nœud enfant de lui-même) va lancer une exception.

Vous pouvez faire un nœud à la racine de l'arbre en configurant la colonne "parent_id" à null :

```
$aCategory = $categoriesTable->get(10);
$aCategory->parent_id = null;
$categoriesTable->save($aCategory);
```

Les enfants pour un nouveau nœud à la racine seront préservés.

Supprimer les Nœuds

Supprimer un nœud et tout son sous-arbre (tout enfant qu'il peut avoir à tout niveau dans l'arbre) est facile :

```
$aCategory = $categoriesTable->get(10);
$categoriesTable->delete($aCategory);
```

TreeBehavior va s'occuper de toutes les opérations internes de suppression. Il est aussi possible de supprimer seulement un nœud et de réassigner tous les enfants au nœud parent immédiatement supérieur dans l'arbre :

```
$aCategory = $categoriesTable->get(10);
$categoriesTable->removeFromTree($aCategory);
$categoriesTable->delete($aCategory);
```

Tous les nœuds enfant seront conservés et un nouveau parent leur sera assigné.

La suppression d'un noeud est basée sur les valeurs lft et rght de l'entity. C'est important de le noter quand on fait une boucle des différents enfants d'un noeud pour des suppressions conditionnelles :

```
$descendants = $teams->find('children', ['for' => 1]);

foreach ($descendants as $descendant) {
    $team = $teams->get($descendant->id); // cherche l'objet entity mis à jour
    if ($team->expired) {
        $teams->delete($team); // la suppression re-trie les entrées lft et rght de la␣
↪base de données
    }
}
```

TreeBehavior re-trie les valeurs lft et rght des enregistrements de la table quand un noeud est supprimé. Telles quelles, les valeurs lft et rght des entities dans `$descendants` (sauvegardées avant l'opération de suppression) seront erronées. Les entities devront être chargées et modifiées à la volée pour éviter les incohérences dans la table.

Créer un Behavior

Dans les exemples suivants, nous allons créer un `SluggableBehavior` très simple. Ce behavior va nous autoriser à remplir un champ slug avec les résultats de `Text::slug()` basé sur un autre champ.

Avant de créer notre behavior, nous devrions comprendre les conventions pour les behaviors :

- Les fichiers Behavior sont localisés dans **src/Model/Behavior**, ou dans `MyPlugin\Model\Behavior`.
- Les classes de Behavior devraient être dans le namespace `App\Model\Behavior`, ou le namespace `MyPlugin\Model\Behavior`.
- Les noms de classe de Behavior finissent par `Behavior`.
- Les Behaviors étendent `Cake\ORM\Behavior`.

Pour créer notre behavior sluggable. Mettez ce qui suit dans **src/Model/Behavior/SluggableBehavior.php** :

```
namespace App\Model\Behavior;

use Cake\ORM\Behavior;

class SluggableBehavior extends Behavior
{
}
```

Comme les tables, les behaviors ont également un hook `initialize()` où vous pouvez mettre le code d'initialisation, si nécessaire :

```
public function initialize(array $config): void
{
    // Code d'initialisation ici
}
```

Nous pouvons maintenant ajouter ce behavior à l'une de nos classes de table. Dans cet exemple, nous allons utiliser un `ArticlesTable`, puisque les articles ont souvent des propriétés slug pour créer de belles URLs :

```
namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{

    public function initialize(array $config): void
    {
        $this->addBehavior('Sluggable');
    }
}
```

Notre nouveau behavior ne fait pas beaucoup plus pour le moment. Ensuite, nous allons ajouter une méthode mixin et un event listener pour que lorsque nous sauvegarderons les entities, nous puissions automatiquement slugger un champ.

Définir les Méthodes Mixin

Toute méthode public définie sur un behavior sera ajoutée en méthode "mixin" sur l'objet table sur laquelle il est attaché. Si vous attachez deux behaviors qui fournissent les mêmes méthodes, une exception sera levée. Si un behavior fournit la même méthode en classe de table, la méthode du behavior ne sera pas appelable à partir de la table. Les méthodes mixin de Behavior vont recevoir exactement les mêmes arguments qui sont fournis à la table. Par exemple, si notre SluggableBehavior définit la méthode suivante :

```
public function slug($value)
{
    return Text::slug($value, $this->_config['replacement']);
}
```

Il pourrait être invoqué de la façon suivante :

```
$slug = $articles->slug('My article name');
```

Limiter ou renommer les Méthodes Mixin Exposed

Lors de la création de behaviors, il peut y avoir des situations où vous ne voulez pas montrer les méthodes public en méthodes mixin. Dans ces cas, vous pouvez utiliser la clé de configuration `implementedMethods` pour renommer ou exclure les méthodes mixin. Par exemple si nous voulions préfixer notre méthode slug(), nous pourrions faire ce qui suit :

```
protected $_defaultConfig = [
    'implementedMethods' => [
        'superSlug' => 'slug',
    ]
];
```

Appliquer cette configuration rendra votre `slug()` non appelable, cependant elle va ajouter une méthode mixin `superSlug()` à la table. Cependant, si notre behavior implémentait d'autres méthodes public, elles **n'auraient** pas été disponibles en méthodes mixin avec la configuration ci-dessus.

Alors que les méthodes montrées sont définies par configuration, vous pouvez aussi renommer/retirer les méthodes mixin lors de l'ajout d'un behavior à la table. Par exemple :

```
// Dans une méthode initialize() de la table.
$this->addBehavior('Sluggable', [
    'implementedMethods' => [
        'superSlug' => 'slug',
    ]
]);
```

Définir des Event Listeners

Maintenant que notre behavior a une méthode mixin pour slugger les champs, nous pouvons implémenter un listener de callback pour slugger automatiquement un champ quand les entities sont sauvegardées. Nous allons aussi modifier notre méthode slug pour accepter une entity plutôt que juste une valeur plain. Notre behavior devrait maintenant ressembler à ceci :

```
namespace App\Model\Behavior;

use ArrayObject;
use Cake\Datasource\EntityInterface;
use Cake\Event\Event;
use Cake\ORM\Behavior;
use Cake\ORM\Entity;
use Cake\ORM\Query;
use Cake\Utility\Text;

class SluggableBehavior extends Behavior
{
    protected $_defaultConfig = [
        'field' => 'title',
        'slug' => 'slug',
        'replacement' => '-',
    ];

    public function slug(Entity $entity)
    {
        $config = $this->config();
        $value = $entity->get($config['field']);
        $entity->set($config['slug'], Text::slug($value, $config['replacement']));
    }

    public function beforeSave(Event $event, EntityInterface $entity, ArrayObject
↪$options)
    {
        $this->slug($entity);
    }

}
```

Le code ci-dessus montre quelques fonctionnalités intéressantes des behaviors :

- Les Behaviors peuvent définir des méthodes callback en définissant des méthodes qui suivent les conventions des *Callbacks du Cycle de Vie*.
- Les Behaviors peuvent définir une propriété de configuration par défaut. Cette propriété est fusionnée avec les valeurs données lorsqu'un behavior est attaché à la table.

Pour empêcher l'enregistrement de continuer, arrêtez simplement la propagation de l'évènement dans votre callback :

```
public function beforeSave(Event $event, EntityInterface $entity, ArrayObject $options)
{
    if (...) {
        $event->stopPropagation();

        return;
    }
    $this->slug($entity);
}
```

Définir des Finders

Maintenant que nous sommes capable de sauvegarder les articles avec les valeurs de slug, nous allons implémenter une méthode de find afin de pouvoir récupérer les articles par leur slug. Les méthodes find de behavior utilisent les mêmes conventions que les *Méthodes Finder Personnalisées*. Notre méthode `find('slug')` ressemblerait à ceci :

```
public function findSlug(Query $query, array $options)
{
    return $query->where(['slug' => $options['slug']]);
}
```

Une fois que notre behavior a la méthode ci-dessus, nous pouvons l'appeler :

```
$article = $articles->find('slug', ['slug' => $value])->first();
```

Limiter ou renommer les Méthodes de Finder Exposed

Lors de la création de behaviors, il peut y avoir des situations où vous ne voulez pas montrer les méthodes find, ou vous avez besoin de renommer les finders pour éviter les méthodes dupliquées. Dans ces cas, vous pouvez utiliser la clé de configuration `implementedFinders` pour renommer ou exclure les méthodes find. Par exemple, si vous vouliez renommer votre méthode `find(slug)`, vous pourriez faire ce qui suit :

```
protected $_defaultConfig = [
    'implementedFinders' => [
        'slugged' => 'findSlug',
    ]
];
```

Utiliser cette configuration fera que `find('slug')` attrapera une erreur. Cependant, cela rendra `find('slugged')` disponible. Notamment si notre behavior implémente d'autres méthodes find, elles **ne** seront pas disponibles puisqu'elles ne sont pas inclues dans la configuration.

Depuis que les méthodes montrées sont décidées par configuration, vous pouvez aussi renommer/retirer les méthodes find lors de l'ajout d'un behavior à la table. Par exemple :

```
// Dans la méthode initialize() de la table.
$this->addBehavior('Sluggable', [
```

(suite sur la page suivante)

(suite de la page précédente)

```
    'implementedFinders' => [
        'slugged' => 'findSlug',
    ]
]);
```

Transformer les Données de la Requête en Propriétés de l'Entity

Les Behaviors peuvent définir de la logique sur la façon dont les champs personnalisés qu'ils fournissent sont marshalled en implémentant `Cake\ORM\PropertyMarshalInterface`. Cette interface nécessite une méthode unique à implémenter :

```
public function buildMarshalMap($marshaller, $map, $options)
{
    return [
        'custom_behavior_field' => function ($value, $entity) {
            // Transform the value as necessary
            return $value . '123';
        }
    ];
}
```

`TranslateBehavior` a une implémentation non banal de cette interface que vous pouvez aller consulter.

Retirer les Behaviors Chargés

Pour retirer un behavior de votre table, vous pouvez appeler la méthode `removeBehavior()` :

```
// Retire le behavior chargé
$this->removeBehavior('Sluggable');
```

Accéder aux Behaviors Chargés

Une fois que vous avez attaché les behaviors à votre instance de Table, vous pouvez interroger les behaviors chargés ou accéder à des behaviors spécifiques en utilisant le `BehaviorRegistry` :

```
// Regarde les behaviors qui sont chargés
$table->behaviors()->loaded();

// Vérifie si un behavior spécifique est chargé.
// N'utilisez pas les préfixes de plugin.
$table->behaviors()->has('CounterCache');

// Récupère un behavior chargé
// N'utilisez pas les préfixes de plugin.
$table->behaviors()->get('CounterCache');
```

Re-configurer les Behaviors Chargés

Pour modifier la configuration d'un behavior déjà chargé, vous pouvez combiner la commande `BehaviorRegistry::get` avec la commande `config` fournie par le trait `InstanceConfigTrait`.

Par exemple si une classe parente (par ex `AppTable`) charge le behavior `Timestamp`, vous pouvez faire ce qui suit pour ajouter, modifier ou retirer les configurations pour le behavior. Dans ce cas, nous ajouterons un event pour lequel nous souhaitons que Timestamp réponde :

```
namespace App\Model\Table;

use App\Model\Table\AppTable; // similar to AppController

class UsersTable extends AppTable
{
    public function initialize(array $options): void
    {
        parent::initialize($options);

        // par ex si notre parent appelle $this->addBehavior('Timestamp');
        // et que nous souhaitons ajouter un event supplémentaire
        if ($this->behaviors()->has('Timestamp')) {
            $this->behaviors()->get('Timestamp')->config([
                'events' => [
                    'Users.login' => [
                        'last_login' => 'always'
                    ],
                ],
            ]);
        }
    }
}
```

Système de Schéma

CakePHP dispose d'un système de schéma qui est capable de montrer et de générer les informations de schéma des tables dans les stockages de données SQL. Le système de schéma peut générer/montrer un schéma pour toute plateforme SQL que CakePHP supporte.

Les principales parties du système de schéma sont `Cake\Database\Schema\Collection` et `Cake\Database\Schema\TableSchema`. Ces classes vous donnent accès respectivement à la base de donnée toute entière et aux fonctionnalités de l'objet TableSchema.

L'utilisation première du système de schéma est pour les *Fixtures*. Cependant, il peut aussi être utilisé dans votre application si nécessaire.

Objets Schema\TableSchema

class `Cake\Database\Schema\`**`TableSchema`**

Le sous-système de schéma fournit un objet TableSchema pour récupérer les données d'une table dans la base de données. Cet objet est retourné par les fonctionnalités de réflection de schéma :

```
use Cake\Database\Schema\TableSchema;

// Crée une table colonne par colonne.
$schema = new TableSchema('posts');
$schema->addColumn('id', [
  'type' => 'integer',
  'length' => 11,
  'null' => false,
  'default' => null,
])->addColumn('title', [
```

(suite sur la page suivante)

(suite de la page précédente)

```
  'type' => 'string',
  'length' => 255,
  // Create a fixed length (char field)
  'fixed' => true
])->addConstraint('primary', [
  'type' => 'primary',
  'columns' => ['id']
]);

// Les classes Schema\TableSchema peuvent aussi être créées avec des données de tableau
$schema = new TableSchema('posts', $columns);
```

Les objets `Schema\TableSchema` vous permettent de construire des informations sur le schéma d'une table. Il aide à normaliser et à valider les données utilisées pour décrire une table. Par exemple, les deux formulaires suivants sont équivalents :

```
$schema->addColumn('title', 'string');
// et
$schema->addColumn('title', [
  'type' => 'string'
]);
```

Bien qu'équivalent, le 2ème formulaire donne plus de détails et de contrôle. Ceci émule les fonctionnalités existantes disponibles dans les fichiers de Schéma + le schéma de fixture dans 2.x.

Accéder aux Données de Colonne

Les colonnes sont soit ajoutées en argument du constructeur, soit via *addColumn()*. Une fois que les champs sont ajoutés, les informations peuvent être récupérées en utilisant *column()* ou *columns()* :

```
// Récupère le tableau de données d'une colonne
$c = $schema->column('title');

// Récupère la liste de toutes les colonnes.
$cols = $schema->columns();
```

Index et Contraintes

Les index sont ajoutés en utilisant `addIndex()`. Les contraintes sont ajoutées en utilisant `addConstraint()`. Les index et contraintes ne peuvent pas être ajoutés pour les colonnes qui n'existent pas puisque cela donnerait un état invalide. Les index sont différents des contraintes et des exceptions seront levées si vous essayez de mélanger les types entre les méthodes. Un exemple des deux méthodes est :

```
$schema = new TableSchema('posts');
$schema->addColumn('id', 'integer')
  ->addColumn('author_id', 'integer')
  ->addColumn('title', 'string')
  ->addColumn('slug', 'string');

// Ajoute une clé primaire.
$schema->addConstraint('primary', [
  'type' => 'primary',
```

(suite sur la page suivante)

(suite de la page précédente)

```
  'columns' => ['id']
]);
// Ajoute une clé unique
$schema->addConstraint('slug_idx', [
  'columns' => ['slug'],
  'type' => 'unique',
]);
// Ajoute un indice
$schema->addIndex('slug_title', [
  'columns' => ['slug', 'title'],
  'type' => 'index'
]);
// Ajoute une clé étrangère
$schema->addConstraint('author_id_idx', [
  'columns' => ['author_id'],
  'type' => 'foreign',
  'references' => ['authors', 'id'],
  'update' => 'cascade',
  'delete' => 'cascade'
]);
```

Si vous ajoutez une contrainte de clé primaire à une colonne unique integer, elle va automatiquement être convertie en une colonne auto-incrémentée/série selon la plateforme de la base de données :

```
$schema = new TableSchema('posts');
$schema->addColumn('id', 'integer')
->addConstraint('primary', [
    'type' => 'primary',
    'columns' => ['id']
]);
```

Dans l'exemple ci-dessus, la colonne `id` générerait le SQL suivant dans MySQL :

```
CREATE TABLE `posts` (
    `id` INTEGER AUTO_INCREMENT,
    PRIMARY KEY (`id`)
)
```

Si votre clé primaire contient plus d'une colonne, aucune d'elle ne sera automatiquement convertie en une valeur auto-incrémentée. A la place, vous devrez dire à l'objet table quelle colonne dans la clé composite vous voulez auto-incrémenter :

```
$schema = new TableSchema('posts');
$schema->addColumn('id', [
    'type' => 'integer',
    'autoIncrement' => true,
])
->addColumn('account_id', 'integer')
->addConstraint('primary', [
    'type' => 'primary',
    'columns' => ['id', 'account_id']
]);
```

L'option `autoIncrement` ne fonctionne qu'avec les colonnes `integer` et `biginteger`.

Lire les Index et les Contraintes

Les index et les contraintes peuvent être lus d'un objet table en utilisant les méthodes d'accesseur. En supposant que `$schema` est une instance de table remplie, vous pourriez faire ce qui suit :

```
// Récupère les contraintes. Va retourner les noms de toutes les
// contraintes.
$constraints = $schema->constraints()

// Récupère les données sur une contrainte unique.
$constraint = $schema->constraint('author_id_idx')

// Récupère les index. Va retourner les noms de tous les index
$indexes = $schema->indexes()

// Récupère les données d'un index unique.
$index = $schema->index('author_id_idx')
```

Ajouter des Options de Table

Certains drivers (principalement MySQL) supportent et nécessitent des meta données de table supplémentaires. Dans le cas de MySQL, les propriétés `CHARSET`, `COLLATE` et `ENGINE` sont nécessaires pour maintenir une structure de table dans MySQL. Ce qui suit pourra être utilisé pour ajouter des options de table :

```
$schema->options([
  'engine' => 'InnoDB',
  'collate' => 'utf8_unicode_ci',
]);
```

Les languages de plateforme ne gèrent que les clés qui les intéressent et ignorent le reste. Toutes les options ne sont pas supportées sur toutes les plateformes.

Convertir les Tables en SQL

En utilisant `createSql()` ou `dropSql()` vous pouvez récupérer du SQL spécifique à la plateforme pour créer ou supprimer une table spécifique :

```
$db = ConnectionManager::get('default');
$schema = new TableSchema('posts', $fields, $indexes);

// Crée une table
$queries = $schema->createSql($db);
foreach ($queries as $sql) {
  $db->execute($sql);
}

// Supprime une table
$sql = $schema->dropSql($db);
$db->execute($sql);
```

En utilisant un driver de connection, les données de schéma peuvent être converties en SQL spécifique à la plateforme. Le retour de `createSql` et `dropSql` est une liste de requêtes SQL nécessaires pour créer une table et les index né-

cessaires. Certaines plateformes peuvent nécessiter plusieurs lignes pour créer des tables avec des commentaires et/ou index. Un tableau de requêtes est toujours retourné.

Collections de Schéma

class Cake\Database\Schema**Collection**

`Collection` fournit un accès aux différentes tables disponibles pour une connection. Vous pouvez l'utiliser pour récupérer une liste des tables ou envoyer les tables dans les objets *TableSchema*. Une utilisation habituelle de la classe ressemble à :

```
$db = ConnectionManager::get('default');

// Crée une collection de schéma.
// Prior to 3.4 use $db->schemaCollection()
$collection = $db->getSchemaCollection();

// Récupère les noms des tables
$schemaables = $collection->listTables();

// Récupère une table unique (instance de Schema\TableSchema)
$schemaable = $collection->describe('posts');
```

Shell du Cache du Schéma

SchemaCacheShell fournit un outil CLI simple pour gérer les caches de metadata de votre application. Dans les situations de déploiement, il est utile de reconstruire le cache des metadata déjà en place sans enlever les données du cache existantes. Vous pouvez faire ceci en lançant :

```
bin/cake schema_cache build --connection default
```

Ceci va reconstruire le cache de metadata pour toutes les tables sur la connection `default`. Si vous avez besoin seulement de reconstruire une table unique, vous pouvez faire ceci en fournissant son nom :

```
bin/cake schema_cache build --connection default articles
```

En plus de construire les données mises en cache, vous pouvez utiliser aussi SchemaCacheShell pour retirer les metadata mis en cache :

```
# Nettoyer toutes les metadata
bin/cake schema_cache clear

# Nettoyer une table unique
bin/cake schema_cache clear articles
```

> **Note**
>
> Avant 3.6, vous devez utiliser `orm_cache` à la place de `schema_cache`.

CHAPITRE 13

La mise en cache

class Cake\Cache**Cache**

La mise en cache est fréquemment utilisée pour réduire le temps pris pour créer ou lire depuis une autre ressource. La mise en cache est souvent utilisée pour rendre la lecture de ressources consommatrices en temps, en ressources moins consommatrice. Vous pouvez aisément stocker en cache les résultats de requêtes consommatrices en ressources ou les accès à distance à des services web qui ne changent pas fréquemment. Une fois mis en cache, re-lire les ressources stockées depuis le cache est moins consommateur en ressource qu'un accès à une ressource distante.

La mise en cache dans CakePHP se fait principalement par la classe `Cache`. Cette classe fournit un ensemble de méthodes statiques qui fournissent une API uniforme pour le traitement des différentes implémentations de mise en cache. CakePHP dispose de plusieurs moteurs de cache intégrés, et fournit un système facile pour implémenter votre propre système de mise en cache. Les moteurs de cache intégrés sont :

— `FileCache` File cache est un cache simple qui utilise des fichiers locaux. C'est le moteur de cache le plus lent, et il ne fournit que peu de fonctionnalités pour les opérations atomiques. Cependant, le stockage sur disque est souvent peu consommateur en ressource, le stockage de grands objets ou des éléments qui sont rarement écrits fonctionne bien dans les fichiers.
— `ApcCache` Le cache APC utilise l'extension PHP APCu [137]. Cette extension utilise la mémoire partagée du serveur Web pour stocker les objets. Cela le rend très rapide, et capable de fournir les fonctionnalités atomiques en lecture/écriture.
— `Wincache` Utilise l'extension Wincache [138]. Wincache offre des fonctionnalités et des performances semblables à APC, mais optimisées pour Windows et IIS.
— `XcacheEngine` Xcache [139]. est une extension PHP qui fournit des fonctionnalités similaires à APC.
— `MemcachedEngine` Utilise l'extension Memcached [140].
— `RedisEngine` Utilise l'extension phpredis [141]. Redis fournit un système de cache cohérent et rapide similaire à Memcached et il permet aussi les opérations atomiques.

137. https://php.net/apcu
138. https://php.net/wincache
139. https://en.wikipedia.org/wiki/List_of_PHP_accelerators#XCache
140. https://php.net/memcached
141. https://github.com/phpredis/phpredis

Quelque soit le moteur de cache que vous choisirez d'utiliser, votre application interagit avec `Cake\Cache\Cache` de manière cohérente. Cela signifie que vous pouvez aisément permuter les moteurs de cache en fonction de l'évolution de votre application.

Configuration de la classe Cache

static Cake\Cache\Cache::**config**(*$key*, *$config = null*)

La configuration de la classe Cache peut être effectuée n'importe où, mais généralement vous voudrez configurer le cache pendant la phase de bootstrap. le fichier **config/app.php** est le lieu approprié pour cette configuration. Vous pouvez configurer autant de configurations de cache dont vous avez besoin, et vous pouvez utiliser tous les mélanges de moteurs de cache. CakePHP utilise deux configurations de cache en interne. `_cake_core_` est utilisé pour stocker des correspondances de fichiers, et les résultats parsés des fichiers de *traduction* . `_cake_model_` est utilisé pour stocker les schémas des models de vos applications. Si vous utilisez APC ou Memcached vous devrez vous assurer de définir des clés uniques pour les caches du noyau. Ceci vous évitera qu'une application vienne réécrire les données cache d'une autre application.

L'utilisation de plusieurs configurations vous permet également de changer le stockage comme vous l'entendez. Par exemple vous pouvez mettre ceci dans votre **config/app.php** :

```
// ...
'Cache' => [
    'short' => [
        'className' => 'File',
        'duration' => '+1 hours',
        'path' => CACHE,
        'prefix' => 'cake_short_'
    ],
    // Utilisation d'un espace de nom complet.
    'long' => [
        'className' => 'Cake\Cache\Engine\FileEngine',
        'duration' => '+1 week',
        'probability' => 100,
        'path' => CACHE . 'long' . DS,
    ]
]
// ...
```

Les options de configuration peuvent également être fournies en tant que chaine *DSN*. C'est utile lorsque vous travaillez avec des variables d'environnement ou des fournisseurs *PaaS* :

```
Cache::config('short', [
    'url' => 'memcached://user:password@cache-host/?timeout=3600&prefix=myapp_',
]);
```

Lorsque vous utilisez une chaine DSN, vous pouvez définir des paramètres/options supplémentaires en tant qu'arguments de query string.

Vous pouvez également configurer les moteurs de cache pendant l'exécution :

```
// Utilisation d'un nom court
Cache::config('short', [
    'className' => 'File',
    'duration' => '+1 hours',
```

(suite sur la page suivante)

(suite de la page précédente)

```
    'path' => CACHE,
    'prefix' => 'cake_short_'
]);

// Utilisation d'un espace de nom complet.
Cache::config('long', [
    'className' => 'Cake\Cache\Engine\FileEngine',
    'duration' => '+1 week',
    'probability' => 100,
    'path' => CACHE . 'long' . DS,
]);

// utilisation d'un objet.
$object = new FileEngine($config);
Cache::config('other', $object);
```

> **Note**
>
> Vous devez spécifier le moteur à utiliser. Il ne met **pas** File par défaut.

En insérant le code ci-dessus dans votre **config/app.php** vous aurez deux configurations de cache supplémentaires. Le nom de ces configurations "short" ou "long" est utilisé comme paramètre `$config` pour *`Cake\Cache\Cache::write()`* et *`Cake\Cache\Cache::read()`*. Lors de la configuration des moteurs de cache, vous pouvez vous référer au nom de la classe en utilisant les syntaxes suivantes :

- Un nom raccourci sans "Engine" ou namespace (espace de nom). Il déduira que que vous voulez utiliser `Cake\Cache\Engine` ou `App\Cache\Engine`.
- Utiliser la *syntaxe de plugin* qui permet de charger des moteurs depuis un plugin spécifique.
- Utiliser un nom de classe complet incluant le namespace. Cela vous permet d'utiliser des classes situées en dehors des emplacements classiques.
- Utiliser un objet qui étend la classe `CacheEngine`

> **Note**
>
> Lorsque vous utilisez le FileEngine vous pourriez avoir besoin d'utiliser l'option `mask` pour assurer que les fichiers de cache sont créés avec les autorisations nécessaires.

Configurer un Fallback de Cache

Dans le cas où un moteur de cache n'est pas disponible, comme par exemple le `FileEngine` essayant d'écrire dans un dossier sans les droits d'écriture ou le `RedisEngine` n'arrivant pas à se connecter à Redis, le moteur se repliera sur le moteur "noop" `NullEngine` et déclenchera une erreur qui sera loggée. Cela permet d'éviter que l'application lance une exception qui ne sera pas interceptée à cause d'une erreur de cache.

Vous pouvez configurer vos configurations de Cache pour se replier sur une configuration spécifique en utilisant la clé de configuration `fallback` :

```
Cache::config('redis', [
    'className' => 'Redis',
    'duration' => '+1 hours',
    'prefix' => 'cake_redis_',
    'host' => '127.0.0.1',
```

(suite sur la page suivante)

(suite de la page précédente)

```
    'port' => 6379,
    'fallback' => 'default',
]);
```

Si le serveur Redis tombait en erreur de manière inattendue, l'écriture dans le cache avec la configuration `redis` se repliera sur la configuration `default`. Si l'écriture dans la configuration `default` échouait *elle aussi*, le moteur se replierait à nouveau sur un autre "fallback", ici le `NullEngine`, et empêcherait l'application de lancer une exception.

Suppression de Configuration de Cache

static Cake\Cache\Cache::**drop**(*$key*)

Une fois la configuration créée, vous ne pouvez pas la changer. Au lieu de cela, vous devriez supprimer la configuration et la re-créer à l'aide de *Cake\Cache\Cache::drop()* et *Cake\Cache\Cache::config()*. Supprimer un moteur de cache va supprimer la configuration et détruire l'adaptateur s'il a été construit.

Ecrire dans un Cache

static Cake\Cache\Cache::**write**(*$key*, *$value*, *$config = 'default'*)

`Cache::write()` stocke $value dans le Cache. Vous pouvez lire ou supprimer cette valeur plus tard en vous y référant via `$key`. Vous pouvez spécifier une configuration optionnelle pour y stocker le cache. Si aucune `$config` n'est spécifiée, la configuration par défaut sera utilisée. `Cache::write()` peut stocker tout type d'objet et est idéale pour stocker les résultats des "finds" de vos models :

```
if (($posts = Cache::read('posts')) === false) {
    $posts = $unService->getAllPosts();
    Cache::write('posts', $posts);
}
```

Utiliser `Cache::write()` et `Cache::read()` réduira le nombre d'allers-retours effectués vers la base de données pour récupérer les messages.

> **Note**
>
> Si vous prévoyez de mettre en cache le résultat de requêtes faites avec l'ORM de CakePHP, il est préférable d'utiliser les fonctionnalités de cache intégrées dans l'objet Query, telles que décrites dans la section *mettre les résultats de requête en cache*

Ecrire Plusieurs Clés d'un Coup

static Cake\Cache\Cache::**writeMany**(*$data*, *$config = 'default'*)

Vous pouvez avoir besoin d'écrire plusieurs clés du cache à la fois. Bien que vous pouvez utiliser plusieurs appels à `write()`, `writeMany()` permet à CakePHP l'utilisation d'une API de stockage plus efficace quand cela est possible. Par exemple utiliser `writeMany()` permet de gagner de nombreuses connections réseau lors de l'utilisation de Memcached :

```
$result = Cache::writeMany([
    'article-' . $slug => $article,
    'article-' . $slug . '-comments' => $comments
```

(suite sur la page suivante)

(suite de la page précédente)

```
]);

// $result va contenir
['article-first-post' => true, 'article-first-post-comments' => true]
```

Lire un Cache Distribué

static Cake\Cache\Cache::**remember**(*$key*, *$callable*, *$config = 'default'*)

Cache permet la lecture d'un cache distribué. Si la clé de cache demandée existe, elle sera retournée. Si la clé n'existe pas, le callable sera invoqué et les résultats stockés dans le cache pour la clé fournie.

Par exemple, vous souhaitez souvent mettre en cache les résultats du appel à un service distant. Vous pouvez utiliser `remember()` pour faciliter cela :

```
class IssueService
{

    function allIssues($repo)
    {
        return Cache::remember($repo . '-issues', function () use ($repo) {
            return $this->fetchAll($repo);
        });
    }

}
```

Lire depuis un Cache

static Cake\Cache\Cache::**read**(*$key*, *$config = 'default'*)

`Cache::read()` est utilisée pour lire la valeur mise en cache stockée dans `$key` dans la `$config`. Si `$config` est null la configuration par défaut sera utilisée. `Cache::read()` renverra la valeur mise en cache si le cache est valide ou `false` si le cache a expiré ou n'existe pas. Le contenu du cache peut être mal évalué, donc assurez vous d'utiliser les opérateurs de comparaison stricts : `===` ou `!==`.

Par exemple :

```
$cloud = Cache::read('cloud');

if ($cloud !== false) {
    return $cloud;
}

// Genère des données cloud
// ...

// Stocke les données en cache
Cache::write('cloud', $cloud);

return $cloud;
```

Lire Plusieurs Clés d'un Coup

static Cake\Cache\Cache::**readMany**(*$keys*, *$config = 'default'*)

Après avoir écrit plusieurs clés d'un coup, vous voudrez probablement les lire également. Bien que vous pouvez utiliser plusieurs appels à `read()`, `readMany()``permet à CakePHP l'utilisation d'une API de stockage plus efficace quand cela est possible. Par exemple utiliser ``readMany()` permet de gagner de nombreuses connections réseau lors de l'utilisation de Memcached :

```
$result = Cache::readMany([
    'article-' . $slug,
    'article-' . $slug . '-comments'
]);
// $result contiendra
['article-first-post' => '...', 'article-first-post-comments' => '...']
```

Suppression d'un Cache

static Cake\Cache\Cache::**delete**(*$key*, *$config = 'default'*)

`Cache::delete()` vous permettra de supprimer complètement un objet mis en cache du stockage :

```
// Supprime la clé
Cache::delete('my_key');
```

Supprimer Plusieurs Clés d'un Coup

static Cake\Cache\Cache::**deleteMany**(*$keys*, *$config = 'default'*)

Après avoir écrit plusieurs clés d'un coup, vous voudrez probablement les supprimer également. Bien que vous pouvez utiliser plusieurs appels à `delete()`, `deleteMany()` permet à CakePHP l'utilisation d'une API de stockage plus efficace quand cela est possible. Par exemple utiliser `deleteMany()` permet de gagner de nombreuses connections réseau lors de l'utilisation de Memcached :

```
$result = Cache::deleteMany([
    'article-' . $slug,
    'article-' . $slug . '-comments'
]);
// $result contiendra
['article-first-post' => true, 'article-first-post-comments' => true]
```

Effacer les Données du Cache

static Cake\Cache\Cache::**clear**(*$check*, *$config = 'default'*)

Détruit toute les valeurs pour une configuration de cache. Pour les moteurs tels que APC, Memcached et Wincache, le préfixe de la configuration du cache est utilisé pour supprimer les données de cache. Assurez-vous que les différentes configurations de cache ont des préfixes différents :

```
// Détruira uniquement les clés expirées.
Cache::clear(true);
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Détruira toutes les clés.
Cache::clear(false);
```

static Cake\Cache\Cache::**gc**(*$config*)

Garbage collects entries in the cache configuration. C'est principalement utilisé par FileEngine. Elle ne devra être implémentée par tout moteur de Cache qui a besoin d'une suppresion manuelle des données mises en cache.

> **Note**
>
> Comme APC et Wincache utilisent des caches isolés pour le serveur web et le CLI, ils doivent être supprimés séparément (CLI ne peut pas nettoyer le serveur web et vice et versa).

Utiliser le Cache pour Stocker les Compteurs

static Cake\Cache\Cache::**increment**(*$key*, *$offset = 1*, *$config = 'default'*)

static Cake\Cache\Cache::**decrement**(*$key*, *$offset = 1*, *$config = 'default'*)

Les compteurs de votre application sont de bons candidats pour le stockage dans un cache. Par exemple, un simple compte à rebours pour des places restantes dans un concours peut être stocké dans le cache. La classe Cache expose des opérations atomiques pour incrémenter/décrémenter les valeurs du compteur. Les opérations atomiques sont importantes pour ces valeurs, car elle réduisent le risque de contention, et la capacité pour deux utilisateurs d'abaisser simultanément la valeur, ce qui entraînerait une valeur incorrecte.

Après avoir défini une valeur entière, vous pouvez la manipuler à l'aide des fonctions `increment()` et `decrement()` :

```
Cache::write('initial_count', 10);

// Plus tard
Cache::decrement('initial_count');

// Ou
Cache::increment('initial_count');
```

> **Note**
>
> L'incrémentation et la décrementation ne fonctionne pas avec FileEngine. A la place, vous devez utiliser APC, Wincache, Redis ou Memcached.

Utiliser le Cache pour Stocker les Résultats de Requêtes Courantes

Vous pouvez considérablement améliorer les performances de votre application en mettant dans le cache les résultats qui changent rarement, ou qui sont soumis à de nombreuses lectures. Un exemple parfait serait les résultats de `Cake\ORM\Table::find()`. l'objet Query vous permet de mettre les résultats en cache en utilisant la méthode `cache`. Voir la section *mettre les résultats de requête en cache* pour plus d'information.

Utilisation des Groupes

Parfois vous voudrez marquer plusieurs entrées de cache comme appartenant à un même groupe ou un namespace. C'est une exigence courante pour invalider de sgrosses quantités de clés alors que quelques changements d'informations sont partagés pour toutes les entrées dans un même groupe. Cela est possible en déclarant les groupes dans la configuration de cache :

```
Cache::config('site_home', [
    'className' => 'Redis',
    'duration' => '+999 days',
    'groups' => ['comment', 'article']
]);
```

Cake\Cache\Cache::**clearGroup**(*$group*, *$config = 'default'*)

Disons que vous voulez stocker le HTML généré pour votre page d'accueil dans le cache, mais vous voulez aussi invalider automatiquement ce cache à chaque fois qu'un commentaire ou un post est ajouté à votre base de données. En ajoutant les groupes `comment` et `article`, nous avons effectivement taggé les clés stockées dans la configuration du cache avec les noms des deux groupes.

Par exemple, dès qu'un post est ajouté, nous pouvons dire au moteur de Cache de retirer toutes les entrées associées au groupe `article` :

```
// src/Model/Table/ArticlesTable.php
public function afterSave($event, $entity, $options = [])
{
    if ($entity->isNew()) {
        Cache::clearGroup('article', 'site_home');
    }
}
```

static Cake\Cache\Cache::**groupConfigs**(*$group = null*)

`groupConfigs()` peut être utilisée pour récupérer la correspondance entre des groupes et des configurations, par exemple ayant le même groupe :

```
// src/Model/Table/ArticlesTable.php

/**
 * Une variante de l'exemple précédent qui efface toutes les configurations
 * ayant le même groupe
 */
public function afterSave($event, $entity, $options = [])
{
    if ($entity->isNew()) {
        $configs = Cache::groupConfigs('article');
        foreach ($configs['article'] as $config) {
            Cache::clearGroup('article', $config);
        }
    }
}
```

Les groupes sont partagés à travers toutes les configs de cache en utilisant le même moteur et le même préfixe. Si vous utilisez les groupes et voulez tirer profit de la suppression de groupe, choisissez un préfixe commun pour toutes vos configs.

Activer ou Désactiver Globalement le Cache

static Cake\Cache\Cache::**disable**

Vous pourriez avoir besoin de désactiver toutes les lectures/écritures du Cache en essayant de comprendre des problèmes liés à l'expiration du cache. Vous pouvez le faire en utilisant `enable()` et `disable()` :

```
// Désactive toutes les lectures/écritures
Cache::disable();
```

Une fois désactivé, toutes lecture/écriture renverra `null`.

static Cake\Cache\Cache::**enable**

Une fois désactivé, utilisez `enable()` pour réactiver le cache :

```
// Active de nouveau toutes les lectures/écritures
Cache::enable();
```

static Cake\Cache\Cache::**enabled**

Si vous voulez vérifier l'état du Cache, utilisez `enabled()`.

Création d'un moteur de stockage pour le Cache

Vous pouvez fournir vos propre adaptateurs `Cache` dans `App\Cache\Engine` ou dans un plugin en utilisant `$plugin\Cache\Engine`. Les moteurs de cache src/plugin peuvent aussi remplacer les moteurs du cœur. Les adaptateurs de cache doivent être dans un répertoire cache. Si vous avez un moteur de cache nommé `MyCustomCacheEngine` il devra être placé soit dans **src/Cache/Engine/MyCustomCacheEngine.php** comme une app/libs ou dans **plugin/Cache/Engine/MyCustomCacheEngine.php** faisant parti d'un plugin. Les configurations de cache venant d'un plugin doivent utiliser la notation par points de plugin :

```
Cache::config('custom', [
    'engine' => 'CachePack.MyCustomCache',
    // ...
]);
```

Les moteurs de cache personnalisés doivent étendre *Cake\Cache\CacheEngine* qui définit un certain nombre de méthodes d'abstraction ainsi que quelques méthodes d'initialisation.

L'API requise pour CacheEngine est

class Cake\Cache**CacheEngine**

La classe de base pour tous les moteurs de cache utilisée avec le Cache.

Cake\Cache\CacheEngine::**write**(*$key*, *$value*, *$config = 'default'*)

Retourne
un booléen en cas de succès.

Écrit la valeur d'une clé dans le cache, la chaîne optionnelle $config spécifie le nom de la configuration à écrire.

Cake\Cache\CacheEngine::**read**(*$key*)

Retourne
La valeur mise en cache ou `false` en cas d'échec.

Lit une clé depuis le cache. Retourne `false` pour indiquer que l'entrée a expiré ou n'existe pas.

`Cake\Cache\CacheEngine::delete($key)`

Retourne
Un booléen `true` en cas de succès.

Efface une clé depuis le cache. Retourne `false` pour indiquer que l'entrée n'existe pas ou ne peut être effacée.

`Cake\Cache\CacheEngine::clear($check)`

Retourne
Un booléen `true` en cas de succès.

Efface toutes les clés depuis le cache. Si $check est à `true`, vous devez valider que chacune des valeurs a réellement expirée.

`Cake\Cache\CacheEngine::clearGroup($group)`

Renvoie
Un booléen `true` en cas de succès.

Supprime toutes les clés à partir du cache appartenant au même groupe.

`Cake\Cache\CacheEngine::decrement($key, $offset = 1)`

Retourne
Un booléen `true` en cas de succès.

Décrémente un nombre dans la clé et retourne la valeur décrémentée

`Cake\Cache\CacheEngine::increment($key, $offset = 1)`

Retourne
Un booléen `true` en cas de succès.

Incrémente un nombre dans la clé et retourne la valeur incrémentée

`static Cake\Cache\CacheEngine::gc`

Non requise, mais utilisée pour faire du nettoyage quand les ressources expirent. Le moteur FileEngine utilise cela pour effacer les fichiers qui contiennent des contenus expirés.

CHAPITRE 14

Console Bake

Cette page a été déplacée [142].

142. https://book.cakephp.org/bake/1.x/fr/

CHAPITRE 15

Commandes sur la Console

En plus d'un framework web, CakePHP fournit aussi un framework de console pour créer des outils et applications en ligne de commande. Les applications sur console sont idéales pour traiter diverses tâches en arrière-plan ou opérations de maintenance qui influent sur la configuration existante de votre application, vos modèles, plugin et logique de domaine.

CakePHP propose plusieurs outils de console pour interagir avec les fonctionnalités de CakePHP telles que l'internationalisation et le routage, qui vous permettent d'introspecter votre application et des générer des fichiers correspondant.

La Console CakePHP

La console CakePHP utilise un système de type dispatcher pour charger des commandes, parser leurs arguments et appeler la bonne commande. Les exemples qui suivent utilisent bash ; cependant la console CakePHP est compatible avec n'importe quel shell *nix et Windows.

Une application CakePHP contient les répertoires **src/Command**, **src/Shell** et **src/Shell/Task**, qui contiennent ses shells et ses tâches. Elle est aussi livrée avec un exécutable dans le répertoire **bin** :

```
$ cd /path/to/app
$ bin/cake
```

> **Note**
>
> Pour Windows, il faut taper la commande `bin\cake` (notez le backslash).

Si vous lancez la Console sans arguments, vous obtiendrez la liste des commandes disponibles. Vous pouvez ensuite lancer une de ces commandes en tapant son nom :

```
# lancer le shell du serveur
bin/cake server
```

(suite sur la page suivante)

(suite de la page précédente)

```
# lancer le shell des migrations
bin/cake migrations -h

# lancer bake (avec un préfixe de plugin)
bin/cake bake.bake -h
```

Vous pouvez appeler des commandes de plugin sans le préfixe du plugin si le nom de la commande n'entre pas en collision avec un shell de l'application ou du framework. Dans le cas où deux plugins proposent une commande avec le même nom, l'alias court correspondra au plugin chargé en premier. Vous pouvez toujours utiliser le format `plugin.command` pour faire référence à une commande de manière non ambigüe.

Applications sur Console

Par défaut, CakePHP cherchera automatiquement toutes les commandes dans votre application et vos plugins. Quand vous créerez des applications autonoms sur console, vous voudrez peut-être restreindre le nombre de commandes accessibles. Vous pouvez utiliser le crochet `console()` de `Application` pour limiter et renommer les commandes exposées :

```php
// dans src/Application.php
namespace App;

use App\Command\UserCommand;
use App\Command\VersionCommand;
use Cake\Console\CommandCollection;
use Cake\Http\BaseApplication;

class Application extends BaseApplication
{
    public function console(CommandCollection $commands): CommandCollection
    {
        // Ajouter par nom de classe
        $commands->add('user', UserCommand::class);

        // Ajouter une instance
        $commands->add('version', new VersionCommand());

        return $commands;
    }
}
```

Dans cet exemple, seules les commandes `help`, `version` et `user` seraient disponibles. Consultez la section *Commands* pour savoir comment ajouter des commandes dans vos plugins.

> **Note**
>
> Quand vous ajoutez plusieurs commandes qui utilisent la même classe Command, la commande `help` affichera l'option la plus courte.

Renommer des Commandes

Dans certains cas, vous aurez besoin de renommer des commandes, ou de créer des commandes imbriquées ou des sous-commandes. La découverte automatique des commandes ne fera pas cela, cependant vous pouvez déclarer vos commandes pour créer la dénomination désirée.

Vous pouvez personnaliser les noms de commandes en définissant chaque commande dans votre plugin :

```
public function console(CommandCollection $commands): CommandCollection
{
    // Ajouter des commandes avec une dénomintaion imbriquée
    $commands->add('user dump', UserDumpCommand::class);
    $commands->add('user:show', UserShowCommand::class);

    // Renommer entièrement une commande
    $commands->add('lazer', UserDeleteCommand::class);

    return $commands;
}
```

Quand vous réécrivez le crochet `console()` de votre application, pensez à appeler `$commands->autoDiscover()` pour ajouter des commandes de CakePHP, de votre application, et des plugins.

Si vous avez besoin de renommer ou supprimer une commande attachée, vous pouvez utiliser l'événement `Console.buildCommands` dans le gestionnaire d'événements de votre application pour modifier les commandes disponibles.

Commandes

Rendez-vous au chapitre *Objets Command* pour créer votre première commande. Puis, pour en savoir plus sur les commandes :

Objets Command

class Cake\Console**Command**

CakePHP met à disposition des commandes pour accélerer vos développements et automatiser les tâches routinières. Vous pouvez utiliser ces mêmes librairies pour créer des commandes pour votre application et vos plugins.

Créer une Commande

Créons maintenant notre première commande. Pour cet exemple, nous allons créer une commande Hello world toute simple. Dans le répertoire **src/Command** de votre application, créez **HelloCommand.php**. Mettez-y le code suivant :

```
<?php
namespace App\Command;

use Cake\Command\Command;
use Cake\Console\Arguments;
use Cake\Console\ConsoleIo;

class HelloCommand extends Command
{
    public function execute(Arguments $args, ConsoleIo $io): int
    {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        $io->out('Hello world.');

        return static::CODE_SUCCESS;
    }
}
```

Les classes Command doivent avoir une méthode `execute()` qui fait la plus grande partie du travail. Cette méthode est appelée quand une commande est lancée. Appelons la première commande de notre application, exécutez :

```
bin/cake hello
```

Vous devriez voir la sortie suivante :

```
Hello world.
```

Notre méthode `execute()` n'est pas très intéressente, ajoutons des entrées à partir de la ligne de commande :

```
<?php
namespace App\Command;

use Cake\Command\Command;
use Cake\Console\Arguments;
use Cake\Console\ConsoleIo;
use Cake\Console\ConsoleOptionParser;

class HelloCommand extends Command
{
    protected function buildOptionParser(ConsoleOptionParser $parser):␣
↪ConsoleOptionParser
    {
        $parser->addArgument('name', [
            'help' => 'Quel est votre nom'
        ]);

        return $parser;
    }

    public function execute(Arguments $args, ConsoleIo $io): int
    {
        $name = $args->getArgument('name');
        $io->out("Hello {$name}.");

        return static::CODE_SUCCESS;
    }
}
```

Après avoir sauvegardé ce fichier, vous devriez pouvoir exécuter la commande suivante :

```
bin/cake hello jillian

# Affiche
Hello jillian
```

Changer le Nom Par Défaut de la Commande

CakePHP va s'appuyer sur des conventions pour générer le nom que vos commandes utilisent en ligne de commande. Si vous voulez remplacer le nom généré, implémentez la méthode `defaultName()` dans votre commande :

```
public static function defaultName(): string
{
    return 'oh_hi';
}
```

Ceci rendrait `HelloCommand` accessible par `cake oh_hi` au lieu de `cake hello`.

Définir les Arguments et les Options

Comme nous avons vu dans le dernier exemple, nous pouvons utiliser la méthode hook `buildOptionParser()` pour définir des arguments. Nous pouvons aussi définir des options. Par exemple, nous pouvons ajouter une option `yell` à notre `HelloCommand` :

```
// ...
protected function buildOptionParser(ConsoleOptionParser $parser): ConsoleOptionParser
{
    $parser
        ->addArgument('name', [
            'help' => 'Quel est votre nom'
        ])
        ->addOption('yell', [
            'help' => 'Crier le nom',
            'boolean' => true
        ]);

    return $parser;
}

public function execute(Arguments $args, ConsoleIo $io): int
{
    $name = $args->getArgument('name');
    if ($args->getOption('yell')) {
        $name = mb_strtoupper($name);
    }
    $io->out("Hello {$name}.");

    return static::CODE_SUCCESS;
}
```

Consultez la section *Option Parsers* pour plus d'information.

Créer une Sortie

Les commands reçoivent une instance `ConsoleIo` quand elles sont exécutées. Cet objet vous permet d'interagir avec `stdout`, `stderr` et de créer des fichiers. Consultez la section *Entrée/Sortie de Commande* pour plus d'information.

Utiliser les Models dans les Commands

Vous aurez souvent besoin d'accéder à logique métier de votre application depuis les commandes console. Vous pouvez charger des modèles dans les commandes, exactement comme vous le feriez dans un controller en utilisant `$this->fetchTable()`, puisque les commandes utilisent `LocatorAwareTrait` :

```
<?php
declare(strict types=1);
namespace App\Command;

use Cake\Command\Command;
use Cake\Console\Arguments;
use Cake\Console\ConsoleIo;
use Cake\Console\ConsoleOptionParser;

class UserCommand extends Command
{
    // Définit la table par défaut. Cela vous permet d'utiliser `fetchTable()` sans␣
↪argument.
    protected $defaultTable = 'Users';

    protected function buildOptionParser(ConsoleOptionParser $parser):␣
↪ConsoleOptionParser
    {
        $parser
            ->addArgument('name', [
                'help' => 'Quel est votre nom'
            ]);

        return $parser;
    }

    public function execute(Arguments $args, ConsoleIo $io): int
    {
        $name = $args->getArgument('name');
        $user = $this->fetchTable()->findByUsername($name)->first();

        $io->out(print_r($user, true));

        return static::CODE_SUCCESS;
    }
}
```

La commande ci-dessus va récupérer un utilisateur par son nom d'utilisateur et afficher les informations stockées dans la base de données.

Codes de Sortie et Arrêter l'Execution

Quand vos commandes rencontrent une erreur irrécupérable, vous pouvez utiliser la méthode `abort()` pour terminer l'exécution :

```
// ...
public function execute(Arguments $args, ConsoleIo $io): int
{
```

(suite sur la page suivante)

(suite de la page précédente)

```
    $name = $args->getArgument('name');
    if (strlen($name) < 5) {
        // Halt execution, output to stderr, and set exit code to 1
        $io->error('Name must be at least 4 characters long.');
        $this->abort();
    }

    return static::CODE_SUCCESS;
}
```

Vous pouvez aussi utiliser `abort()` sur l'objet `$io` pour émettre un message et un code :

```
public function execute(Arguments $args, ConsoleIo $io): int
{
    $name = $args->getArgument('name');
    if (strlen($name) < 5) {
        // Arrête l'exécution, affiche vers stderr, et définit le code de sortie à 99
        $io->abort('Le nom doit avoir au moins 4 caractères.', 99);
    }

    return static::CODE_SUCCESS;
}
```

Vous pouvez passer n'importe quel code de sortie dans `abort()`.

Astuce

Évitez les codes de sortie 64 - 78, car ils ont une signification particulière décrite par `sysexits.h`. Évitez les codes de sortie au-dessus de 127, car ils sont utilisés pour indiquer une sortie de processus par signal tel que SIGKILL ou SIGSEGV.

Vous pouvez en savoir plus à propos des codes de sortie sur la manpage de sysexit sur la plupart des systèmes Unix (`man sysexits`), ou la page d'aide `System Error Codes` sous Windows.

Appeler d'Autres Commandes

Vous pouvez avoir besoin d'appeler d'autres commandes depuis votre commande. Pour ce faire, utilisez `executeCommand` :

```
// Vous pouvez passer un tableau d'options CLI et d'arguments.
$this->executeCommand(OtherCommand::class, ['--verbose', 'deploy']);

// Possibilité de passer une instance de commande si elle a des arguments de constructeur
$command = new OtherCommand($otherArgs);
$this->executeCommand($command, ['--verbose', 'deploy']);
```

Note

Quand vous appelez `executeCommand()` dans une boucle, il est recommandé de passer l'instance `ConsoleIo` de la commande parente en 3ème argument optionnel pour éviter une potentielle limite de fichiers ouverts, ce qui pourrait arriver dans certains environnements.

Définir la Description de la Commande

Vous pouvez définir une description de commande via :

```
class UserCommand extends Command
{
    public static function getDescription(): string
    {
        return 'Ma description personnalisée';
    }
}
```

Cela affichera votre description dans la CLI de Cake :

```
bin/cake

App:
  - user
  └── Ma description personnalisée
```

Ainsi que dans la section *help* de votre commande :

```
cake user --help
Ma description personnalisée

Usage:
cake user [-h] [-q] [-v]
```

Tester les Commandes

Pour faciliter les tests des applications de console, CakePHP fournit le trait `ConsoleIntegrationTestTrait` que vous pouvez utiliser pour tester les applications console et faire des assertions sur leurs résultats.

Pour commencer à tester votre application de console, créez un cas de test qui utilise le trait `Cake\TestSuite\ConsoleIntegrationTestTrait`. Ce trait contient une méthode `exec()` qui est utilisée pour exécuter votre commande. Vous pouvez y passer la même chaîne que celle que vous passeriez en ligne de commande.

> **Note**
>
> Pour CakePHP 4.4 et au-delà, il faut utiliser le namespace de `Cake\Console\TestSuite\ConsoleIntegrationTestTrait`

Commençons avec une commande très simple qui se trouve dans **src/Command/UpdateTableCommand.php** :

```
namespace App\Command;

use Cake\Command\Command;
use Cake\Console\Arguments;
use Cake\Console\ConsoleIo;
use Cake\Console\ConsoleOptionParser;

class UpdateTableCommand extends Command
{
    protected function buildOptionParser(ConsoleOptionParser $parser):␣
```

(suite sur la page suivante)

(suite de la page précédente)

```
↪ConsoleOptionParser
    {
        $parser->setDescription('Mon application de console super cool');

        return $parser;
    }
}
```

Pour écrire un test d'intégration pour ce shell, nous créons un cas de test dans **tests/TestCase/Command/UpdateTableTest.php** qui utilise le trait `Cake\TestSuite\ConsoleIntegrationTestTrait`. Ce shell ne fait pas grand chose pour le moment, mais testons simplement si la description de notre shell description s'affiche dans `stdout` :

```
namespace App\Test\TestCase\Command;

use Cake\TestSuite\ConsoleIntegrationTestTrait;
use Cake\TestSuite\TestCase;

class UpdateTableCommandTest extends TestCase
{
    user ConsoleIntegrationTestTrait;

    public function testDescriptionOutput()
    {
        $this->exec('update_table --help');
        $this->assertOutputContains('Mon application de console super cool');
    }
}
```

Notre test passe ! Bien que ce soit un exemple très facile, cela montre que créer un cas de test d'intégration pour nos applications de console peut suivre les conventions de la ligne de commande. Continuons en ajoutant plus de logique à notre commande :

```
namespace App\Command;

use Cake\Command\Command;
use Cake\Console\Arguments;
use Cake\Console\ConsoleIo;
use Cake\Console\ConsoleOptionParser;
use Cake\I18n\FrozenTime;

class UpdateTableCommand extends Command
{
    protected function buildOptionParser(ConsoleOptionParser $parser):␣
↪ConsoleOptionParser
    {
        $parser
            ->setDescription('Mon application de console super cool')
            ->addArgument('table', [
                'help' => 'Table à mettre à jour',
                'required' => true
            ]);
```

(suite sur la page suivante)

(suite de la page précédente)

```
        return $parser;
    }

    public function execute(Arguments $args, ConsoleIo $io): int
    {
        $table = $args->getArgument('table');
        $this->fetchTable($table)->query()
            ->update()
            ->set([
                'modified' => new FrozenTime()
            ])
            ->execute();

        return static::CODE_SUCCESS;
    }
}
```

C'est un shell plus complet qui a des options obligatoires et une logique associée. Modifions notre cas de test en y intégrant le code suivant :

```
namespace Cake\Test\TestCase\Command;

use Cake\Command\Command;
use Cake\I18n\FrozenTime;
use Cake\TestSuite\ConsoleIntegrationTestTrait;
use Cake\TestSuite\TestCase;

class UpdateTableCommandTest extends TestCase
{
    use ConsoleIntegrationTestTrait;

    protected $fixtures = [
        // assume que vous avez une UsersFixture
        'app.Users'
    ];

    public function testDescriptionOutput()
    {
        $this->exec('update_table --help');
        $this->assertOutputContains('Mon application de console super cool');
    }

    public function testUpdateModified()
    {
        $now = new FrozenTime('2017-01-01 00:00:00');
        FrozenTime::setTestNow($now);

        $this->loadFixtures('Users');

        $this->exec('update_table Users');
        $this->assertExitCode(Command::CODE_SUCCESS);
```

(suite sur la page suivante)

(suite de la page précédente)

```
        $user = $this->getTableLocator()->get('Users')->get(1);
        $this->assertSame($user->modified->timestamp, $now->timestamp);

        FrozenTime::setTestNow(null);
    }
}
```

Comme vous pouvez le voir dans la méthode `testUpdateModified`, nous testons que notre commande met à jour la table que nous passons en premier argument. Premièrement, nous faisons l'assertion que la commande se termine avec le bon code de sortie `0`. Ensuite nous vérifions que notre commande a fait le travail, qui est de mettre à jour la table que nous avons fournie et d'insérer la date et l'heure actuelle dans la colonne `modified`.

Souvenez-vous que `exec()` va prendre la même chaîne que si vous tapiez dans le CLI, donc vous pouvez inclure des options et des arguments dans la chaîne de votre commande.

Tester les Shells Interactifs

Les consoles sont souvent interactives. Pour tester les shells interactifs avec le trait `Cake\TestSuite\ConsoleIntegrationTestTrait`, vous devez seulement passer les entrées attendues en deuxième paramètre de `exec()`. Ils doivent être présentés dans un tableau dans l'ordre dans lequel vous voulez les passer.

Continuons notre exemple de commande, et ajoutons une confirmation interactive. Mettez à jour la classe de commande de la façon suivante :

```
namespace App\Command;

use Cake\Command\Command;
use Cake\Console\Arguments;
use Cake\Console\ConsoleIo;
use Cake\Console\ConsoleOptionParser;
use Cake\I18n\FrozenTime;

class UpdateTableCommand extends Command
{
    protected function buildOptionParser(ConsoleOptionParser $parser):␣
↪ConsoleOptionParser
    {
        $parser
            ->setDescription('Mon application de console super cool')
            ->addArgument('table', [
                'help' => 'Table à mettre à jour',
                'required' => true
            ]);

        return $parser;
    }

    public function execute(Arguments $args, ConsoleIo $io): int
    {
        $table = $args->getArgument('table');
        $this->loadModel($table);
        if ($io->ask('Êtes-vous sûr ?', 'n', ['o', 'n']) === 'n') {
```

(suite sur la page suivante)

(suite de la page précédente)

```
            $io->error('Vous devez être sûr.');
            $this->abort();
        }
        $this->fetchTable($table)->query()
            ->update()
            ->set([
                'modified' => new FrozenTime()
            ])
            ->execute();

        return static::CODE_SUCCESS;
    }
}
```

Maintenant que nous avons une sous-commande interactive, nous pouvons ajouter un cas de test qui vérifie que nous recevons une réponse positive et un qui vérifie que nous recevons une réponse négative. Retirez la méthode `testUpdateModified` et ajoutez les méthodes qui suivent dans **tests/TestCase/Command/UpdateTableCommandTest.php** :

```
public function testUpdateModifiedSure()
{
    $now = new FrozenTime('2017-01-01 00:00:00');
    FrozenTime::setTestNow($now);

    $this->loadFixtures('Users');

    $this->exec('update_table Users', ['o']);
    $this->assertExitCode(Command::CODE_SUCCESS);

    $user = $this->getTableLocator()->get('Users')->get(1);
    $this->assertSame($user->modified->timestamp, $now->timestamp);

    FrozenTime::setTestNow(null);
}

public function testUpdateModifiedUnsure()
{
    $user = $this->getTableLocator()->get('Users')->get(1);
    $original = $user->modified->timestamp;

    $this->exec('my_console best_framework', ['n']);
    $this->assertExitCode(Command::CODE_ERROR);
    $this->assertErrorContains('You need to be sure.');

    $user = $this->getTableLocator()->get('Users')->get(1);
    $this->assertSame($original, $user->timestamp);
}
```

Dans le premier cas de test, nous confirmons la question, et les enregistrements sont mis à jour. Dans le deuxième test, nous ne confirmons pas et les enregistrements ne sont pas mis à jour, et nous pouvons vérifier que le message d'erreur a été écrit dans `stderr`.

Méthodes d'Assertion

Le trait `Cake\TestSuite\ConsoleIntegrationTestTrait` fournit de nombreuses méthodes d'assertion qui aident à vérifier la sortie de la console :

```
// vérifie que le shell s'est terminé avec le code attendu
$this->assertExitCode($expected);

// vérifie que stdout contient une chaîne de caractères
$this->assertOutputContains($expected);

// vérifie que stderr contient une chaîne de caractères
$this->assertErrorContains($expected);

// vérifie que stdout répond à une expression régulière
$this->assertOutputRegExp($expected);

// vérifie que stderr répond à une expression régulière
$this->assertErrorRegExp($expected);
```

Entrée/Sortie de Commande

class `Cake\Console\`**`ConsoleIo`**

CakePHP fournit l'objet `ConsoleIo` aux commandes afin qu'elles puissent lire interactivement les informations entrées par l'utilisateur et afficher d'autres informations en sortie pour l'utilisateur.

Helpers (Assistants) de commande

Les Helpers (Assistants) de commande sont accessibles et utilisables depuis n'importe quelle commande, shell ou tâche :

```
// Affiche des données en tant que tableau.
$io->helper('Table')->output($data);

// Récupère un helper depuis un plugin.
$io->helper('Plugin.HelperName')->output($data);
```

Vous pouvez aussi récupérer des instances de Helpers et appeler n'importe quelle méthode publique dessus :

```
// Récupérer et utiliser le helper Progress.
$progress = $io->helper('Progress');
$progress->increment(10);
$progress->draw();
```

Créer des Helpers (Assistants)

CakePHP est fourni avec quelques helpers de commande, cependant vous pouvez en créer d'autres dans votre application ou vos plugins. À titre d'exemple, nous allons créer un helper simple pour générer des titres élégants. Créez d'abord le fichier **src/Command/Helper/HeadingHelper.php** et insérez-y ceci :

```
<?php
namespace App\Command\Helper;
```

(suite sur la page suivante)

(suite de la page précédente)

```
use Cake\Console\Helper;

class HeadingHelper extends Helper
{
    public function output($args)
    {
        $args += ['', '#', 3];
        $marker = str_repeat($args[1], $args[2]);
        $this->_io->out($marker . ' ' . $args[0] . ' ' . $marker);
    }
}
```

Nous pouvons alors utiliser ce nouvel Helper dans l'une de nos commandes shell en l'appelant :

```
// Avec ### de chaque coté
$this->helper('Heading')->output(['Ça marche !']);

// Avec ~~~~ de chaque coté
$this->helper('Heading')->output(['Ça marche !', '~', 4]);
```

Les Helpers implémentent généralement la méthode `output()` qui prend un tableau de paramètres. Cependant, comme les Console Helper sont des classes vanilla, ils implémentent des méthodes suplémentaires qui prennent n'importe quelle forme d'arguments.

> **Note**
>
> Les Helpers peuvent aussi être placés dans `src/Shell/Helper` pour des raisons de retro-compatibilité.

Les Helpers inclus

Helper Table

Le TableHelper aide à faire des tableaux ASCII bien formatés. L'utilisation est assez simple :

```
$data = [
    ['Header 1', 'Header', 'Long Header'],
    ['short', 'Longish thing', 'short'],
    ['Longer thing', 'short', 'Longest Value'],
];
$io->helper('Table')->output($data);

// Affiche
+--------------+---------------+---------------+
| Header 1     | Header        | Long Header   |
+--------------+---------------+---------------+
| short        | Longish thing | short         |
| Longer thing | short         | Longest Value |
+--------------+---------------+---------------+
```

Vousp ouvez utiliser la balise de formatage `<text-right>` dans les tables pour en aligner son contenu à droite :

```
$data = [
    ['Nom 1', 'Prix Total'],
    ['Cake Mix', '<text-right>1.50</text-right>'],
];
$io->helper('Table')->output($data);

// Outputs
+----------+------------+
| Nom 1    | Prix Total |
+----------+------------+
| Cake Mix |       1.50 |
+----------+------------+
```

Ajouté dans la version 4.2.0 : La balise de formatage `<text-right>` a été ajoutée dans 4.2.

Helper Progress

Le ProgressHelper peut être utilisé de deux façons. Le mode simple vous permet de fournir un callback qui est appelé jusqu'à ce que l'avancement soit complet :

```
$io->helper('Progress')->output(['callback' => function ($progress) {
    // Faire des choses ici.
    $progress->increment(20);
    $progress->draw();
}]);
```

Vous pouvez contrôler davantage la barre de progression en fournissant des options supplémentaires :

- `total` Le nombre total d'éléments dans la barre de progression. La valeur par défaut est 100.
- `width` La largeur de la barre de progression. La valeur par défaut est 80.
- `callback` Le callback qui sera appelé dans une boucle pour faire avancer la barre de progression.

Voici un exemple de toutes les options utilisées :

```
$io->helper('Progress')->output([
    'total' => 10,
    'width' => 20,
    'callback' => function ($progress) {
        $progress->increment(2);
        $progress->draw();
    }
]);
```

Le ProgressHelper peut aussi être utilisé manuellement pour incrémenter et réafficher la barre de progression selon les besoins :

```
$progress = $io->helper('Progress');
$progress->init([
    'total' => 10,
    'width' => 20,
]);

$progress->increment(4);
$progress->draw();
```

Récupérer l'entrée utilisateur

`Cake\Console\ConsoleIo::ask($question, $choices = null, $default = null)`

Lorsque vous créez des applications de console interactives, vous aurez besoin de récupérer des entrées de l'utilisateur. CakePHP fournit un moyen de le faire :

```
// Obtenir un texte quelconque de l'utilisateur.
$color = $io->ask('Quelle couleur aimez-vous ?');

// Obtenir un choix de l'utilisateur.
$selection = $io->askChoice('Rouge ou Vert ?', ['R', 'V'], 'R');
```

La validation de la sélection est insensible à la casse.

Créer des fichiers

`Cake\Console\ConsoleIo::createFile($path, $contents)`

Souvent, une partie importante des commandes de console consiste à créer des fichiers, afin d'aider à automatiser le développement et le déploiement. La méthode `createFile()` donne une interface simple pour créer des fichiers, avec une confirmation interactive :

```
// Créer un fichier demandant de confirmer l'écrasement
$io->createFile('bower.json', $stuff);

// Forcer l'écrasement sans demander
$io->createFile('bower.json', $stuff, true);
```

Créer une sortie

Une autre opération courante dans CakePHP consiste à écrire dans `stdout` et `stderr` :

```
// Écrire dans stdout
$io->out('Message normal');

// Écrire dans stderr
$io->err('Message d\'erreur');
```

En plus des méthodes de sortie vanilla, CakePHP fournit des méthodes *wrappers* qui stylisent la sortie avec les couleurs ANSI appropriées :

```
// Texte vert dans stdout
$io->success('Message de réussite');

// Texte cyan dans stdout
$io->info('Texte informatif');

// Texte bleu dans stdout
$io->comment('Supplément de contexte');

// Texte rouge dans stderr
$io->error('Texte d\'erreur');
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Texte jaune dans stderr
$io->warning('Texte d\'avertissement');
```

Le formatage en couleur sera automatiquement désactivé si `posix_isatty` renvoie true, ou si la variable d'environnement `NO_COLOR` est définie.

`ConsoleIo` fournit deux méthodes de confort à propos du niveau de sortie :

```
// N'apparaît que lorsque la sortie verbose est activée. (-v)
$io->verbose('Message verbeux');

// Apparaîtrait à tous les niveaux.
$io->quiet('Message succinct');
```

Vous pouvez également créer des lignes vierges ou tracer des lignes de tirets :

```
// Affiche 2 ligne vides
$io->out($this->nl(2));

// Dessiner une ligne horizontale
$io->hr();
```

Pour finir, vous pouvez mettre à jour la ligne de texte actuelle à l'écran :

```
$io->out('Compte à rebours');
$io->out('10', 0);
for ($i = 9; $i > 0; $i--) {
    sleep(1);
    $io->overwrite($i, 0, 2);
}
```

Note

Il est important de se rappeler que vous ne pouvez pas écraser le texte une fois qu'un retour à la ligne a été affiché.

Output Levels

Les applications de console ont souvent besoin de différents niveaux de verbosité. Par exemple, lors de l'exécution d'une tâche cron, la plupart des sorties ne sont pas nécessaires. Vous pouvez utiliser les niveaux de sortie pour baliser l'affichage de manière appropriée. L'utilisateur de l'interpréteur de commandes peut alors décider du niveau de détail qui l'intéresse en sélectionnant le bon indicateur lors de l'appel de la commande. Il y a 3 niveaux :

- `QUIET` - Seulement les informations absolument importantes devraient être marquées en sortie *quiet*.
- `NORMAL` -Le niveau par défaut, pour un usage normal.
- `VERBOSE` - Marquez `VERBOSE` les messages qui peuvent être trop verbeux pour un usage régulier, mais utiles pour du débogage .

Vous pouvez marquer la sortie comme ceci :

```
// Apparaîtra à tous les niveaux.
$io->out('Message succinct', 1, ConsoleIo::QUIET);
$io->quiet('Message succinct');

// N'apparaît pas lorsque la sortie silencieuse est activée.
```

(suite sur la page suivante)

(suite de la page précédente)

```
$io->out('message normal', 1, ConsoleIo::NORMAL);
$io->out('message bavard', 1, ConsoleIo::VERBOSE);
$io->verbose('Sortie verbeuse');

// N'apparaît que lorsque la sortie verbose est activée.
$io->out('message extra', 1, ConsoleIo::VERBOSE);
$io->verbose('Sortie verbeuse');
```

Vous pouvez contrôler le niveau de sortie des shells, en utilisant les options `--quiet` et `--verbose`. Ces options sont ajoutées par défaut, et vous permettent de contrôler les niveaux de sortie à l'intérieur de vos commandes CakePHP.

Les options `--quiet` et `--verbose` contrôlent aussi l'affichage des données de journalisation dans stdout/stderr. Normalement, les messages de journalisation d'information et supérieurs sont affichés dans stdout/stderr. Avec `--verbose`, le journal de débogage sera affiché dans stdout. Avec `--quiet`, seuls les messages d'avertissement et supérieurs seront affichés dans stderr.

Styliser la sortie

Vous pouvez donner un style à la sortie en incluant des balises dans votre sortie - comme en HTML. Ces balises seront remplacées par la bonne séquence de code ANSI, ou supprimées si vous êtes sur une console qui ne supporte pas les codes ANSI. Il existe plusieurs styles intégrés, et vous pouvez en créer d'autres. Ceux qui sont intégrés sont :

- `success` Messages de succès. Texte vert.
- `error` Messages d'erreur. Texte rouge.
- `warning` Messages d'avertissement. Texte jaune.
- `info` Messages d'information. Texte cyan.
- `comment` Texte additionnel. Texte bleu.
- `question` Texte qui est une question, ajouté automatiquement par le shell.

Vous pouvez créer des styles supplémentaires en utilisant `$io->setStyle()`. Pour déclarer un nouveau style de sortie, vous pouvez faire :

```
$io->setStyle('flashy', ['text' => 'magenta', 'blink' => true]);
```

Cela vous permettrait alors d'utiliser une balise `<flashy>` dans votre sortie shell, et si les couleurs ANSI sont activées, ce texte serait affiché en magenta clignotant `$this->out('<flashy>Ouaaaah</flashy> Il y a un problème');`. En définissant des styles, vous pouvez utiliser les couleurs suivantes pour les attributs `text` et `background` :

- black
- blue
- cyan
- green
- magenta
- red
- white
- yellow

Vous pouvez également utiliser les options suivantes en tant que commutateurs booléens. Ils deviennent actifs quand vous leur attribuez une valeur évaluée à `true`.

- blink
- bold
- reverse
- underline

Une fois ajouté, un style est disponible sur toutes les instances de ConsoleOutput, de sorte que vous n'avez pas à redéclarer des styles à la fois pour des objets stdout et stderr.

Désactiver la Couleur

Bien que la colorisation soit très jolie, il peut arriver que vous souhaitiez la désactiver, ou la forcer à s'activer :

```
$io->outputAs(ConsoleOutput::RAW);
```

Cette instruction placera l'objet de sortie en mode de sortie brute. En mode de sortie brute, il n'y a aucun style. Vous pouvez utiliser trois modes.

- `ConsoleOutput::COLOR` - Sortie avec les codes d'échappement de couleur en place.
- `ConsoleOutput::PLAIN` - Sortie en texte simple, les balises de style connues seront supprimées de la sortie.
- `ConsoleOutput::RAW` - Sortie brute sans style ni formatage. C'est un mode approprié si vous générez du XML ou si vous voulez déboguer pour voir pourquoi votre style ne fonctionne pas.

Par défaut sur les systèmes *nix, les objets ConsoleOutput sont en mode sortie couleur. Sur les systèmes Windows, la sortie est par défaut en texte simple sauf si la variable d'environment `ANSICON` est présente.

Option Parsers

class `Cake\Console\`**`ConsoleOptionParser`**

Console applications typically take options and arguments as the primary way to get information from the terminal into your commands.

Defining an OptionParser

Commands and Shells provide a `buildOptionParser($parser)` hook method that you can use to define the options and arguments for your commands :

```
protected function buildOptionParser($parser)
{
    // Define your options and arguments.

    // Return the completed parser
    return $parser;
}
```

Shell classes use the `getOptionParser()` hook method to define their option parser :

```
public function getOptionParser()
{
    // Get an empty parser from the framework.
    $parser = parent::getOptionParser();

    // Define your options and arguments.

    // Return the completed parser
    return $parser;
}
```

Using Arguments

`Cake\Console\ConsoleOptionParser::`**`addArgument`**(*$name*, *$params = []*)

Positional arguments are frequently used in command line tools, and `ConsoleOptionParser` allows you to define positional arguments as well as make them required. You can add arguments one at a time with `$parser->addArgument();` or multiple at once with `$parser->addArguments();` :

```
$parser->addArgument('model', ['help' => 'The model to bake']);
```

You can use the following options when creating an argument :

- `help` The help text to display for this argument.
- `required` Whether this parameter is required.
- `index` The index for the arg, if left undefined the argument will be put onto the end of the arguments. If you define the same index twice the first option will be overwritten.
- `choices` An array of valid choices for this argument. If left empty all values are valid. An exception will be raised when parse() encounters an invalid value.

Arguments that have been marked as required will throw an exception when parsing the command if they have been omitted. So you don't have to handle that in your shell.

Adding Multiple Arguments

`Cake\Console\ConsoleOptionParser::addArguments`(*array $args*)

If you have an array with multiple arguments you can use `$parser->addArguments()` to add multiple arguments at once.

```
$parser->addArguments([
    'node' => ['help' => 'The node to create', 'required' => true],
    'parent' => ['help' => 'The parent node', 'required' => true]
]);
```

As with all the builder methods on ConsoleOptionParser, addArguments can be used as part of a fluent method chain.

Validating Arguments

When creating positional arguments, you can use the `required` flag, to indicate that an argument must be present when a shell is called. Additionally you can use `choices` to force an argument to be from a list of valid choices :

```
$parser->addArgument('type', [
    'help' => 'The type of node to interact with.',
    'required' => true,
    'choices' => ['aro', 'aco']
]);
```

The above will create an argument that is required and has validation on the input. If the argument is either missing, or has an incorrect value an exception will be raised and the shell will be stopped.

Using Options

`Cake\Console\ConsoleOptionParser::addOption`(*$name*, *$options = []*)

Options or flags are used in command line tools to provide unordered key/value arguments for your commands. Options can define both verbose and short aliases. They can accept a value (e.g `--connection=default`) or be boolean options (e.g `--verbose`). Options are defined with the `addOption()` method :

```
$parser->addOption('connection', [
    'short' => 'c',
    'help' => 'connection',
    'default' => 'default',
]);
```

The above would allow you to use either `cake myshell --connection=other`, `cake myshell --connection other`, or `cake myshell -c other` when invoking the shell.

Boolean switches do not accept or consume values, and their presence just enables them in the parsed parameters :

```
$parser->addOption('no-commit', ['boolean' => true]);
```

This option when used like `cake mycommand --no-commit something` would have a value of `true`, and "something" would be a treated as a positional argument.

When creating options you can use the following options to define the behavior of the option :

- `short` - The single letter variant for this option, leave undefined for none.
- `help` - Help text for this option. Used when generating help for the option.
- `default` - The default value for this option. If not defined the default will be `true`.
- `boolean` - The option uses no value, it's just a boolean switch. Defaults to `false`.
- `choices` - An array of valid choices for this option. If left empty all values are valid. An exception will be raised when parse() encounters an invalid value.

Adding Multiple Options

`Cake\Console\ConsoleOptionParser::addOptions(array $options)`

If you have an array with multiple options you can use `$parser->addOptions()` to add multiple options at once.

```
$parser->addOptions([
    'node' => ['short' => 'n', 'help' => 'The node to create'],
    'parent' => ['short' => 'p', 'help' => 'The parent node']
]);
```

As with all the builder methods on ConsoleOptionParser, addOptions can be used as part of a fluent method chain.

Validating Options

Options can be provided with a set of choices much like positional arguments can be. When an option has defined choices, those are the only valid choices for an option. All other values will raise an `InvalidArgumentException` :

```
$parser->addOption('accept', [
    'help' => 'What version to accept.',
    'choices' => ['working', 'theirs', 'mine']
]);
```

Using Boolean Options

Options can be defined as boolean options, which are useful when you need to create some flag options. Like options with defaults, boolean options always include themselves into the parsed parameters. When the flags are present they are set to `true`, when they are absent they are set to `false` :

```
$parser->addOption('verbose', [
    'help' => 'Enable verbose output.',
    'boolean' => true
]);
```

The following option would always have a value in the parsed parameter. When not included its default value would be `false`, and when defined it will be `true`.

Building a ConsoleOptionParser from an Array

`Cake\Console\ConsoleOptionParser::buildFromArray($spec)`

As previously mentioned, when creating subcommand option parsers, you can define the parser spec as an array for that method. This can help make building subcommand parsers easier, as everything is an array :

```
$parser->addSubcommand('check', [
    'help' => __('Check the permissions between an ACO and ARO.'),
    'parser' => [
        'description' => [
            __("Use this command to grant ACL permissions. Once executed, the "),
            __("ARO specified (and its children, if any) will have ALLOW access "),
            __("to the specified ACO action (and the ACO's children, if any).")
        ],
        'arguments' => [
            'aro' => ['help' => __('ARO to check.'), 'required' => true],
            'aco' => ['help' => __('ACO to check.'), 'required' => true],
            'action' => ['help' => __('Action to check')]
        ]
    ]
]);
```

Inside the parser spec, you can define keys for `arguments`, `options`, `description` and `epilog`. You cannot define `subcommands` inside an array style builder. The values for arguments, and options, should follow the format that *`Cake\Console\ConsoleOptionParser::addArguments()`* and *`Cake\Console\ConsoleOptionParser::addOptions()`* use. You can also use buildFromArray on its own, to build an option parser :

```
public function getOptionParser()
{
    return ConsoleOptionParser::buildFromArray([
        'description' => [
            __("Use this command to grant ACL permissions. Once executed, the "),
            __("ARO specified (and its children, if any) will have ALLOW access "),
            __("to the specified ACO action (and the ACO's children, if any).")
        ],
        'arguments' => [
            'aro' => ['help' => __('ARO to check.'), 'required' => true],
            'aco' => ['help' => __('ACO to check.'), 'required' => true],
            'action' => ['help' => __('Action to check')]
        ]
    ]);
}
```

Merging Option Parsers

`Cake\Console\ConsoleOptionParser::merge($spec)`

When building a group command, you maybe want to combine several parsers for this :

```
$parser->merge($anotherParser);
```

Note that the order of arguments for each parser must be the same, and that options must also be compatible for it work. So do not use keys for different things.

Getting Help from Shells

By defining your options and arguments with the option parser CakePHP can automatically generate rudimentary help information and add a `--help` and `-h` to each of your commands. Using one of these options will allow you to see the generated help content :

```
bin/cake bake --help
bin/cake bake -h
```

Would both generate the help for bake. You can also get help for nested commands :

```
bin/cake bake model --help
bin/cake bake model -h
```

The above would get you the help specific to bake's model command.

Getting Help as XML

When building automated tools or development tools that need to interact with CakePHP shells, it's nice to have help available in a machine parse-able format. By providing the `xml` option when requesting help you can have help content returned as XML :

```
cake bake --help xml
cake bake -h xml
```

The above would return an XML document with the generated help, options, arguments and subcommands for the selected shell. A sample XML document would look like :

```
<?xml version="1.0"?>
<shell>
    <command>bake fixture</command>
    <description>Generate fixtures for use with the test suite. You can use
        `bake fixture all` to bake all fixtures.</description>
    <epilog>
        Omitting all arguments and options will enter into an interactive
        mode.
    </epilog>
    <options>
        <option name="--help" short="-h" boolean="1">
            <default/>
            <choices/>
        </option>
        <option name="--verbose" short="-v" boolean="1">
            <default/>
            <choices/>
        </option>
        <option name="--quiet" short="-q" boolean="1">
            <default/>
            <choices/>
        </option>
        <option name="--count" short="-n" boolean="">
            <default>10</default>
            <choices/>
        </option>
        <option name="--connection" short="-c" boolean="">
```

(suite sur la page suivante)

(suite de la page précédente)

```
            <default>default</default>
            <choices/>
        </option>
        <option name="--plugin" short="-p" boolean="">
            <default/>
            <choices/>
        </option>
        <option name="--records" short="-r" boolean="1">
            <default/>
            <choices/>
        </option>
    </options>
    <arguments>
        <argument name="name" help="Name of the fixture to bake.
            Can use Plugin.name to bake plugin fixtures." required="">
            <choices/>
        </argument>
    </arguments>
</shell>
```

Customizing Help Output

You can further enrich the generated help content by adding a description, and epilog.

Set the Description

`Cake\Console\ConsoleOptionParser::setDescription($text)`

The description displays above the argument and option information. By passing in either an array or a string, you can set the value of the description :

```
// Set multiple lines at once
$parser->setDescription(['line one', 'line two']);

// Read the current value
$parser->getDescription();
```

Set the Epilog

`Cake\Console\ConsoleOptionParser::setEpilog($text)`

Gets or sets the epilog for the option parser. The epilog is displayed after the argument and option information. By passing in either an array or a string, you can set the value of the epilog :

```
// Set multiple lines at once
$parser->setEpilog(['line one', 'line two']);

// Read the current value
$parser->getEpilog();
```

Exécuter des Shells en Tâches Cron (Cron Jobs)

Une action habituelle à faire avec un shell est de l'exécuter par une tâche cron pour nettoyer la base de données de temps en temps ou pour envoyer des newsletters. La configuration est un jeu d'enfant, par exemple :

```
  */5  *    *    *    *  cd /chemin/complet/vers/la/racine && bin/cake monshell monparam
# *    *    *    *    *  commande à exécuter
# |    |    |    |    |
# |    |    |    |    |
# |    |    |    |    \------ jour de la semaine (0 - 6) (0 à 6 vont de dimanche à␣
↪samedi),
# |    |    |    |           ou utilisez des noms)
# |    |    |    \------------ mois (1 - 12)
# |    |    \----------------- jour du mois (1 - 31)
# |    \-------------------- heures (0 - 23)
# \----------------------------- minutes (0 - 59)
```

Vous pouvez avoir plus d'infos ici : https://fr.wikipedia.org/wiki/Cron

Astuce

Utilisez `-q` (or *–quiet*) pour ne pas afficher de sortie pour les cronjobs.

Tâches Cron sur des Serveurs Mutualisés

Sur certains serveurs mutualisés `cd /chemin/complet/vers/la/racine && bin/cake macommande monparam` pourrait ne pas fonctionner. Vous pouvez à la place utiliser `php /chemin/complet/vers/la/racine/bin/cake.php macommande monparam`.

Note

register_argc_argv a besoin d'être activé en incluant `register_argc_argv = 1` dans votre php.ini. Si vous ne pouvez pas changer register_argc_argv de manière globale, vous pouvez préciser à la tâche cron d'utiliser votre propre configuration en la spécifiant via le paramètre `-d register_argc_argv=1`. Exemple : `php -d register_argc_argv=1 /chemin/complet/vers/la/racine/bin/cake.php macommande monparam`.

Commandes Fournies par CakePHP

Shell Cache

Pour vous aider à mieux gérer les données mises en cache dans un environnement CLI, une commande shell a été ajoutée qui montre les méthodes pour effacer les données mises en cache :

```
// Efface une config mise en cache
bin/cake cache clear <configname>

// Efface toutes les configs mises en cache
bin/cake cache clear_all
```

Complétion du Shell

Travailler avec la console donne au développeur beaucoup de possibilités mais devoir complètement connaître et écrire ces commandes peut être fastidieux. Spécialement lors du développement de nouveaux shells où les commandes diffèrent à chaque itération. Les Shells de complétion aident à ce niveau-là en fournissant une API pour écrire les scripts de complétion pour les shells comme bash, zsh, fish etc…

Sous-Commandes

Les Shells de complétion se composent d'un certain nombre de sous-commandes qui permettent au développeur de créer son propre script de complétion. Chacun pour une étape différente dans le processus d'autocomplétion.

Commandes

Pour les premières étapes, les commandes affichent les Commandes de Shell disponibles, y compris le nom du plugin quand il est valable. (Toutes les possibilités retournées, pour celle-ci et les autres sous-commandes, sont séparées par un espace.) Par exemple :

```
bin/cake Completion commands
```

Retourne :

```
acl api bake command_list completion console i18n schema server test testsuite upgrade
```

Votre script de complétion peut sélectionner les commandes pertinentes de cette liste pour continuer avec. (Pour celle-là et les sous-commandes suivantes.)

Sous-Commandes

Une fois que la commande préférée a été choisie, les Sous-commandes apparaissent à la deuxième étape et affiche la sous-commande possible pour la commande de shell donnée. Par exemple :

```
bin/cake Completion subcommands bake
```

Retourne :

```
controller db_config fixture model plugin project test view
```

Options

En troisième et dernière option, les options de sortie pour une (sous) commande donnée comme définies dans getOptionParser. (Y compris les options par défaut héritées du Shell.) Par exemple :

```
bin/cake Completion options bake
```

Retourne :

```
--help -h --verbose -v --quiet -q --everything --connection -c --force -f --plugin -p --
↪prefix --theme -t
```

Vous pouvez passer un autre argument représentant une sous-commande du shell : cela vous retournera les options spécifiques à cette sous-commande.

Activer l'autocomplétion Bash pour la console CakePHP

Tout d'abord, assurez-vous que la librairie **bash-completion** est installée. Si elle ne l'est pas, vous pouvez le faire en exécutant la commande suivante :

```
apt-get install bash-completion
```

Créez un fichier **cake** dans **/etc/bash_completion.d/** et placez-y le *Contenu du fichier bash d'autocomplétion*.

Sauvegardez le fichier et rédémarrez la console.

> **Note**
>
> Si vous utilisez MacOS X, vous pouvez installer la librairie **bash-completion** en utilisant **homebrew** avec la commande suivante : `brew install bash-completion`. Le répertoire cible du fichier **cake** devra être **/usr/local/etc/bash_completion.d/**.

Contenu du fichier bash d'autocomplétion

Voici le code que vous devez saisir dans le fichier **cake** (préalablement créé au bon emplacement pour bénéficier de l'autocomplétion quand vous utilisez la console CakePHP :

```
#
# Fichier de completion Bash pour la console CakePHP
#

_cake()
{
    local cur prev opts cake
    COMPREPLY=()
    cake="${COMP_WORDS[0]}"
    cur="${COMP_WORDS[COMP_CWORD]}"
    prev="${COMP_WORDS[COMP_CWORD-1]}"

    if [[ "$cur" == -* ]] ; then
        if [[ ${COMP_CWORD} = 1 ]] ; then
            opts=$(${cake} Completion options)
        elif [[ ${COMP_CWORD} = 2 ]] ; then
            opts=$(${cake} Completion options "${COMP_WORDS[1]}")
        else
            opts=$(${cake} Completion options "${COMP_WORDS[1]}" "${COMP_WORDS[2]}")
        fi

        COMPREPLY=( $(compgen -W "${opts}" -- ${cur}) )
        return 0
    fi

    if [[ ${COMP_CWORD} = 1 ]] ; then
        opts=$(${cake} Completion commands)
        COMPREPLY=( $(compgen -W "${opts}" -- ${cur}) )
        return 0
    fi
```

(suite sur la page suivante)

(suite de la page précédente)

```
    if [[ ${COMP_CWORD} = 2 ]] ; then
        opts=$(${cake} Completion subcommands $prev)
        COMPREPLY=( $(compgen -W "${opts}" -- ${cur}) )
        if [[ $COMPREPLY = "" ]] ; then
            _filedir
            return 0
        fi
        return 0
    fi

    opts=$(${cake} Completion fuzzy "${COMP_WORDS[@]:1}")
    COMPREPLY=( $(compgen -W "${opts}" -- ${cur}) )
    if [[ $COMPREPLY = "" ]] ; then
        _filedir
        return 0
    fi
    return 0;
}

complete -F _cake cake bin/cake
```

Utilisez l'autocompletion

Une fois activée, l'autocompletion peut être utilisée de la même manière que pour les autres commandes natives du système, en utilisant la touche **TAB**. Trois types d'autocompletion sont fournis. Les examples de retour qui suivent proviennent d'une installation fraîche de CakePHP.

Commandes

Exemple de rendu pour l'autocomplétion des commandes :

```
$ bin/cake <tab>
bake        i18n        schema_cache   routes
console     migrations  plugin         server
```

Sous-commandes

Exemple de rendu pour l'autocomplétion des sous-commandes :

```
$ bin/cake bake <tab>
behavior            helper              shell
cell                mailer              shell_helper
component           migration           template
controller          migration_snapshot  test
fixture             model
form                plugin
```

Options

Exemple de rendu pour l'autocomplétion des options d'une sous-commande :

```
$ bin/cake bake -<tab>
-c            --everything  --force       --help        --plugin      -q            -t  ␣
↪          -v
--connection  -f            -h            -p            --prefix      --quiet       --
↪theme       --verbose
```

Shell I18N

Les fonctionnalités i18n de CakePHP utilisent les fichiers po [143] comme source de traduction. Cela les rend facile à intégrer avec des outils tels que Poedit [144] ou d'autres outils habituels de traduction.

Le Shell i18n est un moyen rapide de générer des fichiers de template po. Les fichiers de template peuvent être donnés aux traducteurs afin qu'ils traduisent les chaînes de caractères dans votre application. Une fois que votre traduction est faîte, les fichiers pot peuvent être fusionnés avec les traductions existantes pour aider la mise à jour de vos traductions.

Générer les Fichiers POT

Les fichiers POT peuvent être générés pour une application existante en utilisant la commande `extract`. Cette commande va scanner toutes les fonctions de type `__()` de l'ensemble de votre application et extraire les chaînes de caractères. Chaque chaîne unique dans votre application sera combinée en un seul fichier POT :

```
bin/cake i18n extract
```

La commande ci-dessus va lancer le shell d'extraction. Le résultat de cette commande va être la création du fichier **src/Locale/default.pot**. Vous utilisez le fichier pot comme un template pour créer les fichiers po. Si vous créez manuellement les fichiers po à partir du fichier pot, pensez à bien corriger le `Plural-Forms` de la ligne d'en-tête.

Générer les Fichiers POT pour les Plugins

Vous pouvez générer un fichier POT pour un plugin spécifique en faisant :

```
bin/cake i18n extract --plugin <Plugin>
```

Cela générera les fichiers POT requis utilisés dans les plugins.

Extraire de plusieurs dossiers à la fois

Vous pouvez parfois avoir besoin d'extraire des chaînes depuis plus d'un dossier de votre application. Par exemple, si vous définissez des chaînes à traduire dans le dossier `config` de votre application, vous voudrez probablement extraire les chaînes de ce dossier en plus de celles du dossier `src`. Vous pouvez le faire en utilisant l'option `--paths`. Elle accepte une liste de chemins absolus séparés par une virgule :

```
bin/cake i18n extract --paths /var/www/app/config,/var/www/app/src
```

Exclure les fichiers

Vous pouvez passer une liste de dossiers séparées par une virgule que vous souhaitez exclure. Tout chemin contenant une partie de chemin avec les valeurs fournies sera ignoré :

```
bin/cake i18n extract --exclude Test,Vendor
```

143. https://fr.wikipedia.org/wiki/GNU_gettext
144. https://www.poedit.net/

Eviter l'Écrasement des Avertissements pour les Fichiers POT Existants

En ajoutant `--overwrite`, le script de shell ne va plus vous avertir si un fichier POT existe déjà et va écraser par défaut :

```
bin/cake i18n extract --overwrite
```

Extraire les Messages des Librairies du Cœur de CakePHP

Par défaut le script de shell d'extraction va vous demander si vous souhaitez extraire les messages utilisés dans les librairies du cœur de CakePHP. Définissez `--extract-core` à yes ou no pour définir le comportement par défaut :

```
bin/cake i18n extract --extract-core yes

// ou

bin/cake i18n extract --extract-core no
```

Shell Plugin

Le shell plugin vous permet de charger et décharger les plugins avec le prompteur de commandes. Si vous avez besoin d'aide, lancez :

```
bin/cake plugin --help
```

Charger les Plugins

Avec la tâche `Load` vous pouvez charger les plugins dans votre **config/bootstrap.php**. Vous pouvez aussi le faire en lançant :

```
bin/cake plugin load MyPlugin
```

Ceci va ajouter ce qui suit dans votre **config/bootstrap.php** :

```
Plugin::load('MyPlugin');
```

Ajouter `-r` ou `-b` à la tâche de chargement va activer le chargement des valeurs `bootstrap` et `routes` du plugin :

```
bin/cake plugin load -b MyPlugin

// Charge le bootstrap.php du plugin
Plugin::load('MyPlugin', ['bootstrap' => true]);

bin/cake plugin load -r MyPlugin

// Charge le routes.php du plugin
Plugin::load('MyPlugin', ['routes' => true]);
```

Si vous chargez un plugin qui ne fournit que des outils CLI - comme bake - vous pouvez mettre à jour votre fichier `bootstrap_cli.php` avec :

```
bin/cake plugin load --cli MyPlugin
bin/cake plugin unload --cli MyPlugin
```

Décharger les Plugins

Vous pouvez décharger un plugin en spécifiant son nom :

```
bin/cake plugin unload MyPlugin
```

Ceci va retirer la ligne `Plugin::load('MyPlugin',...)` de votre **config/bootstrap.php**.

Assets des Plugins

CakePHP sert par défaut les assets des plugins en utilisant le filtre de dispatcher `AssetFilter`. Bien que ce soit pratique, il est recommandé de faire des liens symboliques / copier les assets des plugins dans le dossier webroot de l'application pour qu'ils puissent être directement servis par le serveur web dans invoquer PHP. Vous pouvez faire ceci en lançant :

```
bin/cake plugin_assets symlink
```

Lancer la commande ci-dessus va faire faire un lien symbolique pour tous les assets des plugins dans le dossier webroot de l'application. Sur Windows, qui ne supporte pas les liens symboliques, les assets seront copiés dans les dossiers respectifs plutôt que mis en liens symboliques.

Vous pouvez faire des liens symboliques des assets d'un plugin en particulier en spécifiant son nom :

```
bin/cake plugin_assets symlink MyPlugin
```

Shell Routes

RoutesShell fournit une interface CLI simple d'utilisation pour tester et débugger les routes. Vous pouvez l'utiliser pour tester la façon dont les routes sont parsées et ce que les paramètres de routing des URLs vont générer.

Récupérer une Liste de Toutes les Routes

```
bin/cake routes
```

Tester le parsing de l'URL

Vous pouvez rapidement voir comment une URL sera parsée en utilisant la méthode `check` :

```
bin/cake routes check /bookmarks/edit/1
```

Si votre route contient un paramètre de query string, n'oubliez pas d'entourer l'URL de guillemets :

```
bin/cake routes check "/bookmarks/?page=1&sort=title&direction=desc"
```

Tester la Génération d'URL

Vous pouvez regarder la façon dont un *tableau de routing* va générer l'URL en utilisant la méthode `generate` :

```
bin/cake routes generate controller:Bookmarks action:edit 1
```

Serveur Shell

`ServerShell` vous permet de créer un serveur web simple en utilisant le serveur web de PHP. Bien que ce serveur **ne** soit **pas** fait pour une utilisation en production, il peut être pratique en développement quand vous voulez rapidement essayer une idée et ne voulez pas passer du temps à configurer Apache ou Nginx. Vous pouvez démarrer le serveur shell avec :

```
$ bin/cake server
```

Vous pourrez voir le serveur démarré sur le port 8765. Vous pouvez visiter le sever CLI en visitant `http://localhost:8765` dans votre navigateur. Vous pouvez fermer le serveur en tapant `CTRL-C` dans votre terminal.

> **Note**
>
> Essayez `bin/cake server -H 0.0.0.0` si le serveur est inaccessible depuis d'autres hôtes.

Changer le Port et le Document Root

Vous pouvez personnaliser le port et le document root en utilisant les options :

```
$ bin/cake server --port 8080 --document_root path/to/app
```

Console Interactive (REPL)

Le squelette de l'application CakePHP intègre un REPL(Read Eval Print Loop = Lire Evaluer Afficher Boucler) qui permet l'exploration de CakePHP et de votre application avec une console interactive. Vous pouvez commencer la console interactive en utilisant :

```
$ bin/cake console
```

Cela va démarrer votre application et lancer une console interactive. A ce niveau-là, vous pouvez interagir avec le code de votre application et exécuter des requêtes en utilisant les models de votre application :

```
$ bin/cake console

Welcome to CakePHP v3.0.0 Console
---------------------------------------------------------------
App : App
Path: /Users/mark/projects/cakephp-app/src/
---------------------------------------------------------------
>>> $articles = Cake\ORM\TableRegistry::getTableLocator()->get('Articles');
// object(Cake\ORM\Table)(
//
// )
>>> $articles->find()->all();
```

Puisque votre application a été démarrée, vous pouvez aussi tester le routing en utilisant le REPL :

```
>>> Cake\Routing\Router::parse('/articles/view/1');
// [
//   'controller' => 'Articles',
//   'action' => 'view',
//   'pass' => [
//     0 => '1'
//   ],
//   'plugin' => NULL
// ]
```

Vous pouvez aussi tester la génération d'URL :

```
>>> Cake\Routing\Router::url(['controller' => 'Articles', 'action' => 'edit', 99]);
// '/articles/edit/99'
```

Pour quitter le REPL, vous pouvez utiliser `CTRL-C` ou en tapant `exit`.

Shells

class Cake\Console**Shell**

Obsolète depuis la version 3.6.0 : Les shells sont dépréciés depuis la version 3.6.0, mais ne sera pas retiré avant la version 5.x.

Créer un Shell

Créons un shell for use in the Console. For this example, we'll create a simple Hello world shell. In your application's **src/Shell** directory create **HelloShell.php**. Put the following code inside it :

```
namespace App\Shell;

use Cake\Console\Shell;

class HelloShell extends Shell
{
    public function main()
    {
        $this->out('Hello world.');
    }
}
```

The conventions for shell classes are that the class name should match the file name, with the suffix of Shell. In our shell we created a `main()` method. This method is called when a shell is called with no additional commands. We'll add some more commands in a bit, but for now let's just run our shell. From your application directory, run :

```
bin/cake hello
```

You should see the following output :

```
Hello world.
```

As mentioned before, the `main()` method in shells is a special method called whenever there are no other commands or arguments given to a shell. Since our main method wasn't very interesting let's add another command that does something :

```
namespace App\Shell;

use Cake\Console\Shell;

class HelloShell extends Shell
{
    public function main()
    {
        $this->out('Hello world.');
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    public function heyThere($name = 'Anonymous')
    {
        $this->out('Hey there ' . $name);
    }
}
```

After saving this file, you should be able to run the following command and see your name printed out :

```
bin/cake hello hey_there your-name
```

Any public method not prefixed by an _ is allowed to be called from the command line. As you can see, methods invoked from the command line are transformed from the underscored shell argument to the correct camel-cased method name in the class.

In our `heyThere()` method we can see that positional arguments are provided to our `heyThere()` function. Positional arguments are also available in the `args` property. You can access switches or options on shell applications, which are available at `$this->params`, but we'll cover that in a bit.

When using a `main()` method you won't be able to use the positional arguments. This is because the first positional argument or option is interpreted as the command name. If you want to use arguments, you should use method names other than `main`.

Shell Tasks

There will be times when building more advanced console applications, you'll want to compose functionality into re-usable classes that can be shared across many shells. Tasks allow you to extract commands into classes. For example the `bake` command is made almost entirely of tasks. You define a tasks for a shell using the `$tasks` property :

```
class UserShell extends Shell
{
    public $tasks = ['Template'];
}
```

You can use tasks from plugins using the standard *syntaxe de plugin*. Tasks are stored in `Shell/Task/` in files named after their classes. So if we were to create a new "FileGenerator" task, you would create **src/Shell/Task/FileGeneratorTask.php**.

Each task must at least implement a `main()` method. The ShellDispatcher, will call this method when the task is invoked. A task class looks like :

```
namespace App\Shell\Task;

use Cake\Console\Shell;

class FileGeneratorTask extends Shell
{
    public function main()
    {

    }
}
```

A shell can also access its tasks as properties, which makes tasks great for making re-usable chunks of functionality similar to *Components (Composants)* :

```
// Found in src/Shell/SeaShell.php
class SeaShell extends Shell
{
    // Found in src/Shell/Task/SoundTask.php
    public $tasks = ['Sound'];

    public function main()
    {
        $this->Sound->main();
    }
}
```

You can also access tasks directly from the command line :

```
$ cake sea sound
```

> **Note**
>
> In order to access tasks directly from the command line, the task **must** be included in the shell class" $tasks property.

Also, the task name must be added as a sub-command to the Shell's OptionParser :

```
public function getOptionParser()
{
    $parser = parent::getOptionParser();
    $parser->addSubcommand('sound', [
        // Provide help text for the command list
        'help' => 'Execute The Sound Task.',
        // Link the option parsers together.
        'parser' => $this->Sound->getOptionParser(),
    ]);

    return $parser;
}
```

Loading Tasks On The Fly with TaskRegistry

You can load tasks on the fly using the Task registry object. You can load tasks that were not declared in $tasks this way :

```
$project = $this->Tasks->load('Project');
```

Would load and return a ProjectTask instance. You can load tasks from plugins using :

```
$progressBar = $this->Tasks->load('ProgressBar.ProgressBar');
```

Using Models in Your Shells

You'll often need access to your application's business logic in shell utilities. You can load models in shells, just as you would in a controller using `loadModel()`. The loaded models are set as properties attached to your shell : :

```
namespace App\Shell;

use Cake\Console\Shell;

class UserShell extends Shell
{

    public function initialize(): void
    {
        parent::initialize();
        $this->loadModel('Users');
    }

    public function show()
    {
        if (empty($this->args[0])) {
            return $this->abort('Please enter a username.');
        }
        $user = $this->Users->findByUsername($this->args[0])->first();
        $this->out(print_r($user, true));
    }
}
```

The above shell, will fetch a user by username and display the information stored in the database.

Shell Helpers

If you have complex output generation logic, you can use */console-commands/helpers* to encapsulate this logic in a re-usable way.

Invoking Other Shells from Your Shell

`Cake\Console\Shell::dispatchShell(`*$args*`)`

There are still many cases where you will want to invoke one shell from another though. `Shell::dispatchShell()` gives you the ability to call other shells by providing the `argv` for the sub shell. You can provide arguments and options either as var args or as a string :

```
// As a string
$this->dispatchShell('schema create Blog --plugin Blog');

// As an array
$this->dispatchShell('schema', 'create', 'Blog', '--plugin', 'Blog');
```

The above shows how you can call the schema shell to create the schema for a plugin from inside your plugin's shell.

Passing extra parameters to the dispatched Shell

It can sometimes be useful to pass on extra parameters (that are not shell arguments) to the dispatched Shell. In order to do this, you can now pass an array to `dispatchShell()`. The array is expected to have a `command` key as well as an `extra` key :

```
// Using a command string
$this->dispatchShell([
```

(suite sur la page suivante)

(suite de la page précédente)

```
   'command' => 'schema create Blog --plugin Blog',
   'extra' => [
        'foo' => 'bar'
    ]
]);

// Using a command array
$this->dispatchShell([
   'command' => ['schema', 'create', 'Blog', '--plugin', 'Blog'],
   'extra' => [
        'foo' => 'bar'
    ]
]);
```

Parameters passed through `extra` will be merged in the `Shell::$params` property and are accessible with the `Shell::param()` method. By default, a `requested` extra param is automatically added when a Shell is dispatched using `dispatchShell()`. This `requested` parameter prevents the CakePHP console welcome message from being displayed on dispatched shells.

Parsing CLI Options

Shells use *Option Parsers* to define their options, arguments and automate help generation.

Interacting with Input/Output

Shells allow you to access a `ConsoleIo` instance via the `getIo()` method. See the *Entrée/Sortie de Commande* section for more information.

In addition to the `ConsoleIo` object, Shell classes offer a suite of shortcut methods. These methods are shortcuts and aliases to those found on `ConsoleIo` :

```
// Get arbitrary text from the user.
$color = $this->in('What color do you like?');

// Get a choice from the user.
$selection = $this->in('Red or Green?', ['R', 'G'], 'R');

// Create a file
$this->createFile('bower.json', $stuff);

// Write to stdout
$this->out('Normal message');

// Write to stderr
$this->err('Error message');

// Write to stderr and raise a stop exception
$this->abort('Fatal error');
```

It also provides two convenience methods regarding the output level :

```
// Would only appear when verbose output is enabled (-v)
$this->verbose('Verbose message');
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Would appear at all levels.
$this->quiet('Quiet message');
```

Shell also includes methods for clearing output, creating blank lines, or drawing a line of dashes :

```
// Output 2 newlines
$this->out($this->nl(2));

// Clear the user's screen
$this->clear();

// Draw a horizontal line
$this->hr();
```

Stopping Shell Execution

When your shell commands have reached a condition where you want execution to stop, you can use `abort()` to raise a `StopException` that will halt the process :

```
$user = $this->Users->get($this->args[0]);
if (!$user) {
    // Halt with an error message and error code.
    $this->abort('User cannot be found', 128);
}
```

Status and Error Codes

Command-line tools should return 0 to indicate success, or a non-zero value to indicate an error condition. Since PHP methods usually return `true` or `false`, the Cake Shell `dispatch` function helps to bridge these semantics by converting your `null` and `true` return values to 0, and all other values to 1.

The Cake Shell `dispatch` function also catches the `StopException` and uses its exception code value as the shell's exit code. As described above, you can use the `abort()` method to print a message and exit with a specific code, or raise the `StopException` directly as shown in the example :

```
namespace App\Shell\Task;

use Cake\Console\Shell;

class ErroneousShell extends Shell
{
    public function main()
    {
        return true;
    }

    public function itFails()
    {
        return false;
    }

    public function itFailsSpecifically()
```

(suite sur la page suivante)

(suite de la page précédente)

```
    {
        throw new StopException("", 2);
    }
}
```

The example above will return the following exit codes when executed on a command-line :

```
$ bin/cake erroneousshell ; echo $?
0
$ bin/cake erroneousshell itFails ; echo $?
1
$ bin/cake erroneousshell itFailsSpecifically ; echo $?
2
```

Astuce

Avoid exit codes 64 - 78, as they have specific meanings described by `sysexits.h`. Avoid exit codes above 127, as these are used to indicate process exit by signal, such as SIGKILL or SIGSEGV.

Note

You can read more about conventional exit codes in the sysexit manual page on most Unix systems (`man sysexits`), or the `System Error Codes` help page in Windows.

Hook Methods

`Cake\Console\Shell::initialize()`

Initializes the Shell, acts as constructor for subclasses and allows configuration of tasks prior to shell execution.

`Cake\Console\Shell::startup()`

Starts up the Shell and displays the welcome message. Allows for checking and configuring prior to command or main execution.

Astuce

Override the `startup()` method if you want to remove the welcome information, or otherwise modify the pre-command flow.

Avoid exit codes 64 - 78, as they have specific meanings described by `sysexits.h`. Avoid exit codes above 127, as these are used to indicate process exit by signal, such as SIGKILL or SIGSEGV.

Routage dans l'Environnement de la Console

En ligne de commande (CLI), et spécifiquement dans vos shells et vos tâches, `env('HTTP_HOST')` et les autres variables d'environnement spécifiques au serveur web ne sont pas définies.

Si vous générez des rapports ou si vous envoyez des mails qui utilisent00 `Router::url()`, ils contiendront l'hôte par défaut `http://localhost/`, et donc ils produiront des URL invalides. Dans un tel cas, vous devez spécifier le domaine manuellement. Vous pouvez utiliser pour cela la valeur `App.fullBaseUrl` de Configure depuis votre bootstrap ou votre configuration, par exemple.

Pour envoyer des mails, vous devrez fournir une classe Email avec l'hôte à partir duquel vous voulez envoyer le mail :

```
use Cake\Mailer\Email;

$email = new Email();
$email->setDomain('www.example.org');
```

Cela suppose que les IDs du message généré soient valides et correspondent au domaine à partir duquel les mails sont envoyés.

CHAPITRE 16

Debugger

Le debug est une inévitable et nécessaire partie de tout cycle de développement. Tandis que CakePHP n'offre pas d'outils qui se connectent directement avec tout IDE ou éditeur, CakePHP fournit plusieurs outils pour l'aide au debug et ce qui est lancé sous le capot de votre application.

Debug Basique

debug(*mixed $var*, *boolean $showHtml = null*, *$showFrom = true*)

La fonction `debug()` est une fonction disponible partout qui fonctionne de la même manière que la fonction PHP `print_r()`. La fonction `debug()` vous permet de montrer les contenus d'un variable de différentes façons. Premièrement, si vous voulez que vos données soient montrées d'une façon sympa en HTML, définissez le deuxième paramètre à `true`. La fonction affiche aussi la ligne et le fichier dont ils sont originaires par défaut.

La sortie de cette fonction est seulement montrée si la variable de `$debug` du cœur a été définie à `true`.

stackTrace()

La fonction `stackTrace()` est globalement disponible, et vous permet d'afficher une stack trace quelque soit la fonction appelée.

breakpoint()

Si vous avez installé Psysh [145] vous pouvez utiliser cette fonction dans les environnements CLI pour ouvrir une console interactive avec le scope local courant :

```
// Du code
eval(breakpoint());
```

Ouvrira une console interactive qui peut être utilisée pour vérifier les variables locales et exécuter d'autre code. Vous pouvez fermer le debugger interactif et reprendre l'exécution du script original en tapant `quit` ou `q` dans la session interactive.

145. https://psysh.org/

Utiliser la Classe Debugger

class Cake\Error**Debugger**

Pour utiliser le debugger, assurez-vous d'abord que `Configure::read('debug')` est défini à `true`.

Affichage des Valeurs

static Cake\Error\Debugger::**dump**(*$var*, *$depth = 3*)

Dump affiche le contenu d'une variable. Elle affiche toutes les propriétés et méthodes (s'il y en a) de la variable fournie : :

```
$foo = [1,2,3];

Debugger::dump($foo);

// Outputs
[
    1,
    2,
    3
]

// Simple object
$car = new Car();

Debugger::dump($car);

// Outputs
object(Car) {
    color => 'red'
    make => 'Toyota'
    model => 'Camry'
    mileage => (int)15000
}
```

Masquer des Données

Lorsque vous affichez des données avec `Debugger` ou que des pages d'erreurs sont affichées, vous pouvez souhaiter masquer des données sensibles comme des mots de passes ou des clés d'API. Dans votre fichier `config/bootstrap.php`, vous pouvez spécifier les clés à masquer :

```
Debugger::setOutputMask([
    'password' => 'xxxxx',
    'awsKey' => 'yyyyy',
]);
```

Logging With Stack Traces

static Cake\Error\Debugger::**log**(*$var*, *$level = 7*, *$depth = 3*)

Crée un stack trace log détaillé au moment de l'invocation. La méthode `log()` affiche les données identiques à celles faites par `Debugger::dump()`, mais dans debug.log au lieu de les sortir buffer. Notez que votre répertoire **tmp** (et son contenu) doit être ouvert en écriture par le serveur web pour que le `log()` fonctionne correctement.

Generating Stack Traces

static Cake\Error\Debugger::**trace**(*$options*)

Retourne le stack trace courant. Chaque ligne des traces inclut la méthode appelée, incluant chaque fichier et ligne d'où est originaire l'appel :

```
//Dans PostsController::index()
pr( Debugger::trace() );

//sorties
PostsController::index() - APP/Controller/DownloadsController.php, line 48
Dispatcher::_invoke() - CORE/lib/Cake/Routing/Dispatcher.php, line 265
Dispatcher::dispatch() - CORE/lib/Cake/Routing/Dispatcher.php, line 237
[main] - APP/webroot/index.php, line 84
```

Ci-dessus se trouve le stack trace généré en appelant `Debugger::trace()` dans une action d'un controller. Lire le stack trace de bas en haut montre l'ordre des fonctions lancées actuellement (stack frames).

Getting an Excerpt From a File

static Cake\Error\Debugger::**excerpt**(*$file*, *$line*, *$context*)

Récupérer un extrait du fichier dans $path (qui est un chemin de fichier absolu), mettant en évidence le numéro de la ligne $line avec le nombre de lignes $context autour :

```
pr( Debugger::excerpt(ROOT.DS.LIBS.'debugger.php', 321, 2) );

//sortira ce qui suit.
Array
(
    [0] => <code><span style="color: #000000"> * @access public</span></code>
    [1] => <code><span style="color: #000000"> */</span></code>
    [2] => <code><span style="color: #000000">    function excerpt($file, $line,
↪$context = 2) {</span></code>
    [3] => <span class="code-highlight"><code><span style="color: #000000">        $data␣
↪= $lines = [];</span></code></span>
    [4] => <code><span style="color: #000000">        $data = @explode("\n", file_get_
↪contents($file));</span></code>
)
```

Bien que cette méthode est utilisée en interne, elle peut être pratique si vous créez vos propres messages d'erreurs ou les logs pour les situations personnalisées.

static Cake\Error\Debugger::**getType**(*$var*)

Récupère le type de variable. Les objets retourneront leur nom de classe.

Utiliser les Logs pour Debugger

Logger des messages est une autre bonne façon de debugger les applications, et vous pouvez utiliser *Cake\Log\Log* pour faire le logging dans votre application. Tous les objets qui utilisent `LogTrait` ont une méthode d'instanciation `log()` qui peut être utilisée pour logger les messages :

```
$this->log('Got here', 'debug');
```

Ce qui est au-dessus écrit `Got here` dans le log de debug. Vous pouvez utiliser les logs (log entries) pour faciliter le debug des méthodes qui impliquent des redirections ou des boucles compliquées. Vous pouvez aussi utiliser `Cake\Log\Log::write()`` pour écrire les messages de log. Cette méthode peut être appelée statiquement partout dans votre application où Log a été chargée :

```
// Au début du fichier dans lequel vous voulez logger.
use Cake\Log\Log;

// N'importe où Log a été importé
Log::debug('Got here');
```

Kit de Debug

DebugKit est un plugin qui fournit un nombre de bons outils de debug. Il fournit principalement une barre d'outils dans le HTML rendu, qui fournit une pléthore d'informations sur votre application et la requête courante. Consultez le chapitre sur *Debug Kit* pour plus d'information sur son installation et son utilisation.

CHAPITRE 17

Déploiement

Une fois que votre application CakePHP est prête à être déployée, il reste un certains nombre de choses à faire.

Déplacer les Fichiers

Vous pouvez cloner votre dépôt sur votre serveur de production, puis faire un checkout du commit/tag que vous voulez lancer. Puis, exécutez `composer install`. Bien que cela nécessite quelques connaissances de git et que vous ayez `git` et `composer` installés, cette façon de faire vous permettra de gérer les dépendances de librairies et les permissions des fichiers et des dossiers.

Rappelez-vous que lors d'un déploiement via FTP, vous devrez au moins mettre les bonnes permissions pour les fichiers et les dossiers.

Vous pouvez aussi utiliser cette technique de déploiement pour configurer des versions staging ou demo (pre-production) et les garder à jour avec votre environnement local.

Ajuster la Configuration

Vous voudrez faire quelques ajustements à la configuration de votre application pour un environnement de production. La valeur de `debug` est extrêmement importante. Mettre debug = `false` désactive un certain nombre de fonctionnalités de développement qui ne devraient jamais être exposées sur internet. Désactiver le debug change les types de choses suivantes :

- Les messages de Debug, créés avec *`pr()`*, *`debug()`* et *`dd()`* sont désactivés.
- Les caches du Cœur de CakePHP sont flushés tous les ans (environ 365 jours), au lieu de toutes les 10 secondes en développement.
- Les vues d'Erreur sont moins informatives, et renvoient des messages génériques d'erreur à la place.
- Les Erreurs PHP ne sont pas affichées.
- Les traces de pile d'Exception sont désactivées.

En plus des éléments ci-dessus, beaucoup de plugins et d'extensions d'application utilisent `debug` pour modifier leur comportement.

Vous pouvez créer une variable d'environnement pour définir le niveau de debug dynamiquement entre plusieurs environnements. Cela va éviter de déployer une application avec debug à `true` et vous permet de ne pas avoir à changer de niveau de debug chaque fois avant de déployer vers un environnement de production.

Par exemple, vous pouvez définir une variable d'environnement dans votre configuration Apache :

```
SetEnv CAKEPHP_DEBUG 1
```

Et ensuite vous pouvez définir le niveau de debug dynamiquement dans **config/app_local.php** :

```
$debug = (bool)getenv('CAKEPHP_DEBUG');

return [
    'debug' => $debug,
    .....
];
```

Vérifier Votre Sécurité

Si vous sortez votre application dans la nature, il est bon de vous assurer qu'elle n'a pas de fuites :

- Assurez-vous que vous utilisez le middleware *Middleware Cross Site Request Forgery (CSRF)* activé.
- Vous pouvez activer le component *SecurityComponent (Sécurité)*. Il évite plusieurs types de form tampering et réduit la possibilité des problèmes de mass-assignment.
- Assurez-vous que vos models ont les bonnes règles de *Validation* activées.
- Vérifiez que seul votre répertoire `webroot` est visible publiquement, et que vos secrets (comme votre sel de app, et toutes les clés de sécurité) sont privées et aussi uniques.

Définir le Document Root

Configurer le document root correctement dans votre application est aussi une étape importante pour garder votre code sécurisé et votre application plus sûre. Les applications CakePHP devraient avoir le document root configuré au répertoire `webroot` de l'application. Cela rend les fichiers de l'application et de configuration inaccessibles via une URL. Configurer le document root est différent selon les webserveurs. Regardez la documentation *Réécriture d'URL* pour avoir des informations sur la spécificité de chaque webserveur.

Dans tous les cas, vous devez définir le document de l'hôte/domaine virtuel pour qu'il soit `webroot/`. Cela retire la possibilité que des fichiers soient exécutés en-dehors du répertoire webroot.

Améliorer les Performances de votre Application

Le chargement des classes peut prendre une bonne part du temps d'exécution de votre application. Afin d'éviter ce problème, il est recommandé que vous lanciez cette commande dans votre serveur de production une fois que l'application est déployée :

```
php composer.phar dumpautoload -o
```

Étant donné que la gestion des éléments statiques, comme les images, le Javascript et les fichiers CSS des plugins à travers le `Dispatcher` est incroyablement inefficace, il est chaudement recommandé d'utiliser les liens symboliques pour la production. Ceci peut être fait facilement en utilisant le shell `plugin` :

```
bin/cake plugin assets symlink
```

La commande ci-dessus va faire un lien symbolique du répertoire `webroot` de tous les plugins chargés vers les chemins appropriés dans le répertoire `webroot` de l'application.

Si votre système de fichier ne permet pas de créer des liens symboliques, les répertoires seront copiés à la place des liens symboliques. Vous pouvez aussi explicitement copier les répertoires en utilisant :

```
bin/cake plugin assets copy
```

Déployer une Mise à Jour

Après un déploiement ou une mise à jour, vous pouvez aussi lancer `bin/cake schema_cache clear`, qui fait parti du shell *Shell du Cache du Schéma*.

À chaque dépoloiement, vous aurez sans doute quelques tâches à coordonner sur votre serveur web. Les plus typiques sont :

1. Installer des dépendances avec `composer install`. Évitez d'utiliser `composer update` en déploiement car vous pourriez obtenir des versions inattendues des packages.
2. Lancez les migrations de bases de données, que ce soit avec le plugin Migrations ou un autre outil.
3. Vider le cache du schéma du modèle avec `bin/cake schema_cache clear`. La section *Shell du Cache du Schéma* vous en apprendra plus sur cette commande.

CHAPITRE 18

Mailer

class Cake\Mailer**Mailer**(*string|array|null $profile = null*)

`Mailer` est une classe utilitaire pour envoyer des emails. Avec cette classe, vous pouvez envoyer des emails depuis n'importe endroit de votre application.

Utilisation basique

Premièrement, vous devez vous assurer que la classe est chargée :

```
use Cake\Mailer\Mailer;
```

Après avoir chargé `Mailer`, vous pouvez envoyer un email de la façon suivante :

```
$mailer = new Mailer('default');
$mailer->setFrom(['moi@example.com' => 'Mon Site'])
    ->setTo('toi@example.com')
    ->setSubject('À propos')
    ->send('Mon message');
```

Les méthodes setter de `Mailer` retournent l'instance de la classe, ce qui fait que vous pouvez définir ses propriétés par un chaînage de méthodes.

`Mailer` comporte plusieurs méthodes pour définir les destinataires - `setTo()`, `setCc()`, `setBcc()`, `addTo()`, `addCc()` et `addBcc()`. La principale différence est que les trois premières méthodes vont écraser ce que vous auriez déjà défini, tandis que les suivantes vont seulement ajouter d'autres destinataires dans leurs champs respectifs :

```
$mailer = new Mailer();
$mailer->setTo('to@example.com', 'To Example');
$mailer->addTo('to2@example.com', 'To2 Example');
// Les destinataires de l'email sont: to@example.com et to2@example.com
```

(suite sur la page suivante)

(suite de la page précédente)

```
$mailer->setTo('test@example.com', 'ToTest Example');
// Le destinataire de l'email est: test@example.com
```

Choisir l'émetteur

Quand on envoie des emails de la part d'une autre personne, il est généralement souhaitable de définir l'émetteur original avec le header Sender. Vous pouvez le faire en utilisant `setSender()` :

```
$mailer = new Mailer();
$mailer->setSender('app@example.com', 'MyApp emailer');
```

> **Note**
>
> Quand vous envoyez un mail de la part d'une autre personne, il est également préférable de définir l'émetteur de l'enveloppe . Cela lui évite de recevoir des messages en cas de problèmes de distribution du mail.

Configuration

Les configurations des profils du Mailer et du transport d'emails sont définies dans les fichiers de configuration de votre application. Les clés `Email` et `MailerTransport` définissent respectivement les configurations des profils du mailer et du transport des emails. Pendant le bootstrap de l'application, les paramètres de configuration sont repris de la classe `Configure` vers les classes `Mailer` et `TransportFactory` en utilisant `setConfig()`. En définissant des profils et des transports, vous pouvez alléger le code de votre application de toutes les données de configuration, et éviter des duplications de code qui vous compliqueraient la maintenance et le déploiement.

Pour charger une configuration prédéfinie, vous pouvez utiliser la méthode `setProfile()` ou la passer au constructeur de `Mailer` :

```
$mailer = new Mailer();
$mailer->setProfile('default');

// ou dans le constructeur::
$mailer = new Mailer('default');
```

Plutôt que de passer une chaîne avec le nom d'une configuration prédéfinie, vous pouvez aussi charger tout simplement un tableau d'options :

```
$mailer = new Mailer();
$mailer->setProfile(['from' => 'moi@example.org', 'transport' => 'transport_perso']);

// ou dans le constructeur::
$mailer = new Mailer(['from' => 'moi@example.org', 'transport' => 'transport_perso']);
```

Profils de Configurations

Le fait de définir des profils d'envoi vous permet de consolider les paramètres habituels des emails dans des profils réutilisables. Votre application peut avoir autant de profils que nécessaire. Voici les clés de configuration utilisées :

- `'from'` : Mailer ou un tableau d'expéditeur. Voir `Mailer::setFrom()`.
- `'sender'` : Mailer ou un tableau d'expéditeur réel. Voir `Mailer::setSender()`.
- `'to'` : Mailer ou un tableau de destinataires. Voir `Mailer::setTo()`.
- `'cc'` : Mailer ou un tableau de destinataires en copie. Voir `Mailer::setCc()`.

- `'bcc'` : Mailer ou un tableau de destinataires en copie cachée. Voir `Mailer::setBcc()`.
- `'replyTo'` : Mailer ou un tableau d'adresse de réponse. Voir `Mailer::setReplyTo()`.
- `'readReceipt'` : Adresse Mailer ou un tableau d'adresse pour recevoir un accusé de réception. Voir `Mailer::setReadReceipt()`.
- `'returnPath'` : Adresse Mailer ou un tableau d'adresse à laquelle écrire en cas d'erreur. Voir `Mailer::setReturnPath()`.
- `'messageId'` : ID du message de l'e-mail. Voir `Mailer::setMessageId()`.
- `'subject'` : Sujet du message. Voir `Mailer::setSubject()`.
- `'message'` : Contenu du message. Ne définissez pas ce champ si vous utilisez un contenu rendu par l'application.
- `'priority'` : Nombre exprimant la priorité de l'email (généralement de 1 à 5, 1 étant la priorité la plus haute).
- `'headers'` : En-têtes à inclure. Voir `Mailer::setHeaders()`.
- `'viewRenderer'` : Si vous utilisez un contenu généré par l'application, définissez ici le nom de classe de la vue. Voir `ViewBuilder::setClassName()`.
- `'template'` : Si vous utilisez un contenu généré par l'application, définissez ici le nom du template. Voir `Mailer::setTemplate()`.
- `'theme'` : Thème utilisé pour le rendu du template. Voir `Mailer::setTheme()`.
- `'layout'` : Si vous utilisez un contenu généré par l'application, définissez ici le layout à rendre. Voir `ViewBuilder::setTemplate()`.
- `'autoLayout'` : Si vous voulez rendre un template sans layout, définissez ce champ à `false`. Voir `ViewBuilder::disableAutoLayout()`.
- `'viewVars'` : Si vous utilisez un contenu généré par l'application, définissez ici le tableau contenant les variables devant être rendues dans la vue. Voir `Mailer::setViewVars()`.
- `'attachments'` : Liste des pièces jointes. Voir `Mailer::setAttachments()`.
- `'emailFormat'` : Format de l'email (html, text ou both). Voir `Mailer::setEmailFormat()`.
- `'transport'` : Nom de la configuration du transport. Voir *Configurer les Transports*.
- `'log'` : Niveau de log pour la journalisation des headers et du message. `true` utilisera LOG_DEBUG. Voir *Utiliser les Niveaux*. Notez que les logs seront émis sous le scope nommé `email`. Voir aussi *Scopes de Journalisation*.
- `'helpers'` : Tableau de helpers utilisés dans le template de l'email. `ViewBuilder::setHelpers()`/`ViewBuilder::addHelpers()`

Note

Les valeurs des clés ci-dessus qui utilisent Mailer ou un tableau, telles que from, to, cc, etc, seront passées comme premier paramètre des méthodes correspondantes. L'équivalent de `Mailer::setFrom('mon@example.com', 'Mon Site')` pourrait être défini par `'from' => ['mon@example.com' => 'Mon Site']` dans votre configuration.

Définir les Headers

Dans `Mailer`, vous êtes libre de définir les headers que vous souhaitez. N'oubliez pas d'ajouter le préfixe `X-` pour vos headers personnalisés.

Voir `Mailer::setHeaders()` et `Mailer::addHeaders()`

Envoyer des Mails d'après un Template

Les mails sont souvent bien plus que de simples messages avec du texte. Pour vous faciliter la vie, CakePHP propose d'envoyer des emails en utilisant la *couche de rendu* de CakePHP.

Les templates pour les emails se trouvent dans le dossier spécial `templates/email` de votre application. Les vues d'emails peuvent aussi utiliser des layouts et des éléments, comme les vues normales :

```
$mailer = new Mailer();
$mailer
    ->setEmailFormat('html')
    ->setTo('bob@example.com')
    ->setFrom('app@domain.com')
    ->viewBuilder()
        ->setTemplate('bienvenue')
        ->setLayout('sympa');

$mailer->deliver();
```

Ceci utilisera **templates/email/html/bienvenue.php** comme vue, et **templates/layout/email/html/sympa.php** comme layout. Vous pouvez aussi envoyer des emails templatés multipart :

```
$mailer = new Mailer();
$mailer
    ->setTemplate('bienvenue', 'sympa')
    ->setMailerFormat('both')
    ->setTo('bob@example.com')
    ->setFrom('app@domain.com')
    ->send();

$mailer->deliver();
```

Ce qui utiliserait les fichiers de template suivants :

- **templates/email/text/bienvenue.php**
- **templates/layout/email/text/sympa.php**
- **templates/email/html/bienvenue.php**
- **templates/layout/email/html/sympa.php**

Quand vous envoyez des emails templatés, vous avez la possibilité d'envoyer en `text`, `html` ou `both`.

Vous pouvez définir toute la configuration se rapportant à la vue à partir du *view builder* obtenu par `Mailer::viewBuilder()`, comme vous le feriez dans un controller.

Vous pouvez définir des variables de vue avec `Mailer::setViewVars()` :

```
$mailer = new Mailer('templated');
$mailer->setViewVars(['value' => 12345]);
```

Ou bien vous pouvez utiliser les méthodes du *view builder* `ViewBuilder::setVar()` et `ViewBuilder::setVars()`.

Dans votre template d'email, vous pouvez les utiliser ainsi :

```
<p>Voici votre valeur: <b><?= $value; ?></b></p>
```

Vous pouvez aussi utiliser les helpers dans les emails, un peu comme vous le faites dans des fichiers de template normaux. Seul `HtmlHelper` est chargé par défaut. Vous pouvez charger d'autres helpers en utilisant la méthode `ViewBuilder::addHelpers()` :

```
$mailer->viewBuilder()->addHelpers(['Html', 'Perso', 'Text']);
```

Quand vous ajoutez des helpers, assurez-vous d'inclure "Html" sinon il sera retiré des helpers chargés dans votre template d'email.

> **Note**
>
> Dans les versions antérieures à 4.3.0, vous deviez utilisez `setHelpers()` à la place.

Si vous voulez envoyer un email en utilisant des templates dans un plugin, vous pouvez le faire en utilisant la *syntaxe de plugin* qui doit vous être familière :

```
$mailer = new Mailer();
$mailer->viewBuilder()->setTemplate('Blog.nouveau_commentaire');
```

Ceci utiliserait par exemple les templates du plugin Blog.

Dans certains cas, vous pouvez avoir besoin de substituer le template par défaut fourni par les plugins. Vous pouvez le faire en utilisant les thèmes :

```
$mailer->viewBuilder()
    ->setTemplate('Blog.nouveau_commentaire')
    ->setLayout('Blog.auto_message')
    ->setTheme('TestTheme');
```

Cela vous permet de remplacer le template *nouveau_commentaire* dans votre theme sans modifier le plugin Blog. Le fichier de template devra être créé sous le chemin suivant : **templates/plugin/TestTheme/plugin/Blog/email/text/nouveau_commentaire.php**.

Envoyer des Pièces Jointes

`Cake\Mailer\Mailer::setAttachments($attachments)`

Vous pouvez aussi rattacher des pièces jointes aux emails. Il y a quelques formats différents selon le type de fichier vous avez, et la façon dont vous voulez que les noms de fichiers apparaissent dans la boîte de réception du client :

1. Tableau : `$mailer->setAttachments(['/chemin/complet/vers/le/fichier.png'])` attacher ce fichier avec le nom fichier.png.
2. Tableau avec clé : `$mailer->setAttachments(['photo.png' => '/chemin/un_hash.png'])` va attacher un_hash.png avec le nom photo.png. Le destinataire verra photo.png, pas some_hash.png.
3. Tableaux imbriqués :

   ```
   $mailer->setAttachments([
       'photo.png' => [
           'file' => '/chemin/un_hash.png',
           'mimetype' => 'image/png',
           'contentId' => 'mon-id-unique'
       ]
   ]);
   ```

   Ceci attachera le fichier avec un mimetype différent et avec un content ID personnalisé (quand vous définissez le content ID, la pièce jointe est transformée en inline). Le mimetype et le contentId sont optionnels sous cette forme.

   3.1. Quand vous utilisez le `contentId`, vous pouvez utiliser le fichier dans le corps HTML avec `<img src="cid:mon-content-id">`.

   3.2. Vous pouvez utiliser l'option `contentDisposition` pour désactiver le header `Content-Disposition` pour une pièce jointe. C'est utile pour l'envoi d'invitations ical à des clients utilisant outlook.

3.3 Au lieu de l'option `file`, vous pouvez fournir les contenus de fichiers en tant que chaîne de caractères en utilisant l'option `data`. Cela vous permet d'attacher des fichiers sans avoir besoin d'un chemin sur le disque.

Assouplir les Règles de Validation d'Adresses

`Cake\Mailer\Mailer::setEmailPattern($pattern)`

Si vous avez des problèmes de validation lors de l'envoi vers des adresses considérées comme non conformes, vous pouvez assouplir le pattern utilisé pour valider les adresses email. C'est parfois nécessaire quand il s'agit de certains ISP Japonais :

```
$mailer = new Mailer('default');

// Assouplir le pattern d'email, de façon à pouvoir écrire
// à des adresses non conformes.
$mailer->setEmailPattern($newPattern);
```

Envoyer des Messages Rapidement

Parfois vous avez besoin d'un moyen rapide d'envoyer un email, et vous n'avez pas particulièrement envie de définir une tonne de configuration juste pour cela. `Cake\Mailer\Mailer::deliver()` est fait pour vous.

Vous pouvez créer votre configuration dans `Cake\Mailer\Email::config()`, ou utiliser un tableau avec toutes les options dont avez besoin, puis utiliser la méthode statique `Mailer::deliver()`. Exemple :

```
Mailer::deliver('toi@example.com', 'Sujet', 'Message', ['from' => 'moi@example.com']);
```

Cette méthode enverra un email à toi@example.com, à partir de moi@example.com avec le sujet « Sujet » et le contenu « Message ».

La valeur de retour de `deliver()` est une instance `Cake\Mailer\Email` entièrement configurée. Si vous ne voulez pas envoyer l'email tout de suite et souhaitez configurer encore certaines choses avant de l'envoyer, vous pouvez définir le 5ème paramètre à `false`.

Le 3ème paramètre est le contenu du message ou un tableau avec les variables (quand vous utilisez un contenu généré par l'application).

Le 4ème paramètre peut être un tableau avec la configuration ou une chaîne de caractères avec le nom d'une configuration figurant dans `Configure`.

Si vous voulez, vous pouvez passer null pour les arguments *to*, *subject* et *message*, et passer toutes les configurations dans le 4ème paramètre (en tableau ou en utilisant `Configure`). Faites un tour par la liste des *configurations* pour connaître toutes les configs acceptées.

Envoyer des Emails en Ligne de Commande

Quand vous envoyez des emails depuis un script CLI (Shells, Tasks, ...), vous devez définir manuellement le nom de domaine à utiliser pour Mailer. Il sera utilisé comme nom d'hôte pour l'id du message (puisqu'il n'y a pas de nom d'hôte dans un environnement CLI) :

```
$email->setDomain('www.example.org');
// Envoie des ids de message tels que ``<UUID@www.example.org>`` (valide)
// au lieu de `<UUID@>`` (invalide)
```

Un id de message valide peut permettre à ce message de ne pas finir dans les spams.

Créer des Emails Réutilisables

Jusqu'à présent, nous avons vu comment utiliser la classe `Mailer` pour créer et envoyer des emails. Mais la principale fonctionnalité d'un mailer est de vous permettre de créer des emails réutilisables n'importe où dans votre application. Ils peuvent aussi servir à contenir différentes configurations d'emails en un seul et même endroit, ce qui vous aide à garder votre code DRY et à déplacer la configuration des emails en dehors des autres parties de votre application.

Dans cet exemple, vous allez créer un `Mailer` qui contient des emails dépendant des utilisateurs. Pour créer votre `UserMailer`, créez un fichier **src/Mailer/UserMailer.php**. Le contenu de ce fichier devra ressembler à ceci :

```
namespace App\Mailer;

use Cake\Mailer\Mailer;

class UserMailer extends Mailer
{
    public function welcome($user)
    {
        $this
            ->setTo($user->email)
            ->setSubject(sprintf('Bienvenue %s', $user->name))
            ->viewBuilder()
                ->setTemplate('message_de_bienvenue', 'personnalisé'); // Par défaut le␣
↪template utilisé a le même nom que la méthode.
    }

    public function resetPassword($user)
    {
        $this
            ->setTo($user->email)
            ->setSubject('Reset password')
            ->setViewVars(['token' => $user->token]);
    }
}
```

Dans notre exemple, nous avons créé deux méthodes, une pour envoyer l'email de bienvenue et l'autre pour envoyer un email de réinitialisation de mot de passe. Chacune de ces méthodes reçoit une `Entity` `User` et utilise ses propriétés pour configurer chacun des emails.

Vous pouvez maintenant utiliser votre `UserMailer` pour envoyer tous les emails dépendant des utilisateurs, depuis n'importe quel endroit de votre application. Par exemple, pour envoyer l'email de bienvenue, vous pouvez faire la chose suivante :

```
namespace App\Controller;

use Cake\Mailer\MailerAwareTrait;

class UsersController extends AppController
{
    use MailerAwareTrait;

    public function register()
    {
        $user = $this->Users->newEmptyEntity();
```

(suite sur la page suivante)

(suite de la page précédente)

```
        if ($this->request->is('post')) {
            $user = $this->Users->patchEntity($user, $this->request->getData())
            if ($this->Users->save($user)) {
                $this->getMailer('User')->send('welcome', [$user]);
            }
        }
        $this->set('user', $user);
    }
}
```

Si vous voulez complètement séparer l'envoi de l'email de bienvenue du code de l'application, votre `UserMailer` peut écouter l'évènement `Model.afterSave`. En utilisant l'évènement, vous pouvez séparer complètement la logique d'envoi d'emails du reste de votre logique « utilisateurs ». Vous pourriez par exemple ajouter ceci à votre `UserMailer` :

```
public function implementedEvents()
{
    return [
        'Model.afterSave' => 'onRegistration'
    ];
}

public function onRegistration(EventInterface $event, EntityInterface $entity,␣
↪ArrayObject $options)
{
    if ($entity->isNew()) {
        $this->send('welcome', [$entity]);
    }
}
```

L'objet mailer sera ainsi enregistré en tant qu'écouteur (*listener*) d'événement et la méthode `onRegistration()` sera appelée chaque fois que l'événement `Model.afterSave` sera déclenché :

```
// attache un gestionnaire d'événements sur Users
$this->Users->getEventManager()->on($this->getMailer('User'));
```

> **Note**
>
> Plus d'informations sur la façon d'enregistrer des écouteurs d'événements dans la documentation *Enregistrer les Listeners*.

Configurer les Transports

Les emails sont délivrés par des "transports". Divers transports vous permettent d'envoyer des messages par la fonction `mail()` de PHP, par des serveurs SMTP, voire pas du tout, ce qui est utile pour le débogage. La configuration des transports vous permet de garder les données de configuration en-dehors du code de votre application, et simplifie le déploiement puisque vous pouvez changer facilement les données de configuration. Voici un exemple de configuration de transport :

```
// Dans config/app.php
'EmailTransport' => [
    // Exemple de configuration de Mail
```

(suite sur la page suivante)

(suite de la page précédente)

```
    'default' => [
        'className' => 'Mail',
    ],
    // Exemple de configuration SMTP
    'gmail' => [
        'host' => 'smtp.gmail.com',
        'port' => 587,
        'username' => 'mon@gmail.com',
        'password' => 'secret',
        'className' => 'Smtp',
        'tls' => true
    ]
],
```

Vous pouvez aussi configurer les transports pendant l'exécution, en utilisant `TransportFactory::setConfig()` :

```
use Cake\Mailer\TransportFactory;

// Définit un transport SMTP
TransportFactory::setConfig('gmail', [
    'host' => 'ssl://smtp.gmail.com',
    'port' => 465,
    'username' => 'mon@gmail.com',
    'password' => 'secret',
    'className' => 'Smtp'
]);
```

Vous pouvez configurer des serveurs SMTP SSL, tels que GMail. Pour ce faire, ajoutez le préfixe `ssl://` dans le nom d'hôte et configurez le numéro de port de façon correspondante. Vous pouvez aussi activer le protocole SMTP TLS en utilisant l'option `tls` :

```
use Cake\Mailer\TransportFactory;

TransportFactory::setConfig('gmail', [
    'host' => 'smtp.gmail.com',
    'port' => 587,
    'username' => 'mon@gmail.com',
    'password' => 'secret',
    'className' => 'Smtp',
    'tls' => true
]);
```

La configuration ci-dessus active la communication TLS pour les emails.

Pour configurer votre mailer pour qu'il utilise un transport spécifique, vous pouvez utiliser la méthode `Cake\Mailer\Mailer::setTransport()` ou placer le transport dans votre configuration :

```
// Utilise un transport nommé déjà configuré dans TransportFactory::setConfig()
$mailer->setTransport('gmail');

// Utilise un objet construit.
$mailer->setTransport(new \Cake\Mailer\Transport\DebugTransport());
```

Avertissement

Pour que cela fonctionne avec Google, vous devrez activer l'accès aux applications moins sécurisées : Allowing less secure apps to access your account [146].

Note

Configuration SMTP Gmail [147].

Note

Pour utiliser SSL et SMTP, SSL devra être configuré dans votre installation PHP.

Les options de configuration peuvent aussi être fournies en chaîne *DSN*. C'est utile quand vous travaillez avec des variables d'environnement ou des fournisseurs *PaaS* :

```
TransportFactory::setConfig('default', [
    'url' => 'smtp://mon@gmail.com:secret@smtp.gmail.com:587?tls=true',
]);
```

Quand vous utilisez une chaîne DSN, vous pouvez définir d'autres paramètres ou options en tant qu'arguments query string.

static `Cake\Mailer\Mailer::`**drop**(*$key*)

Une fois configurés, les transports ne peuvent plus être modifiés. Pour modifier un transport, vous devez d'abord le supprimer puis le reconfigurer.

Créer des Transports Personnalisés

Vous pouvez créer vos propres transports pour des situations telles que l'envoi d'emails par des services comme SendGrid, MailGun ou Postmark. Pour créer votre transport, commencez par créer le fichier **src/Mailer/Transport/ExampleTransport.php** (où "Example" est le nom de votre transport). Au départ, votre fichier doit ressembler à cela :

```
namespace App\Mailer\Transport;

use Cake\Mailer\AbstractTransport;
use Cake\Mailer\Message;

class ExampleTransport extends AbstractTransport
{
    public function send(Message $message): array
    {
        // Faire quelque chose.
    }
}
```

Vous devez implémenter la méthode `send(Message $message)` avec votre propre logique.

146. https://support.google.com/accounts/answer/6010255
147. https://support.google.com/a/answer/176600?hl=en

Envoyer des Emails sans utiliser Mailer

Le `Mailer` est une classe à haut niveau d'abstraction, qui agit commme un pont entre les classes `Cake\Mailer\Message`, `Cake\Mailer\Renderer` et `Cake\Mailer\AbstractTransport` pour configuer les emails avec une interface fluide.

Si vous voulez, vous pouvez aussi utiliser ces classes directement avec le Mailer.

Par exemple :

```
$render = new \Cake\Mailer\Renderer();
$render->viewBuilder()
    ->setTemplate('perso')
    ->setLayout('brillant');

$message = new \Cake\Mailer\Message();
$message
    ->setFrom('admin@cakephp.org')
    ->setTo('user@foo.com')
    ->setBody($render->render());

$transport = new \Cake\Mailer\Transport\MailTransport();
$result = $transport->send($message);
```

Vous pouvez aussi écarter le `Renderer` et définir directement le corps du message avec les méthodes `Message::setBodyText()` et `Message::setBodyHtml()`.

Tester les Mailers

Pour tester les mailers, ajoutez `Cake\TestSuite\EmailTrait` à vos cas de test. Le `MailerTrait` utilise les crochets de PHPUnit pour remplacer les transports d'emails de votre application par un proxy qui intercepte les messages et vous permet de faire des assertions sur le mail qui aurait été envoyé.

Pour commencer à tester les emails, ajoutez le trait à votre cas de test, et chargez les routes au cas où vos emails auraient besoin de générer des URL :

```
namespace App\Test\TestCase\Mailer;

use App\Mailer\WelcomeMailer;
use App\Model\Entity\User;

use Cake\TestSuite\EmailTrait;
use Cake\TestSuite\TestCase;

class WelcomeMailerTestCase extends TestCase
{
    use EmailTrait;

    public function setUp(): void
    {
        parent::setUp();
        $this->loadRoutes();
    }
}
```

Supposons maintenant que nous ayons un mailer qui consiste à envoyer des emails de bienvenue quand un nouvel utilisateur s'inscrit. Nous voulons vérifier que le sujet et le corps du message contiennent le nom de l'utilisateur :

```
// dans notre classe WelcomeMailerTestCase.
public function testName()
{
    $user = new User([
        'name' => 'Alice Alittea',
        'email' => 'alice@example.org',
    ]);
    $mailer = new WelcomeMailer();
    $mailer->send('welcome', [$user]);

    $this->assertMailSentTo($user->email);
    $this->assertMailContainsText('Bonjour ' . $user->name);
    $this->assertMailContainsText('Bienvenue dans CakePHP!');
}
```

Méthodes d'Assertion

Le trait `Cake\TestSuite\EmailTrait` fournit les assertions suivantes :

```
// Un nombre précis d'emails ont été envoyés
$this->assertMailCount($count);

// Aucun email n'a été envoyé
$this->assertNoMailSent();

// Un email a été envoyé à une certaine adresse
$this->assertMailSentTo($address);

// Un email a été envoyé par une certaine adresse
$this->assertMailSentFrom($emailAddress);
$this->assertMailSentFrom([$emailAddress => $displayName]);

// Un email contient un certain contenu
$this->assertMailContains($contents);

// Un email contient un certain contenu HTML
$this->assertMailContainsHtml($contents);

// Un email contient un certain contenu en texte brut
$this->assertMailContainsText($contents);

// Un email contient une certaine valeur dans un getter de Message (par exemple "subject
↪")
$this->assertMailSentWith($expected, $parameter);

// L'email à l'index spécifié a été envoyé à une certaine adresse
$this->assertMailSentToAt($at, $address);

// L'email à l'index spécifié a été envoyé depuis une certaine adresse
$this->assertMailSentFromAt($at, $address);
```

(suite sur la page suivante)

(suite de la page précédente)

```
// L'email à l'index spécifié contient un certain contenu
$this->assertMailContainsAt($at, $contents);

// L'email à l'index spécifié contient un certain contenu HTML
$this->assertMailContainsHtmlAt($at, $contents);

// L'email à l'index spécifié contient un certain contenu en texte brut
$this->assertMailContainsTextAt($at, $contents);

// Un email contient une certaine pièce jointe
$this->assertMailContainsAttachment('test.png');

// L'email à l'index spécifié contient une certaine valeur dans un getter de Message (par␣
↪example, "cc")
$this->assertMailSentWithAt($at, $expected, $parameter);

// Le sujet d'un email contient un certain texte
$this->assertMailSubjectContains('Offre gratuite');

// L'email à l'index spécifié a un sujet qui contient un certain texte
$this->assertMailSubjectContainsAt(1, 'Offre gratuite');
```

CHAPITRE 19

Gestion des Erreurs & Exceptions

Les applications CakePHP sont fournies avec une gestion des erreurs et exceptions prête à l'emploi. Les erreurs PHP sont récupérées et affichées ou loggées. Les exceptions non interceptées sont affichées automatiquement dans des pages d'erreur.

Configuration des Erreurs et des Exceptions

La configuration des Erreurs est faite dans le fichier **config/app.php** de votre application. Par défaut CakePHP utilise `Cake\Error\ErrorHandler` pour traiter aussi bien les erreurs PHP que les exceptions par défaut. La configuration des erreurs vous permet de personnaliser le traitement de l'erreur pour votre application. Les options proposées sont les suivantes :

- `errorLevel` - int - Le niveau d'erreurs que vous souhaitez pour la capture. Utilisez les constantes d'erreur intégrées à PHP et les bitmasks pour sélectionner le niveau d'erreur qui vous intéresse. Cf. *Avertissements de dépréciation* pour supprimer les avertissements de dépréciation.
- `trace` - bool - Inclut les stack traces (contexte de débuggage) pour les erreurs dans les fichiers de log. Les Stack traces seront inclus dans le log après chaque erreur. Ceci est utile pour trouver où/quand des erreurs sont générées.
- `exceptionRenderer` - string - La classe responsable de rendre les exceptions non interceptées. Si vous choisissez une classe personnalisée, vous devrez placer le fichier de cette classe dans le dossier **src/Error**. Cette classe doit implémenter une méthode `render()`.
- `log` - bool - Si `true`, les exceptions et leur stack traces seront loguées vers *`Cake\Log\Log`*.
- `skipLog` - array - Un tableau des noms de classe d'exception qui ne doivent pas être mises dans des fichiers de log. C'est utile pour supprimer les NotFoundExceptions ou toute autre message de log sans intérêt.
- `extraFatalErrorMemory` - int - Définit le nombre de megaoctets duquel doit être augmenté la limite de mémoire en cas d'erreur fatale. Cela permet d'allouer un petit espace mémoire supplémentaire pour les logs ainsi que la gestion d'erreurs.
- `errorLogger` - `Cake\Error\ErrorLoggerInterface` - La classe chargée de logger les erreurs et les exceptions non interceptées. Par défaut, il s'agit de `Cake\Error\ErrorLogger`.
- `ignoredDeprecationPaths` - array - Une liste de chemins compatibles glob à l'intérieur desquels les erreurs de dépréciation devraient être ignorées. Ajouté dans la version 4.2.0

Par défaut, les erreurs PHP sont affichées quand `debug` est `true` et loggées quand `debug` est `false`. Le gestionnaire d'erreurs fatales va être appelé indépendamment de `debug` ou de la configuration de `errorLevel`, mais le résultat va être différent, basé sur le niveau de `debug`. Le comportement par défaut pour les erreurs fatales est d'afficher une page avec une erreur interne du serveur (`debug` désactivé) ou une page avec le message, le fichier et la ligne (`debug` activé).

> **Note**
>
> Si vous utilisez un gestionnaire d'erreurs personnalisé, les options supportées dépendent de votre gestionnaire.

Avertissements de dépréciation

CakePHP utilise les avertissements de dépréciation pour indiquer quand des fonctionnalités ont été dépréciées. Nous recommandons également ce système pour le code de vos plugins ou de vos applications quand c'est utile. Vous pouvez déclencher un avertissement de dépréciation avec `deprecationWarning()` :

```
deprecationWarning('5.0', 'La méthode example() est dépréciée. Veuillez utiliser
getExample() à la place.');
```

Quand vous mettez à jour CakePHP ou des plugins, vous pouvez y découvrir de nouveaux avertissements de dépréciation. Vous pouvez les désactiver temporairement en utilisant un de ces moyens :

1. Placer le paramètre `Error.errorLevel` à `E_ALL ^ E_USER_DEPRECATED` pour ignorer *tous* les avertissements de dépréciation.
2. Utiliser l'option de configuration `Error.ignoredDeprecationPaths` pour ignorer les dépréciations selon une expression glob. Par exemple :

   ```
   'Error' => [
       'ignoredDeprecationPaths' => [
           'vendors/company/contacts/*',
           'src/Models/*',
       ]
   ],
   ```

   ignorerait toutes les dépréciations depuis votre répertoire `Models` et le plugin `Contacts` dans votre application.

class ExceptionRenderer(*Exception $exception*)

Modifier la gestion des exceptions

La gestion des exceptions vous offre plusieurs moyens pour affiner la façon dont les exceptions sont gérées. Chaque approche vous donne un avantage différent dans le contrôle du processus de gestion de l'exception.

1. *Templates d'erreur personnalisés* vous permet de changer les templates de la vue affichée, de la même façon que vous changeriez n'importe quel autre template de votre application..
2. *ErrorController personnalisé* vous permet de contrôler comment les pages d'exception sont affichées.
3. *ExceptionRenderer personnalisé* vous permet de contrôler comment les pages d'exception et de log sont réalisées.
4. *Créez et enregistrez votre propre gestionnaire d'erreurs* vous donne le contrôle total sur la façon dont les erreurs et exceptions sont gérées, loggées et affichées.

Templates d'erreur personnalisés

Le gestionnaire d'erreur par défaut affiche toutes les exceptions non interceptées soulevées par votre application en s'appuyant sur `Cake\Error\ExceptionRenderer`, et sur l'"`ErrorController` de votre application.

Les vues de la page d'erreur sont situées dans **templates/Error/**. Toutes les erreurs 4xx errors utilisent le template **error400.php**, et les erreurs 5xx utilisent **error500.php**. Vos templates d'erreur disposeront des variables suivantes :

- `message` Le message de l'exception.
- `code` Le code de l'exception.
- `url` L'URL demandée.
- `error` L'objet exception.

En mode debug, si votre erreur sous-classe `Cake\Core\Exception\Exception`, les données renvoyées par `getAttributes()` seront aussi exposées comme variables de vue.

> **Note**
>
> Vous aurez besoin de définir `debug` à false pour voir vos templates **error404** et **error500**. En mode debug, vous verrez la page d'erreur de développement de CakePHP.

Layout personnalisé de la page d'erreur

Par défaut les templates d'erreur utilisent comme layout **templates/layout/error.php**. Vous pouvez utiliser la propriété `layout` pour aller chercher un layout différent :

```
// à l'intérieur de templates/Error/error400.php
$this->layout = 'my_error';
```

Le code ci-dessus utiliserait **templates/layout/my_error.php** comme layout pour vos pages d'erreur.

Beaucoup d'exceptions soulevées par CakePHP vont afficher des templates de vue spécifiques en mode debug. Lorsque le mode debug est désactivé, toutes les exceptions soulevées par CakePHP utiliseront soit **error400.php** soit **error500.php** selon leur code de statut.

ErrorController personnalisé

La classe `App\Controller\ErrorController` est utilisée par le moteur de rendu des exceptions de CakePHP pour rendre la vue de la page d'erreur. Elle reçoit tous les événements du cycle de vie d'une requête standard. En modifiant cette classe, vous pouvez contrôler quels composants sont utilisés et quels templates sont rendus.

Si votre application utilise *Prefix de Routage*, vous pouvez créer des contrôleurs d'erreur personnalisés pour chaque préfixe de route. Par exemple, si vous aviez un préfixe `Admin`, vous pourriez créer la classe suivante :

```
namespace App\Controller\Admin;

use App\Controller\AppController;
use Cake\Event\EventInterface;

class ErrorController extends AppController
{
    /**
     * Initialization hook method.
     *
     * @return void
```

(suite sur la page suivante)

(suite de la page précédente)

```
     */
    public function initialize(): void
    {
        $this->loadComponent('RequestHandler');
    }

    /**
     * beforeRender callback.
     *
     * @param \Cake\Event\EventInterface $event Event.
     * @return void
     */
    public function beforeRender(EventInterface $event)
    {
        $this->viewBuilder()->setTemplatePath('Error');
    }
}
```

Ce contrôleur serait seulement utilisé quand une erreur est rencontrée dans un contrôleur préfixé, et vous permet de définir une logique ou des templates spécifiques au préfixe en tant que de besoin.

ExceptionRenderer personnalisé

Si vous voulez contrôler tout l'affichage de l'exception et le processus de log, vous pouvez utiliser l'option `Error.exceptionRenderer` dans **config/app.php** pour choisir une classe qui va faire le rendu des pages d'exception. Le fait de changer l'ExceptionRenderer est utile quand vous voulez changer la logique utilisée pour créer un contrôleur d'erreur, choisir un template d'erreur, ou contrôler l'intégralité du processus de rendu.

Votre classe personnalisée d'affichage des erreurs devrait être placée dans **src/Error**. Supposons que notre application utilise `App\Exception\MissingWidgetException` pour indiquer un widget manquant. Nous pourrions créer un renderer d'exceptions qui affiche des pages d'erreur spécifiques quand l'erreur est traitée :

```
// Dans src/Error/AppExceptionRenderer.php
namespace App\Error;
use Cake\Error\ExceptionRenderer;

class AppExceptionRenderer extends ExceptionRenderer
{
    public function missingWidget($error)
    {
        $response = $this->controller->getResponse();

        return $response->withStringBody('Oups ! Ce widget est introuvable.');
    }
}

// Dans config/app.php
'Error' => [
    'exceptionRenderer' => 'App\Error\AppExceptionRenderer',
    // ...
],
// ...
```

Le code ci-dessus traiterait notre `MissingWidgetException`, et nous permettrait de fournir une logique personnalisée d'affichage et/ou de gestion pour ces exceptions de l'application. Les méthodes de rendu des exceptions reçoivent en argument l'exception traitée, et devraient retourner un objet `Response`. Vous pouvez aussi implémenter des méthodes pour ajouter une logique supplémentaire dans la gestion des erreurs CakePHP :

```
// Dans src/Error/AppExceptionRenderer.php
namespace App\Error;

use Cake\Error\ExceptionRenderer;

class AppExceptionRenderer extends ExceptionRenderer
{
    public function notFound($error)
    {
        // Faire quelque chose avec les objets NotFoundException.
    }
}
```

Changer la classe ErrorController

Le renderer d'exception dicte le contrôleur à utiliser pour le rendu des exceptions. Si vous voulez changer le contrôleur à utiliser pour rendre les exceptions, réécrivez la méthode `_getController()` dans votre renderer d'exceptions :

```
// dans src/Error/AppExceptionRenderer
namespace App\Error;

use App\Controller\SuperCustomErrorController;
use Cake\Controller\Controller;
use Cake\Error\ExceptionRenderer;

class AppExceptionRenderer extends ExceptionRenderer
{
    protected function _getController(): Controller
    {
        return new SuperCustomErrorController();
    }
}

// dans config/app.php
'Error' => [
    'exceptionRenderer' => 'App\Error\AppExceptionRenderer',
    // ...
],
// ...
```

Créer vos Propres Gestionnaires d'Erreurs

En remplaçant le gestionnaire d'erreurs, vous pouvez personnaliser la façon dont sont gérées les erreurs PHP et les exceptions qui ne sont pas interceptées par un middleware. Les gestionnaires d'erreurs sont différents pour la partie HTTP et la partie Console de votre application.

Pour créer un gestionnaire d'erreurs pour les requêtes HTTP, vous devriez étendre `Cake\Error\ErrorHandler`. À titre d'exemple, nous pourrions définir une classe appelée `AppError` pour gérer les erreurs dans les requêtes HTTP :

```
// Dans src/Error/AppError.php
namespace App\Error;

use Cake\Error\ErrorHandler;
use Throwable;


class AppError extends ErrorHandler
{
    protected function _displayError(array $error, bool $debug): void
    {
        echo 'Il y a eu une erreur!';
    }

    protected function _displayException(Throwable $exception): void
    {
        echo 'Il y a eu un exception';
    }
}
```

Ensuite nous pouvons enregistrer notre gestionnaire en tant que gestionnaire d'erreurs PHP :

```
// Dans config/bootstrap.php
use App\Error\AppError;

if (PHP_SAPI !== 'cli') {
    $errorHandler = new AppError();
    $errorHandler->register();
}
```

Pour finir, nous pouvons utiliser notre gestionnaire d'erreurs dans l'`ErrorHandlerMiddleware` :

```
// dans src/Application.php
public function middleware(MiddlewareQueue $middlewareQueue): MiddlewareQueue
{
    $error = new AppError(Configure::read('Error'));
    $middleware->add(new ErrorHandlerMiddleware($error));

    return $middleware;
}
```

Pour la gestion d'erreurs par console, vous devez étendre `Cake\Error\ConsoleErrorHandler` au lieu de `Cake\Error\ErrorHandler` :

```
// Dans /src/Error/AppConsoleErrorHandler.php
namespace App\Error;
use Cake\Error\ConsoleErrorHandler;

class AppConsoleErrorHandler extends ConsoleErrorHandler {

    protected function _displayException(Throwable $exception): void {
        parent::_displayException($exception);
        if (isset($exception->queryString)) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
            $this->_stderr->write('Query String: ' . $exception->queryString);
        }
    }

}
```

Puis nous pouvons enregistrer notre gestionnaire d'erreurs sur console en tant que gestionnaire d'erreurs PHP :

```
// Dans config/bootstrap.php
use App\Error\AppConsoleErrorHandler;
$isCli = PHP_SAPI === 'cli';
if ($isCli) {
    (new AppConsoleErrorHandler(Configure::read('Error')))->register();
}
```

Les objets ErrorHandler ont quelques méthodes que vous pourriez vouloir implémenter :

- `_displayError(array $error, bool $debug)` est utilisée quand des erreurs sont déclenchées.
- `_displayException(Throwable $exception)` est appelée lorsqu'il y a une exception non interceptée.
- `_logError($level, array $error)` est appelée lorsqu'une erreur doit être loggée.
- `logException(Throwable $exception)` est appelée lorsqu'une exception doit être loggée.

Changer le Comportement des Erreurs Fatales

Les gestionnaires d'erreurs convertissent les erreurs fatales en exceptions et réutilisent la logique de gestion des erreurs pour rendre une page d'erreur. Si vous ne voulez pas montrer la page d'erreur standard, vous pouvez la réécrire :

```
// Dans src/Error/AppError.php
namespace App\Error;

use Cake\Error\BaseErrorHandler;

class AppError extends BaseErrorHandler
{
    // Autres méthodes.

    public function handleFatalError(int $code, string $description, string $file, int
↪$line): bool
    {
        echo 'Une erreur fatale est survenue';
    }
}
```

Logging Personnalisé des Erreurs

Les gestionnaires d'erreurs utilisent des instances de `Cake\Error\ErrorLoggingInterface` pour créer des messages de log et les logger au bon endroit. Vous pouvez remplacer le logger d'erreurs en utilisant la propriété de configuration `Error.errorLogger`. Un exemple d'"error logger :

```
namespace App\Error;

use Cake\Error\ErrorLoggerInterface;
use Psr\Http\Message\ServerRequestInterface;
```

(suite sur la page suivante)

(suite de la page précédente)

```
use Throwable;

/**
 * Logger vers `Cake\Log\Log` les erreurs et les exceptions non interceptées
 */
class ErrorLogger implements ErrorLoggerInterface
{
    /**
     * @inheritDoc
     */
    public function logMessage($level, string $message, array $context = []): bool
    {
        // Logger les erreurs PHP
    }

    public function log(Throwable $exception, ?ServerRequestInterface $request = null):␣
↪bool
    {
        // Logger les exceptions
    }
}
```

Créer vos propres Exceptions d'Application

Vous pouvez créer vos propres exceptions d'application en utilisant l'une des exceptions intégrées SPL exceptions [148], `Exception` , ou *`Cake\Core\Exception\Exception`*. Si votre application contenait l'exception suivante :

```
use Cake\Core\Exception\Exception;

class MissingWidgetException extends Exception
{
}
```

Vous pourriez produire des erreurs de développement élégantes en créant **templates/Error/missing_widget.php**. En production, l'erreur ci-dessus serait traitée comme une erreur 500 et utiliserait le template **error500**.

Si vos exceptions ont un code compris entre `400` et `506`, le code de l'exception sera utilisé comme code de réponse HTTP.

Le constructeur pour *`Cake\Core\Exception\Exception`* vous permet de passer des données supplémentaires. Ces données supplémentaires sont interpolées dans le `_messageTemplate`. Cela vous permet de créer des exceptions riches en données, qui fournissent plus de contexte autour de vos erreurs :

```
use Cake\Core\Exception\Exception;

class MissingWidgetException extends Exception
{
    // Les données de contexte sont interpolées dans cette chaîne formatée.
    protected $_messageTemplate = 'On dirait qu'il manque %s.';

    // Vous pouvez aussi définir un code d'exception par défaut.
```

(suite sur la page suivante)

148. https://php.net/manual/en/spl.exceptions.php

(suite de la page précédente)

```
    protected $_defaultCode = 404;
}

throw new MissingWidgetException(['widget' => 'Pointy']);
```

Lors du rendu, le template de votre vue disposerait d'une variable `$widget` déjà définie. Si vous castez l'exception en string ou si vous utilisez sa méthode `getMessage()`, vous obtiendrez `On dirait qu'il manque Pointy.`.

Logger des Exceptions

Avec la gestion d'erreurs intégrée, vous pouvez faire logger par ErrorHandler toutes les exceptions auxquelles vous aurez affaire en définissant l'option `log` à `true` dans votre **config/app.php**. Le fait de l'activer va logger toutes les exceptions dans *Cake\Log\Log* et les loggers configurés.

> **Note**
>
> Si vous utilisez un gestionnaire d'exceptions personnalisé, ce paramètre n'aura aucun effet. À moins que vous ne le référenciez depuis votre implémentation.

Exceptions Intégrées de CakePHP

Il existe plusieurs exceptions intégrées à l'intérieur de CakePHP, en plus des exceptions d'infrastructure internes, et il existe plusieurs exceptions pour les méthodes HTTP.

Exceptions HTTP

exception `Cake\Http\Exception\`**`BadRequestException`**

Utilisée pour faire une erreur 400 de Mauvaise Requête.

exception `Cake\Http\Exception\`**`UnauthorizedException`**

Utilisée pour faire une erreur 401 Non Autorisé.

exception `Cake\Http\Exception\`**`ForbiddenException`**

Utilisée pour faire une erreur 403 Interdite.

exception `Cake\Http\Exception\`**`InvalidCsrfTokenException`**

Utilisée pour faire une erreur 403 causée par un token CSRF invalide.

exception `Cake\Http\Exception\`**`NotFoundException`**

Utilisée pour faire une erreur 404 Non Trouvé.

exception `Cake\Http\Exception\`**`MethodNotAllowedException`**

Utilisée pour faire une erreur 405 pour les Méthodes Non Autorisées.

exception `Cake\Http\Exception\`**`NotAcceptableException`**

Utilisée pour faire une erreur 406 Not Acceptable.

exception `Cake\Http\Exception\`**`ConflictException`**

Utilisée pour faire une erreur 409 Conflict.

exception Cake\Http\Exception**GoneException**

Utilisée pour faire une erreur 410 Gone.

Pour plus de détails sur les codes de statut d'erreur HTTP 4xx, regardez **RFC 2616 Section 10.4** [149].

exception Cake\Http\Exception**InternalErrorException**

Utilisée pour faire une erreur 500 du Serveur Interne.

exception Cake\Http\Exception**NotImplementedException**

Utilisée pour faire une erreur 501 Non Implémentée.

exception Cake\Http\Exception**ServiceUnavailableException**

Utilisée pour faire une erreur 503 Service Unavailable.

Pour plus de détails sur les codes de statut d'erreur HTTP 5xx, regardez **RFC 2616 Section 10.5** [150].

Vous pouvez lancer ces exceptions à partir de vos controllers pour indiquer les états d'échecs, ou les erreurs HTTP. Un exemple d'utilisation des exceptions HTTP pourrait être le rendu de pages 404 pour les items qui n'ont pas été trouvés :

```
use Cake\Http\Exception\NotFoundException;

public function view($id = null)
{
    $article = $this->Articles->findById($id)->first();
    if (empty($article)) {
        throw new NotFoundException(__('Article not found'));
    }
    $this->set('article', $article);
    $this->viewBuilder()->setOption('serialize', ['article']);

}
```

En utilisant les exceptions pour les erreurs HTTP, vous pouvez garder à la fois votre code propre, et donner les réponses RESTful aux applications clientes et aux utilisateurs.

Utiliser des Exceptions HTTP dans vos Contrôleurs

Vous pouvez lancer n'importe quelle exception HTTP depuis les actions de vos contrôleurs pour indiquer des états d'échec. Par exemple :

```
use Cake\Network\Exception\NotFoundException;

public function view($id = null)
{
    $article = $this->Articles->findById($id)->first();
    if (empty($article)) {
        throw new NotFoundException(__('Article introuvable'));
    }
    $this->set('article', 'article');
    $this->viewBuilder()->setOption('serialize', ['article']);
}
```

Ce qui précède va faire que le gestionnaire d'exception qui a été configuré attrape et traite la `NotFoundException`. Par défaut, cela créera une page d'erreur et loggera l'exception.

149. https://datatracker.ietf.org/doc/html/rfc2616.html#section-10.4
150. https://datatracker.ietf.org/doc/html/rfc2616.html#section-10.5

Autres Exceptions Intégrées

De plus, CakePHP utilise les exceptions suivantes :

exception `Cake\View\Exception\`**`MissingViewException`**

La classe View choisie n'a pas pu être trouvée.

exception `Cake\View\Exception\`**`MissingTemplateException`**

Le fichier de template choisi n'a pas pu être trouvé.

exception `Cake\View\Exception\`**`MissingLayoutException`**

Le layout choisi n'a pas pu être trouvé.

exception `Cake\View\Exception\`**`MissingHelperException`**

Un helper n'a pas pu être trouvé.

exception `Cake\View\Exception\`**`MissingElementException`**

L'element n'a pas pu être trouvé.

exception `Cake\View\Exception\`**`MissingCellException`**

La classe Cell choisie n'a pas pu être trouvée.

exception `Cake\View\Exception\`**`MissingCellViewException`**

La vue de Cell choisie n'a pas pu être trouvée.

exception `Cake\Controller\Exception\`**`MissingComponentException`**

Un component configuré n'a pas pu être trouvé.

exception `Cake\Controller\Exception\`**`MissingActionException`**

L'action demandée du controller n'a pas pu être trouvé.

exception `Cake\Controller\Exception\`**`PrivateActionException`**

Accès à une action préfixée par _, privée ou protégée.

exception `Cake\Console\Exception\`**`ConsoleException`**

Une classe de la librairie console a rencontré une erreur

exception `Cake\Console\Exception\`**`MissingTaskException`**

Une tâche configurée n'a pas pu être trouvée.

exception `Cake\Console\Exception\`**`MissingShellException`**

Une classe de shell n'a pas pu être trouvée.

exception `Cake\Console\Exception\`**`MissingShellMethodException`**

Une classe de shell choisie n'a pas de méthode de ce nom.

exception `Cake\Database\Exception\`**`MissingConnectionException`**

Une connexion à un model n'existe pas.

exception `Cake\Database\Exception\`**`MissingDriverException`**

Un driver de base de donnée de n'a pas pu être trouvé.

exception `Cake\Database\Exception\`**`MissingExtensionException`**

Une extension PHP est manquante pour le driver de la base de données.

exception `Cake\ORM\Exception\`**`MissingTableException`**

Une table du model n'a pas pu être trouvé.

exception `Cake\ORM\Exception\`**`MissingEntityException`**

Une entity du model n'a pas pu être trouvé.

exception `Cake\ORM\Exception\`**`MissingBehaviorException`**

Une behavior du model n'a pas pu être trouvé.

exception `Cake\ORM\Exception\`**`PersistenceFailedException`**

Une entity n'a pas pu être sauvegardée / supprimée en utilisant *`Cake\ORM\Table::saveOrFail()`* ou *`Cake\ORM\Table::deleteOrFail()`*

exception `Cake\Datasource\Exception\`**`RecordNotFoundException`**

L'enregistrement demandé n'a pas pu être trouvé. Génère une réponse avec une entête 404.

exception `Cake\Routing\Exception\`**`MissingControllerException`**

Le controller requêté n'a pas pu être trouvé.

exception `Cake\Routing\Exception\`**`MissingRouteException`**

L'URL demandée ne pas peut pas être inversée ou ne peut pas être parsée.

exception `Cake\Routing\Exception\`**`MissingDispatcherFilterException`**

Le filtre du dispatcher n'a pas pu être trouvé.

exception `Cake\Core\Exception\`**`Exception`**

Classe de base des exceptions dans CakePHP. Toutes les exceptions lancées par CakePHP étendent cette classe.

Ces classes d'exception étendent toutes *`Exception`*. En étendant Exception, vous pouvez créer vos propres erreurs "framework".

`Cake\Core\Exception\Exception::`**`responseHeader`**(*$header = null*, *$value = null*)

See `Cake\Network\Request::header()`

Toutes les exceptions Http et CakePHP étendent la classe Exception, qui a une méthode pour ajouter les en-têtes à la réponse. Par exemple quand vous lancez une MethodNotAllowedException 405, le rfc2616 dit :

```
"La réponse DOIT inclure un en-tête contenant une liste de méthodes valides
pour la ressource requêtée."
```

CHAPITRE 20

Événements système

La création d'applications maintenables est à la fois une science et un art. Il est bien connu que la clé pour avoir un code de bonne qualité est d'avoir un couplage plus lâche et une cohésion plus élevée. La cohésion signifie que toutes les méthodes et propriétés pour une classe sont fortement liées à la classe elle-même et qu'elles n'essaient pas de faire le travail que d'autre objets devraient faire, alors qu'un couplage plus lâche est la mesure du degré de connexion d'une classe par rapport aux objets externes, et comment cette classe en dépend.

Alors que la plupart des structures CakePHP et des librairies par défaut vous aideront à atteindre ce but, il y a certains cas où vous avez besoin de communiquer proprement avec les autres parties du système sans avoir à coder en dur ces dépendances, ce qui réduit la cohésion et augmente le couplage de classe. Un motif de conception (design pattern) fonctionnant très bien dans l'ingénierie software est le modèle obervateur (Observer pattern), où les objets peuvent générer des événements (events) et notifier à des écouteurs (listeners) possiblement anonymes des changements d'états internes.

Les écouteurs (listener) dans le modèle observateur (Observer pattern) peuvent s'abonner à de tels événements et choisir d'agir sur eux, modifier l'état du sujet ou simplement créer des fichiers de logs. Si vous avez utilisé JavaScript dans le passé, il y a de fortes chances que vous soyez déjà familier avec la programmation événementielle.

CakePHP émule plusieurs aspects sur la façon dont les événements sont déclenchés et gérés dans des frameworks JavaScript comme le populaire jQuery, tout en restant fidèle à sa conception orientée objet. Dans cette implémentation, un objet événement est transporté a travers tous les écouteurs qui détiennent l'information et la possibilité d'arrêter la propagation des événements à tout moment. Les écouteurs peuvent s'enregistrer eux-mêmes ou peuvent déléguer cette tâche à d'autres objets et peuvent modifier l'état et l'événement lui-même pour le reste des callbacks.

Le sous-système d'événement est au cœur des callbacks de Model, de Behavior, de Controller, de View et de Helper. Si vous avez déjà utilisé l'un d'eux, vous êtes quelque part déjà familiarisé avec les événements dans CakePHP.

Exemple d'Utilisation d'Événement

Imaginons que vous être en train de construire un plugin Caddie, mais que vous ne voulez pas vraiment l'encombrer avec une logique d'expédition, expédier un mail à l'utilisateur ou décrémenter les articles depuis le stock, c'est votre souhait de traiter tout cela séparément dans un autre plugin ou dans le code de l'application. Typiquement, quand vous

n'utilisez pas directement le modèle observateur (observer pattern) vous feriez cela en attachant des behaviors à la volée à vos models, et peut être quelques components aux controllers. Faire comme ceci représente des difficultés la plupart du temps, parce qu'il va falloir le code nécessaire pour charger ces behaviors ou pour les attacher aux controllers de votre plugin.

A la place, vous pouvez utiliser les événements pour vous permettre de séparer clairement ce qui concerne votre code et permettre d'ajouter des besoins supplémentaires dans votre plugin en utilisant les événements. Par exemple dans votre plugin Cart, vous avez un model Orders qui gère la création des commandes. Vous voulez notifier au reste de l'application qu'une commande a été créée. Pour garder votre model Orders propre, vous pouvez utiliser les événements :

```
// Cart/Model/Table/OrdersTable.php
namespace Cart\Model\Table;

use Cake\Event\Event;
use Cake\ORM\Table;

class OrdersTable extends Table
{

    public function place($order)
    {
        if ($this->save($order)) {
            $this->Cart->remove($order);
            $event = new Event('Model.Orders.afterPlace', $this, [
                'order' => $order
            ]);
            $this->getEventManager()->dispatch($event);

            return true;
        }

        return false;
    }
}
```

Le code ci-dessus vous permet de notifier aux autres parties de l'application qu'une commande a été créée. Vous pouvez ensuite faire des tâches comme envoyer les notifications par mail, mettre à jour le stock, enregistrer les statistiques pertinentes et d'autres tâches dans des objets séparés qui se focalisent sur ces préoccupations.

Accéder aux Gestionnaires d'Événements

Dans CakePHP, les événements sont attrapés par les gestionnaires d'événements. Les gestionnaires d'événements sont disponibles dans chaque Table, View et Controller en utilisant `eventManager()` :

```
$events = $this->eventManager();
```

Chaque Model a un gestionnaire d'événements séparé, alors que View et Controller en partagent un. Cela permet aux événements de Model d'être autonomes, et permet aux components ou aux controllers d'agir sur les événements créés dans la vue si nécessaire.

Le Gestionnaire d'Événement Global

En plus des gestionnaires au niveau des instances d'événement, CakePHP fournit un gestionnaire d'événements global qui vous permet d'écouter tout événement déclenché dans une application. C'est utile quand attacher des écouteurs à une instance spécifique semble lent ou difficile. Le gestionnaire global est une instance singleton de `Cake\Event\EventManager` qui reçoit chaque événement **avant** que les gestionnaires d'instance ne le reçoivent. En plus de recevoir les événements en premier, le gestionnaire global maintient aussi une pile de priorité distincte pour les écouteurs. Une fois qu'un événement a été dispatché au gestionnaire global, il sera dispatché au gestionnaire au niveau de l'instance. Vous pouvez accéder au gestionnaire global en utilisant une méthode statique :

```
// Dans tout fichier de configuration ou partie de code qui s'execute avant l'événement
use Cake\Event\EventManager;

EventManager::instance()->on(
    'Model.Order.afterPlace',
    $aCallback
);
```

Une chose importante que vous devriez considérer est que les événements qui seront attrapés auront le même nom mais des sujets différents, ainsi le vérifier dans l'objet event est habituellement nécessaire dans toute fonction qui devient attachée globalement afin d'éviter tout bug. Rappelez-vous que la flexibilité du gestionnaire global entraîne une complexité supplémentaire à gérer.

La méthode `Cake\Event\EventManager::dispatch()` accepte l'objet event en argument et notifie à tous les écouteurs et les callbacks qui passent cet objet. Les écouteurs vont gérer toute la logique supplémentaire autour de l'événement `afterPlace`, vous pouvez enregistrer l'horodatage dans les journaux, envoyer les emails, mettre à jour les statistiques d'un utilisateur, si possible dans des objets séparés et même le déléguer à des tâches offline si vous avez ce besoin.

Suivre la Trace des Événements

Pour garder une liste des événements qui sont déclenchés pour un `EventManager` en particulier, vous pouvez activer le tracking d'événements. Pour ce faire, attachez simplement une `Cake\Event\EventList` au gestionnaire :

```
EventManager::instance()->setEventList(new EventList());
```

Après avoir déclenché un événement sur le gestionnaire, vous pouvez le récupérer à partir de la liste d'événements :

```
$eventsFired = EventManager::instance()->getEventList();
$firstEvent = $eventsFired[0];
```

Le tracking peut être désactivé en retirant la liste d'événements ou en appelant `Cake\Event\EventList::trackEvents(false)`

Events du Cœur

Il y a de certain nombre d'événements du cœur du framework que votre application peut écouter. Chaque couche de CakePHP émet des événements que vous pouvez écouter dans votre application.

- *Events de l'ORM et du Model*
- *Events du Controller*
- *Events de View*

Enregistrer les Listeners

Les listeners (écouteurs) sont le meilleur moyen d'enregistrer les callbacks pour un événement. Ceci est fait en intégrant l'interface `Cake\Event\EventListenerInterface` dans toute classe dans laquelle vous souhaitez enregistrer des callbacks. Les classes l'intégrant ont besoin de fournir la méthode `implementedEvents()`. Cette méthode doit retourner un tableau associatif avec tous les noms d'événement que la classe va gérer.

Pour continuer notre exemple précédent, imaginons que nous ayons une classe UserStatistic qui s'occupe de calculer l'historique des achats d'un utilisateur et les compile dans des statistiques globales du site. C'est un bon cas pour utiliser une classe listener. Faire ceci vous permet aussi de vous concentrer sur la logique des statistiques à un endroit et de réagir aux événements si nécessaire. Notre écouteur `UserStatistics` pourrait commencer comme ceci :

```
use Cake\Event\EventListenerInterface;

class UserStatistic implements EventListenerInterface
{

    public function implementedEvents()
    {
        return [
            // La personnalisation des noms d'événements vous permet de
            // concevoir au mieux votre application.
            'Model.Orders.afterPlace' => 'updateBuyStatistic',
        ];
    }

    public function updateBuyStatistic($event, $order)
    {
        // Code to update statistics
    }
}

// Attache l'objet UserStatistic au gestionnaire globale d'événements de la Commande
$statistics = new UserStatistic();
$this->Orders->eventManager()->on($statistics);
```

Comme vous pouvez le voir dans le code ci-dessus, la fonction `on()` va accepter les instances de l'interface `EventListener`. En interne, le gestionnaire d'événements va utiliser `implementedEvents()` pour attacher les bons callbacks.

Enregistrer des Écouteurs Anonymes

Alors que les objets listener d'événement sont généralement une meilleure façon d'intégrer des listeners, vous pouvez aussi lier tout `callable` comme un listener d'événement. Par exemple si nous souhaitons mettre toutes les commandes dans des fichiers de log, nous pourrions utiliser une simple fonction anonyme pour le faire :

```
use Cake\Log\Log;

// Depuis un controller, ou pendant la phase de bootstrap de l'application
$this->Orders->eventManager()->on('Model.Orders.afterPlace', function ($event) {
    Log::write(
        'info',
        'A new order was placed with id: ' . $event->getSubject()->id
    );
```

(suite sur la page suivante)

(suite de la page précédente)

```
});
```

En plus des fonctions anonymes, vous pouvez utiliser tout autre type callable que PHP supporte :

```
$events = [
    'email-sending' => 'EmailSender::sendBuyEmail',
    'inventory' => [$this->InventoryManager, 'decrement'],
];
foreach ($events as $callable) {
    $eventManager->on('Model.Orders.afterPlace', $callable);
}
```

Quand vous travaillez avec des plugins qui ne déclenchent pas d'événement spécifique, vous pouvez utiliser les listeners d'événements sur les événements utilisés par défaut. Prenons un exemple d'un plugin "UserFeedback" qui gère les formulaires de feedback des utilisateurs. A partir de votre application, vous voudrez savoir quand un enregistrement Feedback a été enregistré et en définitive agir sur lui. Vous pourriez écouter l'événement global `Model.afterSave`. Cependant, vous pouvez utiliser une approche plus directe et écouter seulement l'événement dont vous avez réellement besoin :

```
// Vous pouvez créer ce qui suit avant l'opération de sauvegarde
// par exemple dans config/bootstrap.php
use Cake\ORM\TableRegistry;
// Si envoi d'emails
use Cake\Mailer\Email;

// Prior to 3.6 use TableRegistry::get('ThirdPartyPlugin.Feedbacks')
TableRegistry::getTableLocator()->get('ThirdPartyPlugin.Feedbacks')
    ->eventManager()
    ->on('Model.afterSave', function($event, $entity)
    {
        // Par exemple nous pouvons envoyer un email à l'admin
        // Avant 3.4, utilisez les méthodes from()/to()/subject()
        $email = new Email('default');
        $email->setFrom(['info@yoursite.com' => 'Your Site'])
            ->setTo('admin@yoursite.com')
            ->setSubject('New Feedback - Your Site')
            ->send('Body of message');
    });
```

Vous pouvez utiliser cette même approche pour lier les objets listener.

Interagir avec les Listeners Existants

En supposant que plusieurs écouteurs d'événements ont été enregistrés, la présence ou l'absence d'un modèle d'événements particulier peut être utilisé comme base de certaines actions :

```
// Attacher les écouteurs au EventManager.
$this->eventManager()->on('User.Registration', [$this, 'userRegistration']);
$this->eventManager()->on('User.Verification', [$this, 'userVerification']);
$this->eventManager()->on('User.Authorization', [$this, 'userAuthorization']);

// Quelque part ailleurs dans votre application.
$events = $this->eventManager()->matchingListeners('Verification');
```

(suite sur la page suivante)

(suite de la page précédente)

```
if (!empty($events)) {
    // Perform logic related to presence of 'Verification' event listener.
    // For example removing the listener if present.
    $this->eventManager()->off('User.Verification');
} else {
    // Logique liée à l'absence de l'écouteur d'événement 'Verification'
}
```

> **Note**
>
> Le modèle passé à la méthode `matchingListeners` n'est pas sensible à la casse.

Etablir des Priorités

Dans certains cas vous voulez contrôler la commande que les listeners appellent. Par exemple, si nous retournons à notre exemple des statistiques d'utilisateur. Ce serait idéal si le listener était appelé à la fin de la pile. En l'appelant à la fin de la pile, nous pouvons assurer que l'événement n'a pas été annulé et qu'aucun autre listener ne lève d'exception. Nous pouvons aussi obtenir l'état final des objets dans le cas où d'autres listeners ont modifié le sujet ou l'objet event.

Les priorités sont définies comme un nombre entier lors de l'ajout d'un listener. Plus le nombre est haut, plus la méthode sera lancée tardivement. La priorité par défaut pour tous les listeners est `10`. Si vous avez besoin que votre méthode soit lancée plus tôt, en utilisant toute valeur avant que celle par défaut ne fonctionne. D'un autre côté, si vous souhaitez lancer le callback après les autres, utiliser un nombre au-dessus de `10` le fera.

Si deux callbacks ont la même valeur de priorité, elles seront exécutées selon l'ordre dans lequel elles ont été attachées. Vous définissez les priorités en utilisant la méthode `on` pour les callbacks et en la déclarant dans la fonction `implementedEvents()` pour les listeners d'événement :

```
// Définir la priorité pour un callback
$callback = [$this, 'doSomething'];
$this->eventManager()->on(
    'Model.Orders.afterPlace',
    ['priority' => 2],
    $callback
);

// Définir la priorité pour un listener
class UserStatistic implements EventListener
{
    public function implementedEvents()
    {
        return [
            'Model.Orders.afterPlace' => [
                'callable' => 'updateBuyStatistic',
                'priority' => 100
            ],
        ];
    }
}
```

Comme vous le voyez, la principale différence pour les objets `EventListener` est que vous avez besoin d'utiliser un tableau pour spécifier la méthode callable et la préférence de priorité. La clé `callable` est une entrée de tableau spécial que le gestionnaire va lire pour savoir quelle fonction dans la classe il doit appeler.

Obtenir des Données d'Event en Paramètres de Fonction

Quand les événements ont des données fournies dans leur constructeur, les données fournies sont converties en arguments pour les listeners. Un exemple de la couche View est la callback afterRender :

```
$this->eventManager()
    ->dispatch(new Event('View.afterRender', $this, ['view' => $viewFileName]));
```

Les listeners de la callback `View.afterRender` doivent avoir la signature suivante :

```
function (Event $event, $viewFileName)
```

Chaque valeur fournie au constructeur d'Event sera convertie dans les paramètres de fonction afin qu'ils apparaissent dans le tableau de données. Si vous utilisez un tableau associatif, les résultats de `array_values` vont déterminer l'ordre des arguments de la fonction.

> **Note**
>
> Au contraire de 2.x, convertir les données d'événement en arguments du listener est le comportement par défaut et ne peut pas être désactivé.

Dispatcher les Events

Une fois que vous avez obtenu une instance du gestionnaire d'événements, vous pouvez dispatcher les événements en utilisant `dispatch()`. Cette méthode prend une instance de la classe `Cake\Event\Event`. Regardons le dispatch d'un événement :

```
// Crée un nouvel événement et le dispatch.
$event = new Event('Model.Orders.afterPlace', $this, [
    'order' => $order
]);
$this->eventManager()->dispatch($event);
```

`Cake\Event\Event` accepte 3 arguments dans son constructeur. Le premier est le nom de l'événement, vous devriez essayer de garder ce nom aussi unique que possible, en le rendant lisible. Nous vous suggérons une convention comme suit : `Layer.eventName` pour les événements généraux qui arrivent au niveau couche (par ex `Controller.startup`, `View.beforeRender`) et `Layer.Class.eventName` pour les événements qui arrivent dans des classes spécifiques sur une couche, par exemple `Model.User.afterRegister` ou `Controller.Courses.invalidAccess`.

Le deuxième argument est le `subject`, c'est à dire l'objet associé à l'événement, comme une classe attrape les événements sur elle-même, utiliser `$this` sera le cas le plus commun. Même si un `Component` peut aussi déclencher les événements d'un controller. La classe subject est importante parce que les écouteurs auront un accès immédiat aux propriétés de l'objet et pourront les inspecter ou les changer à la volée.

Au final, le troisième argument est une donnée d'événement supplémentaire. Ceci peut être toute donnée que vous considérez utile de passer pour que les écouteurs puissent agir sur eux. Alors que ceci peut être un argument de tout type, nous vous recommandons de passer un tableau associatif.

La méthode `dispatch()` accepte un objet event en argument et notifie à tous les écouteurs qui sont abonnés.

Stopper les Events

Un peu comme les événements DOM, vous voulez peut-être stopper un événement pour éviter aux autres listeners d'être notifiés. Vous pouvez voir ceci pendant les callbacks de mode(par ex beforeSave) dans lesquels il est possible de stopper l'opération de sauvegarde si le code détecte qu'il ne peut pas continuer.

Afin de stopper les événements, vous pouvez soit retourner `false` dans vos callbacks ou appeler la méthode `stopPropagation()` sur l'objet event :

```
public function doSomething($event)
{
    // ...
    return false; // stoppe l'event
}

public function updateBuyStatistic($event)
{
    // ...
    $event->stopPropagation();
}
```

Stopper un événement va éviter à toute callback supplémentaire d'être appelée. En plus, le code attrapant l'événement peut se comporter différemment selon que l'événement est stoppé ou non. Généralement il n'est pas sensé stopper "après" les événements, mais stopper "avant" les événements est souvent utilisé pour empêcher toutes les opérations de se passer.

Pour vérifier si un événement a été stoppé, vous appelez la méthode `isStopped()` dans l'objet event :

```
public function place($order)
{
    $event = new Event('Model.Orders.beforePlace', $this, ['order' => $order]);
    $this->eventManager()->dispatch($event);
    if ($event->isStopped()) {
        return false;
    }
    if ($this->Orders->save($order)) {
        // ...
    }
    // ...
}
```

Dans l'exemple précédent, la commande ne serait pas sauvegardée si l'événement est stoppé pendant le processus `beforePlace`.

Obtenir des Résultats d'Evenement

A chaque fois qu'un callback retourne une valeur non nulle et non false, elle sera stockée dans la propriété `$result` de l'objet event. C'est utile quand vous voulez permettre aux callbacks de modifier l'exécution de l'événement. Prenons à nouveau notre exemple `beforePlace` et laissons les callbacks modifier la donnée `$order`.

Les résultats d'Event peuvent être modifiés soit en utilisant directement la propriété de résultat de l'objet event, soit en retournant la valeur dans le callback elle-même :

```
// Un callback listener
public function doSomething($event)
{
```

(suite sur la page suivante)

(suite de la page précédente)

```
    // ...
    $alteredData = $event->getData('order') + $moreData;

    return $alteredData;
}

// Un autre callback listener
public function doSomethingElse($event)
{
    // ...
    $event->setResult(['order' => $alteredData] + $this->result());
}

// Utiliser les résultats d'event
public function place($order)
{
    $event = new Event('Model.Orders.beforePlace', $this, ['order' => $order]);
    $this->eventManager()->dispatch($event);
    if (!empty($event->getResult()['order'])) {
        $order = $event->getResult()['order'];
    }
    if ($this->Orders->save($order)) {
        // ...
    }
    // ...
}
```

Il est possible de modifier toute propriété d'un objet event et d'avoir les nouvelles données passées à la prochaine callback. Dans la plupart des cas, fournir des objets en données d'event ou en résultat et directement modifier l'objet est la meilleure solution puisque la référence est la même et les modifications sont partagées à travers tous les appels de callback.

Retirer les Callbacks et les Listeners

Si pour certaines raisons, vous voulez retirer toute callback d'un gestionnaire d'événements, appelez seulement la méthode `Cake\Event\EventManager::off()` en utilisant des arguments les deux premiers paramètres que vous utilisiez pour l'attacher :

```
// Attacher une fonction
$this->eventManager()->on('My.event', [$this, 'doSomething']);

// Détacher une fonction
$this->eventManager()->off([$this, 'doSomething']);

// Attacher une fonction anonyme.
$myFunction = function ($event) { ... };
$this->eventManager()->on('My.event', $myFunction);

// Détacher la fonction anonyme
$this->eventManager()->off('My.event', $myFunction);

// Attacher un EventListener
```

(suite sur la page suivante)

(suite de la page précédente)

```
$listener = new MyEventLister();
$this->eventManager()->on($listener);

// Détacher une clé d'événement unique d'un listener
$this->eventManager()->off('My.event', $listener);

// Détacher tous les callbacks intégrés par un listener
$this->eventManager()->off($listener);
```

Les événements sont une bonne façon de séparer les préoccupations dans votre application et rend les classes à la fois cohérentes et découplées des autres, néanmoins l'utilisation des événements n'est pas la solution à tous les problèmes. Les Events peuvent être utilisés pour découpler le code de l'application et rendre les plugins extensibles.

Gardez à l'esprit que beaucoup de pouvoir implique beaucoup de responsabilité. Utiliser trop d'événements peut rendre le debug plus difficile et nécessiter des tests d'intégration supplémentaires.

Lecture Supplémentaire

- *Behaviors (Comportements)*
- *Components (Composants)*
- *Helpers (Assistants)*
- *Tester les Events*

CHAPITRE 21

Internationalisation & Localisation

L'une des meilleures façons pour qu'une application ait une audience plus large est de gérer plusieurs langues. Cela peut souvent se révéler être une tâche gigantesque, mais les fonctionnalités d'internationalisation et de localisation dans CakePHP rendront cela plus facile.

D'abord il est important de comprendre quelques terminologies. *Internationalisation* se réfère à la possibilité qu'a une application d'être localisée. Le terme *localisation* se réfère à l'adaptation qu'a une application de répondre aux besoins d'une langue (ou culture) spécifique (par ex : un « locale »). L'internationalisation et la localisation sont souvent abrégées en respectivement i18n et l10n ; 18 et 10 correspondent au nombre de caractères entre le premier et le dernier caractère respectivement pour internationalisation et localisation.

Internationaliser Votre Application

Il n'y a que quelques étapes à franchir pour passer d'une application mono-langue à une application multi-langue, la première est d'utiliser la fonction `__()` dans votre code. Ci-dessous un exemple d'un code pour une application mono-langue :

```
<h2>Popular Articles</h2>
```

Pour internationaliser votre code, la seule chose à faire est d'entourer la chaîne avec `__()` comme ceci :

```
<h2><?= __('Popular Articles') ?></h2>
```

Si vous ne faîtes rien de plus, ces deux bouts de codes donneront un résultat identique - ils renverront le même contenu au navigateur. La fonction `__()` traduira la chaîne passée si une traduction est disponible, sinon elle la renverra non modifiée.

Fichiers de Langues

Les traductions peuvent être mises à disposition en utilisant des fichiers de langue stockés dans votre application. Le format par défaut pour ces fichiers est le format Gettext [151]. Ces fichiers doivent être placés dans **src/Locale/** et dans ce répertoire, il devrait y avoir un sous-dossier par langue que l'application doit prendre en charge :

```
/src
    /Locale
        /en_US
            default.po
        /en_GB
            default.po
            validation.po
        /es
            default.po
```

Le domaine par défaut est "default", votre dossier `locale` devrait donc contenir au minimum le fichier `default.po` (cf. ci-dessus). Un domaine se réfère à un regroupement arbitraire de messages de traduction. Si aucun groupe n'est utilisé, le groupe par défaut est sélectionné.

Les messages du coeur extraits de la libairie CakePHP peuvent être stockés séparément dans un fichier **cake.po** dans **src/Locale/**. La librairie localized de CakePHP [152] contient des traductions des chaînes de caractère du coeur (du domaine cake) pour l'interface client. Pour utiliser ces fichiers, liez les ou copiez les au bon endroit : **src/Locale/<locale>/cake.po**. Si votre locale est incomplète ou incorrecte, vous pouvez nous envoyer une PR dans ce dépôt pour corriger les erreurs.

Les plugins peuvent également contenir des fichiers de traduction, la convention est d'utiliser la version `under_scored` du nom du plugin comme domaine de la traduction des messages :

```
MyPlugin
    /src
        /Locale
            /fr
                my_plugin.po
            /de
                my_plugin.po
```

Les dossiers de traduction peuvent être composées d'un code à deux ou trois lettres ISO de la langue ou du nom de la locale ICU, par exemple `fr_FR`, `es_AR`, `da_DK`, qui contient en même temps la langue et le pays où elle est parlée.

La liste complète est disponible sur https://www.localeplanet.com/icu/.

Un fichier de traduction pourrait ressembler à ceci :

```
msgid "My name is {0}"
msgstr "Je m'appelle {0}"

msgid "I'm {0,number} years old"
msgstr "J'ai {0,number} ans"
```

Note

Les traductions sont mises en cache - Assurez-vous de toujours vider le cache après avoir apporté des modifications aux traductions ! Vous pouvez soit utiliser *l'outil cache* et exécuter par exemple `bin/cake cache clear`

151. https://en.wikipedia.org/wiki/Gettext
152. https://github.com/cakephp/localized

_cake_core_, soit vider manuellement le dossier `tmp/cache/persistent` (si vous utilisez une mise en cache basée sur des fichiers).

Extraire les Fichiers Pot avec le Shell I18n

Pour créer les fichiers pot à partir de *__()* et des autres types de messages internationalisés qui se trouvent dans votre code, vous pouvez utiliser le shell i18n. Vous pouvez consulter le *chapitre suivant* pour en savoir plus.

Définir la Locale par Défaut

La `locale` par défaut se détermine dans le fichier **config/app.php** en définissant `App.defaultLocale` :

```
'App' => [
    ...
    'defaultLocale' => env('APP_DEFAULT_LOCALE', 'en_US'),
    ...
]
```

Cela permet de contrôler plusieurs aspects de votre application, incluant la langue de traduction par défaut, le format des dates, des nombres, et devises à chaque fois qu'un de ces éléments s'affiche, en utilisant les bibliothèques de localisation fournies par CakePHP.

Modifier la Locale pendant l'Exécution

Pour changer la langue des chaines de caractères traduites, vous pouvez appeler cette méthode :

```
use Cake\I18n\I18n;

// Avant 3.5, utilisez I18n::locale()
I18n::setLocale('de_DE');
```

Cela changera également le formatage des nombres et des dates lorsque vous utilisez les outils de localisation.

Utiliser les Fonctions de Traduction

CakePHP fournit plusieurs fonctions qui vous aideront à internationaliser votre application. La plus fréquemment utilisée est `__()`. Cette fonction est utilisée pour récupérer un message de traduction simple ou retourner la même chaine si aucune traduction n'est trouvée :

```
echo __('Popular Articles');
```

Si vous avez besoin de grouper vos messages, par exemple des traduction à l'intérieur d'un plugin, vous pouvez utiliser la fonction *__d()* pour récupérer les messages d'un autre domaine :

```
echo __d('my_plugin', 'Trending right now');
```

Note

Si vous souhaitez traduire vos plugins et qu'ils ont un « préfixe » de namespace, vous devez nommer votre chaîne de domaine `Namespace/PluginName`. Cependant, notez que le chemin du fichier de langage sera, dans votre dossier de plugin, `plugins/Namespace/PluginName/src/Locale/plugin_name.po`.

Parfois les chaines de traduction peuvent être ambigües pour les personnes les traduisant. Cela se produit lorsque deux chaines sont identiques mais se réfèrent à des choses différentes. Par exemple "lettre" a plusieurs significations en français. Pour résoudre ce problème, vous pouvez utiliser la fonction `__x()` :

```
echo __x('communication écrite', 'He read the first letter');

echo __x('apprentissage de l alphabet', 'He read the first letter');
```

Le premier argument est le contexte du message et le second est le message à traduire.

```
msgctxt "communication écrite"
msgid "He read the first letter"
msgstr "Il a lu le premier courrier"
```

Utiliser des Variables dans les Traductions de Messages

Les fonctions de traduction vous permettent d'interpoler des variables dans les messages en utilisant des marqueurs définis dans le message lui-même ou dans la chaine traduite :

```
echo __("Hello, my name is {0}, I'm {1} years old", ['Sara', 12]);
```

Les marqueurs sont numériques et correspondent aux clés dans le tableau passé. Vous pouvez également passer à la fonction les variables en tant qu'arguments indépendants :

```
echo __("Small step for {0}, Big leap for {1}", 'Man', 'Humanity');
```

Toutes les fonctions de traduction intègrent le remplacement de placeholder :

```
__d('validation', 'The field {0} cannot be left empty', 'Name');

__x('alphabet', 'He read the letter {0}', 'Z');
```

le caractère `'` (guillemet simple ou apostrophe) agit comme un caractère d'échappement dans les messages de traduction. Chaque variable entourée de guillemets simples ne sera pas remplacée et sera traitée en tant que texte littéral. Par exemple :

```
__("This variable '{0}' be replaced.", 'will not');
```

En utilisant deux guillemets simples côte à côte, vos variables seront remplacées correctement :

```
__("This variable ''{0}'' be replaced.", 'will');
```

Ces fonctions profitent des avantages du MessageFormatter ICU [153] pour que vous puissiez traduire des messages, des dates, des nombres et des devises en même temps :

```
echo __(
    'Hi {0}, your balance on the {1,date} is {2,number,currency}',
    ['Charles', new FrozenTime('2014-01-13 11:12:00'), 1354.37]
);

// Retourne
Hi Charles, your balance on the Jan 13, 2014, 11:12 AM is $ 1,354.37
```

Les nombres dans les placeholders peuvent également être formatés avec un contrôle fin et précis sur la sortie :

153. https://php.net/manual/fr/messageformatter.format.php

```
echo __(
    'You have traveled {0,number} kilometers in {1,number,integer} weeks',
    [5423.344, 5.1]
);

// Retourne
You have traveled 5,423.34 kilometers in 5 weeks

echo __('There are {0,number,#,###} people on earth', 6.1 * pow(10, 9));

// Retourne
There are 6,100,000,000 people on earth
```

Voici la liste des balises spécifiques que vous pouvez mettre après le mot `number` :

- `integer` : Supprime la partie décimale
- `currency` : Ajoute le symbole de la devise locale et arrondit les décimales
- `percent` : Formate le nombre en pourcentage

Les dates peuvent également être formatées en utilisant le mot `date` après le nombre placeholder. Les options supplémentaires sont les suivantes :

- `short`
- `medium`
- `long`
- `full`

Le mot `time` après le nombre placeholder est également accepté et il comprend les mêmes options que `date`.

> **Note**
>
> Les placeholders nommés sont supportés dans PHP 5.5+ et sont formatés comme `{name}`. Quand vous utilisez les placeholders nommés, passez les variables dans un tableau en utilisant la paire de clé/valeur, par exemple `['name' => 'Sara', 'age' => 12]`.
>
> Il est recommandé d'utiliser PHP 5.5 ou supérieur quand vous utilisez les fonctionnalités d'internationalisation de CakePHP. L'extension `php5-intl` doit être installée et la version ICU doit être supérieur à 48.x.y (pour vérifier la version ICU `Intl::getIcuVersion()`).

Pluriels

Une partie cruciale de l'internationalisation de votre application est de récupérer vos messages pluralisés correctement suivant les langues affichées. CakePHP fournit plusieurs possibilités pour sélectionner correctement les pluriels dans vos messages.

Utiliser la Sélection Plurielle ICU

La première tire parti du format de message `ICU` qui est fourni par défaut dans les fonctions de traductions. Dans les fichiers de traduction vous pourriez avoir les chaines suivantes

```
msgid "{0,plural,=0{No records found} =1{Found 1 record} other{Found # records}}"
msgstr "{0,plural,=0{Ningún resultado} =1{1 resultado} other{# resultados}}"

msgid "{placeholder,plural,=0{No records found} =1{Found 1 record} other{Found {1}␣
↪records}}"
msgstr "{placeholder,plural,=0{Ningún resultado} =1{1 resultado} other{{1} resultados}}"
```

Et dans votre application utilisez le code suivant pour afficher l'une des traductions pour une telle chaine :

```
__('{0,plural,=0{No records found }=1{Found 1 record} other{Found # records}}', [0]);

// Retourne "Ningún resultado" puisque l'argument {0} est 0

__('{0,plural,=0{No records found} =1{Found 1 record} other{Found # records}}', [1]);

// Retourne "1 resultado" puisque l'argument {0} est 1

__('{placeholder,plural,=0{No records found} =1{Found 1 record} other{Found {1} records}}
↪', [0, 'many', 'placeholder' => 2])
// Retourne "many resultados" puisque l'argument {placeholder} est 2 et
// l'argument {1} est 'many'
```

Regarder de plus près le format que nous avons juste utilisé, rendra évident la méthode de construction des messages :

```
{ [count placeholder],plural, case1{message} case2{message} case3{...} ... }
```

Le `[count placeholder]` peut être le numéro de clé du tableau de n'importe quelle variable passée à la fonction de traduction. Il sera utilisé pour sélectionner la forme plurielle correcte.

Noter que pour faire référence à `[count placeholder]` dans `{message}` vous devez utiliser `#`.

Vous pouvez bien entendu utiliser des id de messages plus simples si vous ne voulez pas taper la séquence plurielle complète dans votre code.

```
msgid "search.results"
msgstr "{0,plural,=0{Ningún resultado} =1{1 resultado} other{{1} resultados}}"
```

Ensuite utilisez la nouvelle chaine dans votre code :

```
__('search.results', [2, 2]);

// Retourne "2 resultados"
```

la dernière version a l'inconvénient que vous aurez besoin d'avoir un fichier de message de traduction même pour la langue par défaut, mais comporte l'avantage de rendre le code plus lisible et de laisser les chaines de sélection de plurielles compliquées dans les fichiers de traduction.

Parfois utiliser directement la correspondance des nombres vers les pluriels est impossible. Par exemple les langues telles que l'Arabe nécessitent un pluriel différent lorsque vous faites référence à une faible quantité et un pluriel différent pour une quantité plus importante. Dans ces cas vous pouvez utiliser la correspondance d'alias ICU. Au lieu d'écrire :

```
=0{No results} =1{...} other{...}
```

Vous pouvez faire :

```
zero{No Results} one{One result} few{...} many{...} other{...}
```

Assurez-vous de lire le Guide des Règles Plurielles des Langues [154] pour obtenir une vue d'ensemble complète des alias que vous pouvez utiliser pour chaque langue.

154. https://www.unicode.org/cldr/charts/latest/supplemental/language_plural_rules.html

Utiliser la Sélection Plurielle Gettext

Le second format de sélection plurielle accepté est d'utiliser les fonctionnalités intégrées de Gettext. Dans ce cas, les pluriels seront enregistrés dans le fichier `.po` en créant une ligne de traduction séparée pour chaque forme plurielle :

```
# Un identificateur de message pour le singulier
msgid "One file removed"
# Une autre pour le pluriel
msgid_plural "{0} files removed"
# Traduction au singulier
msgstr[0] "Un fichero eliminado"
# Traduction au pluriel
msgstr[1] "{0} ficheros eliminados"
```

Lorsque vous utilisez cet autre format, vous devez utiliser une autre fonction de traduction :

```
// Retourne: "10 ficheros eliminados"
$count = 10;
__n('One file removed', '{0} files removed', $count, $count);

// Il est également possible de l'utiliser dans un domaine
__dn('my_plugin', 'One file removed', '{0} files removed', $count, $count);
```

Le nombre à l'intérieur de `msgstr[]` est le nombre assigné par Gettext pour la forme plurielle de la langue. Certaines langues ont plus de deux formes plurielles, le Croate par exemple :

```
msgid "One file removed"
msgid_plural "{0} files removed"
msgstr[0] "{0} datoteka je uklonjena"
msgstr[1] "{0} datoteke su uklonjene"
msgstr[2] "{0} datoteka je uklonjeno"
```

Merci de visiter la page des langues Launchpad [155] pour une explication détaillée sur les nombres de formes plurielles de chaque langue.

Créer Vos Propres Traducteurs

Si vous devez vous écarter des conventions de CakePHP en ce qui concerne l'emplacement et la manière d'enregistrer les messages de traduction, vous pouvez créer votre propre loader de messages traduits. La manière la plus simple de créer votre propre traducteur est de définir un loader pour un seul domaine et une seule locale :

```
use Aura\Intl\Package;

I18n::setTranslator('animals', function () {
    $package = new Package(
        'default', // The formatting strategy (ICU)
        'default' // The fallback domain
    );
    $package->setMessages([
        'Dog' => 'Chien',
        'Cat' => 'Chat',
        'Bird' => 'Oiseau'
```

(suite sur la page suivante)

155. https://translations.launchpad.net/+languages

(suite de la page précédente)

```
        ...
    ]);

    return $package;
}, 'fr_FR');
```

Le code ci-dessus peut être ajouté à votre **config/bootstrap.php** pour que les traductions soient ajoutées avant qu'une fonction de traduction ne soit utilisée. Le minimum absolu nécessaire pour créer un traducteur est que la fonction loader doit retourner un objet `Aura\Intl\Package`. Une fois que le code est en place vous pouvez utiliser les fonctions de traduction comme d'habitude :

```
// Avant 3.5, utilisez I18n::locale()
I18n::setLocale('fr_FR');
__d('animals', 'Dog'); // Retourne "Chien"
```

Comme vous pouvez le voir, les objets `Package` prennent les messages de traduction sous forme de tableau. Vous pouvez passer la méthode `setMessages()` de la manière qui vous plait : avec du code en ligne, en incluant un autre fichier, en appelant une autre fonction, etc. CakePHP fournit quelques fonctions de loader que vous pouvez réutiliser si vous devez juste changer l'endroit où sont chargés les messages. Par exemple, vous pouvez toujours utiliser les fichiers `.po` mais les charger depuis un autre endroit :

```
use Cake\I18n\MessagesFileLoader as Loader;

// Charge les messages depuis src/Locale/folder/sub_folder/filename.po

I18n::setTranslator(
    'animals',
    new Loader('filename', 'folder/sub_folder', 'po'),
    'fr_FR'
);
```

Créer des Parsers de Messages

Il est possible de continuer à utiliser les mêmes conventions que CakePHP utilise mais d'utiliser un autre parser de messages que `PoFileParser`. Par exemple, si vous vouliez charger les messages de traduction en utilisant `YAML`, vous auriez d'abord besoin de créer la classe du parser :

```
namespace App\I18n\Parser;

class YamlFileParser
{

    public function parse($file)
    {
        return yaml_parse_file($file);
    }
}
```

Le fichier doit être créé dans le dossier **src/I18n/Parser** de votre application. Ensuite, créez les fichiers de traduction sous **src/Locale/fr_FR/animals.yaml**

```
Dog: Chien
Cat: Chat
Bird: Oiseau
```

Enfin, configurez le loader de traduction pour le domaine et la locale :

```
use Cake\I18n\MessagesFileLoader as Loader;

// Avant 3.5, utilisez translator()
I18n::setTranslator(
    'animals',
    new Loader('animals', 'fr_FR', 'yaml'),
    'fr_FR'
);
```

Créer des Traducteurs Génériques

Configurer des traducteurs en appelant `I18n::setTranslator()` pour chaque domaine et locale que vous devez supporter peut être fastidieux, spécialement si vous devez supporter plus que quelques locales. Pour éviter ce problème, CakePHP vous permet de définir des loaders de traduction génériques pour chaque domaine.

Imaginez que vous vouliez charger toutes les traductions pour le domaine par défaut et pour chaque langue depuis un service externe :

```
use Aura\Intl\Package;

I18n::config('default', function ($domain, $locale) {
    $locale = Locale::parseLocale($locale);
    $language = $locale['language'];
    $messages = file_get_contents("http://example.com/translations/$lang.json");

    return new Package(
        'default', // Formatter
        null, // Fallback (none for default domain)
        json_decode($messages, true)
    )
});
```

Le code ci-dessus appelle un service externe exemple pour charger un fichier JSON avec les traductions puis construit uniquement un objet `Package` pour chaque locale nécessaire dans l'application.

If you'd like to change how packages are loaded for all packages, that don't have specific loaders set you can replace the fallback package loader by using the `_fallback` package :

```
I18n::config('_fallback', function ($domain, $locale) {
    // Custom code that yields a package here.
});
```

Pluriels et Contexte dans les Traducteurs Personnalisés

les tableaux utilisées pour `setMessages()` peuvent être conçus pour ordonner au traducteur d'enregistrer les messages sous différents domaines ou de déclencher une sélection de pluriel de style Gettext. Ce qui suit est un exemple d'enregistrement de traductions pour la même clé dans différents contextes :

```
[
    'He reads the letter {0}' => [
        'alphabet' => 'Él lee la letra {0}',
        'written communication' => 'Él lee la carta {0}'
    ]
]
```

De même vous pouvez exprimer des pluriels de style Gettext en utilisant le tableau de messages avec une clé de tableau imbriqué par forme plurielle :

```
[
    'I have read one book' => 'He leído un libro',
    'I have read {0} books' => [
        'He leído un libro',
        'He leído {0} libros'
    ]
]
```

Utiliser des Formateurs Différents

Dans les exemples précédents nous avons vu que les Packages sont construits en utilisant `default` en premier argument, et il était indiqué avec un commentaire qu'il correspondait au formateur à utiliser. Les formateurs sont des classes responsables d'interpoler les variables dans les messages de traduction et sélectionner la forme plurielle correcte.

Si vous avez à faire une application datée, ou que vous n'avez pas besoin de la puissance offerte par le formateur de message ICU, CakePHP fournit également le formateur `sprintf` :

```
return Package('sprintf', 'fallback_domain', $messages);
```

Les messages à traduire seront passés à la fonction `sprintf()` pour interpoler les variables :

```
__('Hello, my name is %s and I am %d years old', 'José', 29);
```

Il est possible de définir le formateur par défaut pour tous les traducteurs créés par CakePHP avant qu'ils soient utilisés pour la première fois. Cela n'inclut pas les traducteurs créés manuellement en utilisant les méthodes `setTranslator()` et `config()` :

```
I18n::defaultFormatter('sprintf');
```

Localiser les Dates et les Nombres

Lorsque vous affichez des dates et des nombres dans votre application, vous voudrez souvent qu'elles soient formatées conformément au format du pays ou de la région dans lequel vous souhaitez afficher la page.

Pour changer l'affichage des dates et des nombres, vous devez uniquement changer la locale et utiliser les bonnes classes :

```
use Cake\I18n\I18n;
use Cake\I18n\Time;
use Cake\I18n\Number;

// Avant 3.5, utilisez I18n::locale()
I18n::setLocale('fr-FR');
```

(suite sur la page suivante)

(suite de la page précédente)

```
$date = new Time('2015-04-05 23:00:00');

echo $date; // Affiche 05/04/2015 23:00

echo Number::format(524.23); // Displays 524,23
```

Assurez vous de lire les sections *Date & Time* et *Number* pour en apprendre plus sur les options de formatage.

Par défaut, les dates renvoyées par l'ORM utilisent la classe `Cake\I18n\Time`, donc leur affichage direct dans votre application sera affecté par le changement de la locale.

Parser les Données Datetime Localisées

Quand vous acceptez les données localisées, c'est sympa d'accepter les informations de type datetime dans un format localisé pour l'utilisateur. Dans un controller, ou /controllers/middleware, vous pouvez configurer les types Date, Time, et DateTime pour parser les formats localisés :

```
use Cake\Database\TypeFactory;

// Permet de parser avec le format de locale par défaut.
TypeFactory::build('datetime')->useLocaleParser();

// Configure un parser personnalisé du format de datetime.
TypeFactory::build('datetime')->useLocaleParser()->setLocaleFormat('dd-M-y');

// Vous pouvez aussi utiliser les constantes IntlDateFormatter.
TypeFactory::build('datetime')->useLocaleParser()
    ->setLocaleFormat([IntlDateFormatter::SHORT, -1]);
```

Le parsing du format par défaut est le même que le format de chaîne par défaut.

Convertir des Données de Requêtes du Fuseau Horaire de l'Utilisateur

Quand vous gérez des données d'utilisateurs situés dans différents fuseaux horaires, vous avez besoin de convertir ces datetimes de la requête vers le fuseau horaire de votre application. Vous pouvez utiliser `setUserTimezone()` depuis un controller ou /controllers/middleware pour faire cela plus simplement :

```
// Définir le fuseau horaire de l'utilisateur
TypeFactory::build('datetime')->setUserTimezone($user->timezone);
```

Une fois que c'est fait, quand votre application crée ou met à jour des entities à partir de la requête, l'ORM va convertir automatiquement les valeurs datetime du fuseau de l'utilisateur vers le fuseau de votre application. Cela garantit que l'application travaillera toujours dans le fuseau horaire défini dans `App.defaultTimezone`.

Si votre application gère des données datetime dans de nombreuses actions, vous pouvez utiliser un middleware pour définir à la fois la conversion du décalage horaire et le format localisé :

```
namespace App\Middleware;

use Cake\Database\TypeFactory;
use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;
use Psr\Http\Server\MiddlewareInterface;
```

(suite sur la page suivante)

(suite de la page précédente)

```
use Psr\Http\Server\RequestHandlerInterface;

class DatetimeMiddleare implements MiddlewareInterface
{
    public function process(
        ServerRequestInterface $request,
        RequestHandlerInterface $handler
    ): ResponseInterface {
        // Obtenir l'utilisateur depuis la requête.
        // Cet exemple suppose que votre entity utilisteur a un attribut timezone.
        $user = $request->getAttribute('identity');
        if ($user) {
            TypeFactory::build('datetime')
                ->useLocaleParser()
                ->setUserTimezone($user->timezone);
        }

        return $handler->handle($request);
    }
}
```

Sélection Automatique de Locale Basée sur les Données de Requêtes

En utilisant le `LocaleSelectorFilter` dans votre application, CakePHP définira automatiquement la locale en se basant sur l'utilisateur actuel :

```
// dans src/Application.php
use Cake\I18n\Middleware\LocaleSelectorMiddleware;

// Mise à jour de la méthode "middleware" pour ajouter le nouveau middleware
public function middleware($middleware)
{
    // Add middleware and set the valid locales
    $middleware->add(new LocaleSelectorMiddleware(['en_US', 'fr_FR']));
}

// Avant 3.3.0, il faut utiliser un DispatchFilter
// dans config/bootstrap.php
DispatcherFactory::add('LocaleSelector');

// Limite les locales à en_US et fr_FR uniquement
DispatcherFactory::add('LocaleSelector', ['locales' => ['en_US', 'fr_FR']]);
```

Le `LocaleSelectorFilter` utilisera l'entête `Accept-Language` pour définir automatiquement la locale préférée de l'utilisateur. Vous pouvez utiliser l'option de liste de locale pour limiter les locales pouvant être utilisées automatiquement.

Translate Content/Entities

Si vous voulez traduire du contenu ou des entities, vous pouvez consulter le *Behavior Translate*.

CHAPITRE 22

Journalisation (logging)

Bien que les réglages de la Classe Configure du cœur de CakePHP puissent vraiment vous aider à voir ce qui se passe sous le capot, vous aurez besoin certaines fois de consigner des données sur le disque pour découvrir ce qui se produit. Avec des technologies comme SOAP, AJAX, et les APIs REST, déboguer peut s'avérer difficile.

Le logging (journalisation) peut aussi être une façon de découvrir ce qui s'est passé dans votre application à chaque instant. Quels termes de recherche ont été utilisés ? Quelles sortes d'erreurs ont été vues par mes utilisateurs ? A quelle fréquence est exécutée une requête particulière ?

La journalisation des données dans CakePHP passe par la fonction log() fournie par `LogTrait`, qui est l'ancêtre commun de beaucoup de classes CakePHP. Si le contexte est une classe CakePHP (Controller, Component, View…), vous pouvez loguer (journaliser) vos données. Vous pouvez aussi utiliser `Log::write()` directement. Consultez *Ecrire dans les logs*.

Configuration des flux d'un log (journal)

La configuration de `Log` doit être faite pendant la phase de bootstrap de votre application. Le fichier **config/app.php** est justement prévu pour ceci. Vous pouvez définir autant de jounaux que votre application nécessite. Les journaux doivent être configurés en utilisant *`Cake\Log\Log`*. Un exemple serait :

```
use Cake\Log\Engine\FileLog;
use Cake\Log\Log;

// Nom de classe à partir de la constante 'class' du logger
Log::setConfig('info', [
    'className' => FileLog::class,
    'path' => LOGS,
    'levels' => ['info'],
    'file' => 'info',
]);
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Nom de classe court
Log::setConfig('debug', [
    'className' => 'File',
    'path' => LOGS,
    'levels' => ['notice', 'info', 'debug'],
    'file' => 'debug',
]);

// Nom avec le namespace complet.
Log::setConfig('error', [
    'className' => 'Cake\Log\Engine\FileLog',
    'path' => LOGS,
    'levels' => ['warning', 'error', 'critical', 'alert', 'emergency'],
    'file' => 'error',
]);
```

Le code ci-dessus crée deux journaux. Un appelé `debug`, l'autre appelé `error`. Chacun est configuré pour gérer différents niveaux de message. Ils stockent aussi leurs messages de journal dans des fichiers séparés, ainsi il est possible de séparer les logs de debug/notice/info des erreurs plus sérieuses. Regardez la section sur *Utiliser les Niveaux* pour plus d'informations sur les différents niveaux et ce qu'ils signifient.

Une fois qu'une configuration est créée, vous ne pouvez pas la changer. A la place, vous devez retirer la configuration et la re-créer en utilisant *Cake\Log\Log::drop()* et *Cake\Log\Log::setConfig()*.

Il est aussi possible de créer des loggers en fournissant une closure. C'est utile quand vous devez avoir un contrôle complet sur la façon dont l'objet est construit. La closure doit retourner l'instance de logger construite. Par exemple :

```
Log::setConfig('special', function () {
    return new \Cake\Log\Engine\FileLog(['path' => LOGS, 'file' => 'log']);
});
```

Les options de configuration peuvent également être fournies en tant que chaine *DSN*. C'est utile lorsque vous travaillez avec des variables d'environnement ou des fournisseurs *PaaS* :

```
Log::setConfig('error', [
    'url' => 'file:///full/path/to/logs/?levels[]=warning&levels[]=error&file=error',
]);
```

Avertissement

Si vous ne configurez pas les moteurs de logs, les messages de log ne seront pas enregistrés.

Journalisation des Erreurs et des Exception

Les erreurs et les exception peuvent elles aussi être journalisées. En configurant les valeurs correspondantes dans votre fichier **config/app.php**. Les erreurs seront affichées quand debug est à `true` et loguées quand debug est à `false`. Définissez l'option `log` à `true` pour loguer les exceptions non capturées. Consultez *Configuration* pour plus d'information.

Ecrire dans les logs

Ecrire dans les fichiers peut être réalisé de deux façons. La première est d'utiliser la méthode statique *Cake\Log\Log::write()* :

```
Log::write('debug', 'Quelque chose ne fonctionne pas');
```

La seconde est d'utiliser la fonction raccourcie `log()` disponible dans chacune des classes qui utilisent `LogTrait`. En appelant `log()` cela appellera en interne `Log::write()` :

```
// Exécuter cela dans une classe qui utilise LogTrait:
$this->log("Quelque chose ne fonctionne pas!", 'debug');
```

Tous les flux de log configurés sont écrits séquentiellement à chaque fois que *Cake\Log\Log::write()* est appelée. Si vous n'avez pas configuré de moteurs de log, `log()` va retourner false et aucun message de log ne sera écrit.

Utiliser des Placeholders dans les Messages

Si vous avez besoin de loguer des données définies dynamiquement, vous pouvez utiliser des placeholders dans vos messages de log et fournir un tableau de paires clé/valeur dans le paramètre `$context` :

```
// Enverra le log `Traitement impossible pour userid=1`
Log::write('error', 'Traitement impossible pour userid={user}', ['user' => $user->id]);
```

Les placeholders pour lesquels aucune clé n'a été définie ne seront pas remplacés. Si vous avez besoin d'utiliser des mots entre accolades, vous devez les échapper :

```
// Enverra le log `Pas de {remplacement}`
Log::write('error', 'Pas de \\{remplacement}', ['remplacement' => 'no']);
```

Si vous incluez des objets dans vos placeholders de logs, ces objets devront implémenter une des méthodes suivantes :

- `__toString()`
- `toArray()`
- `__debugInfo()`

Utiliser les Niveaux

CakePHP prend en charge les niveaux de log standards définis par POSIX. Chaque niveau représente un niveau plus fort de sévérité :

- Emergency : system is inutilisable
- Alert : l'action doit être prise immédiatement
- Critical : Conditions critiques
- Error : conditions d'erreurs
- Warning : conditions d'avertissements
- Notice : condition normale mais importante
- Info : messages d'information
- Debug : messages de niveau-debug

Vous pouvez vous référer à ces niveaux par nom en configurant les journaux, et lors de l'écriture des messages de log. Sinon vous pouvez utiliser des méthodes pratiques comme *Cake\Log\Log::error()* pour indiquer clairement le niveau de journalisation. Utiliser un niveau qui n'est pas dans les niveaux ci-dessus va entraîner une exception.

Note

Quand l'option `levels` est une valeur vide dans la configuration du logger, n'importe quel niveau de message sera capturé.

Scopes de Journalisation

Souvent, vous voudrez configurer différents comportements de journalisation pour différents sous-systèmes ou parties de votre application. Prenez l'exemple d'un magasin e-commerce. Vous voudrez probablement gérer la journalisation pour les commandes et les paiements différemment des autres opérations de journalisation moins critiques.

CakePHP expose ce concept dans les scopes de journalisation. Quand les messages d'erreur sont écrits, vous pouvez inclure un nom scope. S'il y a un logger configuré pour ce scope, les messages de log seront dirigés vers ces loggers. Par exemple :

```
use Cake\Log\Engine\FileLog;

// Configure logs/magasins.log pour recevoir tous les types (niveaux de log),
// mais seulement ceux avec les scopes `commandes` et `paiements`
Log::setConfig('magasins', [
    'className' => FileLog::class,
    'path' => LOGS,
    'levels' => [],
    'scopes' => ['commandes', 'paiements'],
    'file' => 'magasins.log',
]);

// Configure logs/paiements.log pour recevoir tous les types, mais seulement
// ceux qui ont un scope `paiements`
Log::setConfig('paiements', [
    'className' => FileLog::class,
    'path' => LOGS,
    'levels' => [],
    'scopes' => ['paiements'],
    'file' => 'paiements.log',
]);

Log::warning('ceci sera écrit seulement dans magasins.log', ['scope' => ['commandes']]);
Log::warning('ceci sera écrit dans magasins.log et dans paiements.log', ['scope' => [
↪'paiements']]);
```

Les scopes peuvent aussi être passées dans une chaîne de texte ou un tableau indexé numériquement. Notez que si vous utilisez cette forme, cela limitera la possibilité de passer d'autres données de contexte :

```
Log::warning('Ceci est un avertissement', ['commandes']);
Log::warning('Ceci est un avertissement', 'paiements');
```

Note

Quand l'option `scopes` est un tableau vide ou `null` dans la configuration d'un logger, les messages de tous les `scopes` seront capturés. Définir l'option à `false` captura seulement les messages sans scope.

Utilisation de l'Adaptateur FileLog

Comme son nom l'indique FileLog écrit les messages log dans des fichiers. Le type des messages de log en cours d'écriture détermine le nom du fichier où le message sera stocké. Si le type n'est pas fourni, `LOG_ERR` est utilisé ce qui a pour effet d'écrire dans le log error. Le chemin par défaut est **logs/$level.log** :

```
// Exécuter ceci dans une classe CakePHP
$this->log("Quelque chose ne fonctionne pas!");

// Aboutit à ce que cela soit ajouté à logs/error.log
// 2007-11-02 10:22:02 Error: Quelque chose ne fonctionne pas!
```

Le répertoire configuré doit être accessible en écriture par le serveur web de l'utilisateur pour que la journalisation fonctionne correctement.

Vous pouvez configurer/changer la localisation de FileLog lors de la configuration du logger. FileLog accepte un `path` qui permet aux chemins personnalisés d'être utilisés :

```
Log::config('chemin_perso', [
    'className' => 'File',
    'path' => '/chemin/vers/endroit/perso/'
]);
```

Le moteur `FileLog` prend en charge les options suivantes :

- `size` Utilisé pour implémenter une rotation basique de fichiers. Si la taille du fichier de log atteint la taille spécifiée, le fichier existant est renommé en ajoutant à son nom un horodatage, et un nouveau fichier de log est créé. Cela peut être un nombre entier d'octets, ou des valeurs lisibles par l'homme telles que "10MB", "100KB" etc. Par défaut 10MB.
- `rotate` Les fichiers de log sont supprimés après un certain nombre de rotations, correspondant à la valeur spécifiée. Si la valeur est 0, les anciennes versions sont supprimées sans rotation. Par défaut 10.
- `mask` Définit les permissions pour les fichiers créés. S'il est vide, ce seront les permissions par défaut qui seront utilisées.

> **Note**
>
> En mode debug, les répertoires inexistants seront créés automatiquement afin d'éviter l'apparition d'erreurs superflues lors de l'utilisation de FileEngine.

Logging vers Syslog

Dans les environnements de production, il est fortement recommandé que vous configuriez votre système pour utiliser syslog plutôt que le logger de fichiers. Cela va fonctionner bien mieux parce que tout sera écrit de façon (presque) non bloquante et le logger de votre système d'exploitation peut être configuré séparément pour faire des rotations de fichier, pré-lancer les écritures ou utiliser un stockage complètement différent pour vos logs.

Utiliser syslog est à peu près comme utiliser le moteur par défaut FileLog, vous devez juste spécifier `Syslog` comme moteur à utiliser pour la journalisation. Le bout de configuration suivant va remplacer le logger par défaut avec syslog, ceci va être fait dans le fichier **config/bootstrap.php** :

```
Log::setConfig('default', [
    'engine' => 'Syslog'
]);
```

Le tableau de configuration accepté pour le moteur de journalisation Syslog comprend les clés suivantes :

— *format* : Un template de chaînes sprintf avec deux placeholders, le premier pour le type d'erreur, et le second pour le message lui-même. Cette clé est utile pour ajouter des informations supplémentaires à propos du serveur ou du processus dans le message de log. Par exemple : `%s - Web Server 1 - %s` va ressembler à `error - Web Server 1 - Une erreur s'est produite dans cette requête` après avoir remplacé les placeholders. Cette option est dépréciée. Utilisez *Formateurs de Logs* à la place.
— *prefix* : Une chaîne qui va préfixer tous les messages de log.
— *flag* : Un drapeau de type integer utilisé pour l'ouverture de la connexion au logger. La valeur par défaut est *LOG_ODELAY*. Regardez la documentation de `openlog` pour plus d'options.
— *facility* : Le slot de journalisation à utiliser dans syslog. Par défaut `LOG_USER` est utilisé. Regardez la documentation de `syslog` pour plus d'options.

Créer des Moteurs de Log

Les moteurs de log peuvent faire partie de votre application, ou faire partie d'un plugin. Supposons par exemple que vous ayez un enregistreur de logs sous forme de bases de données appelé `DatabaseLog`. S'il fait partie de votre application il serait placé dans **src/Log/Engine/DatabaseLog.php**. S'il fait partie d'un plugin il serait être placé dans **plugins/LoggingPack/src/Log/Engine/DatabaseLog.php**. Pour configurer un moteur de logs, vous devez utiliser *`Cake\Log\Log::setConfig()`*. Par example, la configuration de notre DatabaseLog pourrait ressembler à ceci :

```
// Pour src/Log
Log::setConfig('autreFichier', [
    'className' => 'Database',
    'model' => 'LogEntry',
    // ...
]);

// Pour un plugin appelé LoggingPack
Log::setConfig('autreFichier', [
    'className' => 'LoggingPack.Database',
    'model' => 'LogEntry',
    // ...
]);
```

Lorsque vous configurez un moteur de log le paramètre de `className` est utilisé pour localiser et charger le handler de log. Toutes les autres propriétés de configuration sont passées au constructeur du moteur de log sous forme de tableau :

```
namespace App\Log\Engine;
use Cake\Log\Engine\BaseLog;

class DatabaseLog extends BaseLog
{
    public function __construct($options = [])
    {
        parent::__construct($options);
        // ...
    }

    public function log($level, $message, array $context = [])
    {
        // Write to the database.
    }
}
```

CakePHP a besoin que tous les moteurs de log implémentent `Psr\Log\LoggerInterface`. La classe

`CakeLogEngineBaseLog` est un moyen simple de satisfaire l'interface puisqu'elle nécessite seulement que vous implémentiez la méthode `log()`.

Formateurs de Logs

Les formateurs de logs vous permettent de contrôler la façon dont sont formatés les messages de logs indépendamment du moteur de stockage. Chaque moteur de log fourni avec le cœur de CakePHP est accompagné d'un formateur configuré pour maintenir une compatibilité descendante. Cela étant, vous pouvez ajuster les formateurs pour les faire coller à vos besoins. Les formateur sont configurés en même temps que le moteur de log :

```
use Cake\Log\Engine\SyslogLog;
use App\Log\Formatter\CustomFormatter;

// Configuration simple de formatage sans autre option.
Log::setConfig('error', [
    'className' => SyslogLog::class,
    'formatter' => CustomFormatter::class,
]);

// Configurer un formateur avec des options supplémentaires.
Log::setConfig('error', [
    'className' => SyslogLog::class,
    'formatter' => [
        'className' => CustomFormatter::class,
        'key' => 'value',
    ],
]);
```

Pour implémenter votre propre formateur, vous aurez besoin d'étendre `Cake\Log\Format\AbstractFormatter` ou une de ses classes filles. La première méthode que vous aurez besoin d'implémenter est `format($level, $message, $context)`, qui est responsable du formatage des messages de log.

l'API de Log

class `Cake\Log\`**`Log`**

Une simple classe pour écrire dans les logs (journaux).

static `Cake\Log\Log::`**`setConfig`**(*$key*, *$config*)

Paramètres

- **$name** (`string`) – Nom du journal en cours de connexion, utilisé pour rejeter un journal plus tard.
- **$config** (`array`) – Tableau de configuration de l'information et des arguments du constructeur pour le journal.

Récupère ou définit la configuration pour un Journal. Regardez *Configuration des flux d'un log (journal)* pour plus d'informations.

static `Cake\Log\Log::`**`configured`**

Renvoie

Un tableau des journaux configurés.

Obtient les noms des journaux configurés.

static `Cake\Log\Log::`**`drop`**(*$name*)

Paramètres

— **$name** (string) – Nom du journal pour lequel vous ne voulez plus recevoir de messages.

static Cake\Log\Log::**write**(*$level*, *$message*, *$scope = []*)

Écrit un message dans tous les journaux configurés. `$level` indique le niveau de message log étant créé. `$message` est le message de l'entrée de log qui est en train d'être écrite. `$scope` est le scope(s) dans lequel un message de log est créé.

static Cake\Log\Log::**levels**

Appelez cette méthode sans arguments, ex : *Log : :levels()* pour obtenir le niveau de configuration actuel.

Méthodes pratiques

Les méthodes pratiques suivantes ont été ajoutées au journal `$message` avec le niveau de log approprié.

static Cake\Log\Log::**emergency**(*$message*, *$scope = []*)

static Cake\Log\Log::**alert**(*$message*, *$scope = []*)

static Cake\Log\Log::**critical**(*$message*, *$scope = []*)

static Cake\Log\Log::**error**(*$message*, *$scope = []*)

static Cake\Log\Log::**warning**(*$message*, *$scope = []*)

static Cake\Log\Log::**notice**(*$message*, *$scope = []*)

static Cake\Log\Log::**info**(*$message*, *$scope = []*)

static Cake\Log\Log::**debug**(*$message*, *$scope = []*)

Logging Trait

trait Cake\Log**LogTrait**

Un trait qui fournit des raccourcis pour les méthodes de journalisation

Cake\Log\LogTrait::**log**(*$msg*, *$level = LOG_ERR*)

Ecrit un message dans les logs. Par défaut, les messages sont écrits dans les messages ERROR.

Utiliser Monolog

Monolog est un logger populaire pour PHP. Puisqu'il intègre les mêmes interfaces que les loggers de CakePHP, vous pouvez l'utiliser dans votre application comme logger par défaut.

Après avoir installé Monolog en utilisant composer, configurez le logger en utilisant la méthode `Log::setConfig()` :

```
// config/bootstrap.php

use Monolog\Logger;
use Monolog\Handler\StreamHandler;

Log::setConfig('default', function () {
    $log = new Logger('app');
    $log->pushHandler(new StreamHandler('path/to/your/combined.log'));

    return $log;
```

(suite sur la page suivante)

(suite de la page précédente)

```
});

// Optionnellement, coupez les loggers par défaut devenus redondants
Log::drop('debug');
Log::drop('error');
```

Utilisez des méthodes similaires pour configurer un logger différent pour la console :

```
// config/bootstrap_cli.php

use Monolog\Logger;
use Monolog\Handler\StreamHandler;

Log::setConfig('default', function () {
    $log = new Logger('cli');
    $log->pushHandler(new StreamHandler('path/to/your/combined-cli.log'));

    return $log;
});

// Optionnellement, coupez les loggers par défaut devenus redondants
Configure::delete('Log.debug');
Configure::delete('Log.error');
```

Note

Lorsque vous utilisez un logger spécifique pour la console, assurez-vous de configurer conditionnellement le logger de votre application. Cela évitera la duplication des entrées de log.

CHAPITRE 23

Formulaires Sans Models

class Cake\Form**Form**

La plupart du temps, vous aurez des formulaires avec des *entities* et des *tables* de l'ORM en arrière-plan ou d'autres stockages persistants, mais il y a des fois où vous aurez besoin de valider un input de l'utilisateur et effectuer une action si les données sont valides. L'exemple le plus courant est un formulaire de contact.

Créer un Formulaire

Généralement lorsque vous utilisez la classe Form, vous voudrez utiliser une sous classe pour définir votre formulaire. Cela rend les tests plus faciles et vous permet de réutiliser votre formulaire. Les formulaires sont situés dans **src/Form** et ont habituellement `Form` comme suffixe de classe. Par exemple, un simple formulaire de contact ressemblerait à ceci :

```
// Dans src/Form/ContactForm.php
namespace App\Form;

use Cake\Form\Form;
use Cake\Form\Schema;
use Cake\Validation\Validator;

class ContactForm extends Form
{

    protected function _buildSchema(Schema $schema)
    {
        return $schema->addField('name', 'string')
            ->addField('email', ['type' => 'string'])
            ->addField('body', ['type' => 'text']);
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    protected function _buildValidator(Validator $validator)
    {
        return $validator->add('name', 'length', [
                'rule' => ['minLength', 10],
                'message' => 'Un nom est requis'
            ])->add('email', 'format', [
                'rule' => 'email',
                'message' => 'Une adresse email valide est requise',
            ]);
    }

    protected function _execute(array $data)
    {
        // Envoie un email.
        return true;
    }
}
```

Dans l'exemple ci-dessus nous pouvons voir les 3 méthodes de hook fournies par les formulaires :

- `_buildSchema` et utilisé pour définir le schema des données utilisé par FormHelper pour créer le formulaire HTML. Vous pouvez définir le type de champ, la longueur et la précision.
- `_buildValidator` Récupère une instance de *Cake\Validation\Validator* à laquelle vous pouvez attacher des validateurs.
- `_execute` vous permet de définir le comportement que vous souhaitez lorsque `execute()` est appelée et que les données sont valides.

Vous pouvez toujours également définir des méthodes publiques additionnelles si besoin.

Traiter les Données de Requêtes

Une fois que vous avez défini votre formulaire, vous pouvez l'utiliser dans votre controller pour traiter et valider les données de la requête :

```
// Dans un controller
namespace App\Controller;

use App\Controller\AppController;
use App\Form\ContactForm;

class ContactController extends AppController
{
    public function index()
    {
        $contact = new ContactForm();
        if ($this->request->is('post')) {
            if ($contact->execute($this->request->getData())) {
                $this->Flash->success('Nous reviendrons vers vous rapidement.');
            } else {
                $this->Flash->error('Il y a eu un problème lors de la soumission de␣
↪votre formulaire.');
            }
        }
        $this->set('contact', $contact);
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }
}
```

Dans l'exemple ci-dessus, si nous voulons utiliser un autre ensemble de validation que default, nous pouvons utiliser l'option `validate` :

```
if ($contact->execute($this->request->getData(), 'update')) {
    // Gestion du formulaire OK.
}
```

Cette option peut aussi être définie à `false` pour désactiver la validation.

Nous aurions aussi pu utiliser la méthode `validate()` pour valider uniquement les données de requête :

```
$isValid = $form->validate($this->request->getData());

// Vous pouvez aussi utiliser d'autres ensembles de validation. Ceci
// utiliserait les règles définies par `validationUpdate()`
$isValid = $form->validate($this->request->getData(), 'update');
```

Définir des Valeurs pour le Formulaire

Pour définir les valeurs d'un formulaire sans model, vous pouvez utiliser `$this->request->data()` comme dans tous formulaires créés par le FormHelper :

```
// Dans uncontroller
namespace App\Controller;

use App\Controller\AppController;
use App\Form\ContactForm;

class ContactController extends AppController
{
    public function index()
    {
        $contact = new ContactForm();
        if ($this->request->is('post')) {
            if ($contact->execute($this->request->getData())) {
                $this->Flash->success('Nous reviendrons vers vous rapidement.');
            } else {
                $this->Flash->error('Il y a eu un problème lors de la soumission de␣
↪votre formulaire.');
            }
        }

        if ($this->request->is('get')) {
            //Values from the User Model e.g.
            $this->request->data('name', 'John Doe');
            $this->request->data('email','john.doe@example.com');
        }

        $this->set('contact', $contact);
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }
}
```

Les valeurs ne doivent être définies que si la méthode de requête est GET, sinon vous allez surcharger les données POST qui auraient pu être incorrectes et non sauvegardées.

Récupérer les Erreurs d'un Formulaire

Une fois qu'un formulaire a été validé, vous pouvez récupérer les erreurs comme ceci :

```
$errors = $form->errors();
/* $errors contient
[
    'email' => ['Une adresse email valide est requise']
]
*/
```

Invalider un Champ de Formulaire depuis un Controller

Il est possible d'invalider un champ individuel depuis un controller sans utiliser la class Validator. Le scénario le plus courant est lorsque la validation est faite sur un serveur distant. Dans ce cas, vous devez invalider manuellement le champ suivant le retour du serveur distant :

```
// Dans src/Form/ContactForm.php
public function setErrors($errors)
{
    $this->_errors = $errors;
}
```

De la même façon que ce que la classe Validator aurait retourné l'erreur, `$errors` doit être sous ce format :

```
['fieldName' => ['validatorName' => 'The error message to display']]
```

Maintenant vous pourrez invalider des champs de formulaire en définissant le nom du champ suivi du message d'erreur :

```
// Dans un controller
$contact = new ContactForm();
$contact->setErrors(['email' => ['_required' => 'Your email is required']]);
```

Créez un formulaire HTML avec FormHelper pour voir le résultat.

Créer le HTML avec FormHelper

Une fois que vous avez créé une classe Form, vous voudrez probablement créer un formulaire HTML. FormHelper comprend les objets Form de la même manière que des entities de l'ORM :

```
echo $this->Form->create($contact);
echo $this->Form->control('name');
echo $this->Form->control('email');
echo $this->Form->control('body');
```

(suite sur la page suivante)

(suite de la page précédente)

```
echo $this->Form->button('Submit');
echo $this->Form->end();
```

Le code ci-dessus crée un formulaire HTML pour le `ContactForm` que nous avons défini précédemment. Les formulaires HTML créés avec FormHelper utiliseront les schema et validator définis pour déterminer les types de champ, leurs longueurs et les erreurs de validation.

CHAPITRE 24

Plugins

CakePHP vous permet de mettre en place une combinaison de controllers, models et vues et de les distribuer comme un plugin d'application pré-packagé que d'autres peuvent utiliser dans leurs applications CakePHP. Vous avez développé un module de gestion des utilisateurs sympa, un simple blog, ou un module de service web dans une de vos applications ? Pourquoi ne pas en faire un plugin CakePHP ? De cette manière, vous pourrez le réutiliser dans d'autres applications et le partager avec la communauté.

Un plugin CakePHP est séparé de l'application qui l'héberge et fournit généralement des fonctionnalités précises qui sont packagées de manière à être réutilisées très facilement dans d'autres applications. L'application et le plugin fonctionnent dans leurs espaces dédiés mais partage des propriétés spécifiques à l'application (comme les paramètres de connexion à la base de données par exemple) qui sont définies et partagées au travers de la configuration de l'application.

Chaque plugin est censé définir son namespace de top-niveau. Par exemple `DebugKit`. Par convention, les plugins utilisent leur nom de package pour leur namespace. Si vous souhaitez utiliser un namespace différent, vous pouvez configurer le namespace du plugin, quand les plugins sont chargés.

Installer un Plugin Avec Composer

Plusieurs plugins sont disponibles sur Packagist [156] et peuvent être installés avec `Composer`. Pour installer DebugKit, vous feriez ce qui suit :

```
php composer.phar require cakephp/debug_kit
```

Ceci installe la dernière version de DebugKit et met à jour vos fichiers **composer.json**, **composer.lock**, met à jour **vendor/cakephp-plugins.php** et met à jour votre autoloader.

156. https://packagist.org

Installer un Plugin Manuellement

Si le plugin que vous voulez installer n'est pas disponible sur packagist.org. Vous pouvez cloner ou copier le code du plugin dans votre répertoire **plugins**. Si vous voulez installer un plugin appelé `ContactManager`, vous créez un sous-répertoire nommé `ContactManager` dans **plugins**. C'est dans ce sous-répertoire que vous mettrez les répertoires src, tests, et tous les autres répertoires du plugin.

Autoload Manuel des Classes de Plugin

Si vous installez vos plugins par `composer` ou `bake`, vous ne devriez pas avoir besoin de configurer l'autoload de classes pour vos plugins.

Si vous installez manuellemen un plugin appelé par exemple `MyPlugin`, vous devrez modifier le fichier **composer.json** de votre application pour qu'il contienne les informations suivantes :

```
{
    "autoload": {
        "psr-4": {
            "MyPlugin\\": "plugins/MyPlugin/src/"
        }
    },
    "autoload-dev": {
        "psr-4": {
            "MyPlugin\\Test\\": "plugins/MyPlugin/tests/"
        }
    }
}
```

Si vous utilisez un namespace pour vos plugins, le mappage du namespace vers le chemin devra ressembler à :

```
{
    "autoload": {
        "psr-4": {
            "AcmeCorp\\Users\\": "plugins/AcmeCorp/Users/src/",
            "AcmeCorp\\Users\\Test\\": "plugins/AcmeCorp/Users/tests/"
        }
    }
}
```

De plus, vous devrez dire à Composer de rafraîchir son cache d'autoload :

```
php composer.phar dumpautoload
```

Charger un Plugin

Si vous voulez utiliser des routes du plugin, des commandes de console, des middlewares, des écouteurs d'événements, des templates ou des ressources du webroot, il faudra d'abord charger le plugin. C'est la function `bootstrap()` de votre application qui devra s'en charger :

```
// Dans src/Application.php
use Cake\Http\BaseApplication;
use ContactManager\ContactManagerPlugin;
```

(suite sur la page suivante)

(suite de la page précédente)

```
class Application extends BaseApplication {
    public function bootstrap()
    {
        parent::bootstrap();
        // Charge la plugin ContactManager d'après son nom
        $this->addPlugin(ContactManagerPlugin::class);

        // Charger un plugin avec un namespace d'après son "nom court"
        $this->addPlugin('AcmeCorp/ContactManager');

        // Charger ne dépendance de développement qui n'existera pas en environnement de␣
↪production.
        $this->addOptionalPlugin('AcmeCorp/ContactManager');
    }
}
```

Si vous voulez juste utiliser des helpers, behaviors ou components d'un plugin, vous n'avez pas besoin de le charger explicitement, bien que nous recommandions de toujours le faire.

Il existe aussi une commande de shell bien pratique pour activer un plugin. Exécutez cette instruction :

```
bin/cake plugin load ContactManager
```

Cela va mettre à jour la méthode bootstrap de votre application, ou insérer le code `$this->addPlugin('ContactManager');` dans le bootstrap à votre place.

Ajouté dans la version 4.1.0 : Ajout de la méthode `addOptionalPlugin()`.

Configuration du Plugin

Les plugins proposent plusieurs *hooks* permettant à un plugin de s'injecter lui-même aux endroits appropriés de votre application. Les *hooks* sont :

- `bootstrap` Utilisé pour charger les fichiers de configuration par défaut d'un plugin, définir des constantes et d'autres fonctions globales.
- `routes` Utilisé pour charger les routes pour un plugin. Il est déclenché après le chargement des routes de l'application.
- `middleware` Utilisé pour ajouter un middleware de plugin à la file de middlewares de l'application.
- `console` Utilisé pour ajouter des commandes de console à la collection des commandes d'une application.

En chargeant les plugins, vous pouvez configurer quels *hooks* doivent être activés. Par défaut, tous les *hooks* sont désactivés dans les plugins qui n'ont pas de *Plugin Classes*. Les plugins du nouveau style autorisent les auteurs de plugins à définir des valeurs par défaut, que vous pouvez configurer dans votre application :

```
// Dans Application::bootstrap()
use ContactManager\ContactManagerPlugin;

// Désactiver les routes pour le plugin ContactManager
$this->addPlugin(ContactManagerPlugin::class, ['routes' => false]);
```

Vous pouvez configurer les *hooks* avec un tableau d'options, ou par les méthodes fournies par les classes de plugin :

```
// Dans Application::bootstrap()
use ContactManager\ContactManagerPlugin;
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Utiliser disable/enable pour configurer les hooks.
$plugin = new ContactManagerPlugin();

$plugin->disable('bootstrap');
$plugin->enable('routes');
$this->addPlugin($plugin);
```

Les objets plugins connaissent aussi leurs noms et leurs informations de chemin :

```
$plugin = new ContactManagerPlugin();

// Obtenir le nom du plugin.
$name = $plugin->getName();

// Chemin vers la racine du plugin, et autres chemins.
$path = $plugin->getPath();
$path = $plugin->getConfigPath();
$path = $plugin->getClassPath();
```

Utiliser un Plugin

Vous pouvez référencer les controllers, models, components, behaviors et helpers du plugin en préfixant le nom du plugin.

Par exemple, Supposons que vous veuillez utiliser le ContactInfoHelper du plugin ContactManager pour afficher des informations de contact formatées dans une de vos vues. Dans votre controller, vous pouvez utiliser `addHelper()` de la façon suivante :

```
$this->viewBuilder()->addHelper('ContactManager.ContactInfo');
```

> **Note**
>
> Ce nom de classe séparé par un point se réfère à la *syntaxe de plugin*.

Vous serez ensuite capable d'accéder à `ContactInfoHelper` comme tout autre helper dans votre vue, comme ceci :

```
echo $this->ContactInfo->address($contact);
```

Le splugins peuvent utiliser des models, components, behaviors et helper fournis par l'application, ou par d'autres plugins si nécessaire :

```
// Utiliser un component d'application
$this->loadComponent('AppFlash');

// Utiliser un behavior d'un autre plugin
$this->addBehavior('AutrePlugin.AuditLog');
```

Créer Vos Propres Plugins

En exemple de travail, commençons par créer le plugin ContactManager référencé ci-dessus. Pour commencer, nous allons configurer votre structure de répertoire basique. Cela devrait ressembler à ceci :

```
/src
/plugins
    /ContactManager
        /config
        /src
            /ContactManagerPlugin.php
            /Controller
                /Component
            /Model
                /Table
                /Entity
                /Behavior
            /View
                /Helper
        /templates
            /layout
        /tests
            /TestCase
            /Fixture
        /webroot
```

Notez le nom du dossier du plugin, "**ContactManager**". Il est important que ce dossier ait le même nom que le plugin.

Dans le dossier plugin, vous remarquerez qu'il ressemble beaucoup à une application CakePHP, et c'est au fond ce que c'est. Vous n'avez à inclure aucun de vos dossiers si vous ne les utilisez pas. Certains plugins peuvent ne contenir qu'un Component ou un Behavior, et dans ce cas ils peuvent carrément ne pas avoir de répertoire "templates".

Un plugin peut aussi avoir n'importe quels autres répertoires similaires à ceux d'une application comme Config, Console, webroot, etc…

Créer un Plugin en utilisant Bake

Le processus de création des plugins peut être grandement simplifié en utilisant le shell bake.

Pour « cuisiner » (*bake*) un plugin, utilisez la commande suivante :

```
bin/cake bake plugin ContactManager
```

Vous pouvez utiliser bake pour créer des classes dans votre plugin. Par exemple, pour générer un contrôleur de plugin, vous pouvez lancer :

```
bin/cake bake controller --plugin ContactManager Contacts
```

Rendez-vous au chapitre /bake/usage si vous avez le moindre problème avec l'utilisation de la ligne de commande. Assurez-vous de re-générer votre autoloader après avoir créé votre plugin :

```
php composer.phar dumpautoload
```

Plugin Classes

Les Objets Plugin permettent à un auteur de plugin de spécifier une logique de démarrage, de définire des *hooks* par défaut, de charger des routes, un middleware ou des commandes de console. Les objets Plugin se trouvent dans **src/Plugin.php**. Pour notre plugin ContactManager, notre classe de plugin pourrait ressembler à :

```
namespace ContactManager;

use Cake\Core\BasePlugin;
use Cake\Core\ContainerInterface;
use Cake\Core\PluginApplicationInterface;
use Cake\Console\CommandCollection;
use Cake\Http\MiddlewareQueue;

class ContactManagerPlugin extends BasePlugin
{
    public function middleware(MiddlewareQueue $middleware): MiddlewareQueue
    {
        // Ajouter le middleware ici.
        $middleware = parent::middleware($middleware);

        return $middleware;
    }

    public function console(CommandCollection $commands): CommandCollection
    {
        // Ajouter les commandes de console ici.
        $commands = parent::console($commands);

        return $commands;
    }

    public function bootstrap(PluginApplicationInterface $app): void
    {
        // Ajouter des constantes, charger une configuration par défaut.
        // Par défaut, cela chargera `config/bootstrap.php` dans le plugin.
        parent::bootstrap($app);
    }

    public function routes($routes): void
    {
        // Ajouter des routes.
        // Par défaut, cela chargera `config/routes.php` dans le plugin.
        parent::routes($routes);
    }

    /**
     * Enregistrer des services de container d'application.
     *
     * @param \Cake\Core\ContainerInterface $container Le Container à mettre à jour.
     * @return void
     * @link https://book.cakephp.org/4/fr/development/dependency-injection.html
↪#dependency-injection
```

(suite sur la page suivante)

(suite de la page précédente)

```
     */
    public function services(ContainerInterface $container): void
    {
        // Ajoutez vos services ici
    }
}
```

Routes de Plugins

Les plugins peuvent mettre à disposition des fichiers de routes contenant leurs propres routes. Chaque plugin peut contenir un fichier **config/routes.php**. Ce fichier de routes peut être chargé quand le plugin est ajouté, ou dans le fichier de routes de l'application. Pour créer les routes du plugin ContractManager, ajoutez le code suivant dans **plugins/ContactManager/config/routes.php** :

```
<?php
use Cake\Routing\Route\DashedRoute;

$routes->plugin(
    'ContactManager',
    ['path' => '/contact-manager'],
    function ($routes) {
        $routes->setRouteClass(DashedRoute::class);

        $routes->get('/contacts', ['controller' => 'Contacts']);
        $routes->get('/contacts/{id}', ['controller' => 'Contacts', 'action' => 'view']);
        $routes->put('/contacts/{id}', ['controller' => 'Contacts', 'action' => 'update
↪']);
    }
);
```

Le code ci-dessus connectera les routes par défaut de votre plugin. Vous pourrez personnaliser ce fichier plus tard avec des routes plus spécifiques.

Avant de pouvoir accéder à vos controllers, assuez-vous que le plugin est bien chargé et que les routes du plugin le sont également. Dans votre fichier **src/Application.php**, ajoutez la ligne suivante :

```
$this->addPlugin('ContactManager', ['routes' => true]);
```

Vous pouvez également charger les routes du plugin dans la liste des routes de votre application. De cette manière, vous avez plus de contrôle sur le chargement des routes de plugin et cela vous permet d'englober les routes du plugin dans des préfixes et des "scopes" supplémentaires :

```
$routes->scope('/', function ($routes) {
    // Connect other routes.
    $routes->scope('/backend', function ($routes) {
        $routes->loadPlugin('ContactManager');
    });
});
```

Le code ci-dessus vous permettrait d'avoir des URLs de la forme `/backend/contact-manager/contacts`.

Controllers du Plugin

Les controllers pour notre plugin ContactManager seront stockés dans **plugins/ContactManager/src/Controller/**. Puisque notre activité principale est la gestion des contacts, nous aurons besoin d'un ContactsController pour ce plugin.

Ainsi, nous mettons notre nouveau ContactsController dans **plugins/ContactManager/src/Controller** et il ressemblerait à cela :

```
// plugins/ContactManager/src/Controller/ContactsController.php
namespace ContactManager\Controller;

use ContactManager\Controller\AppController;

class ContactsController extends AppController
{

    public function index()
    {
        //...
    }
}
```

Créez également le `AppController` si vous n'en avez pas déjà un :

```
// plugins/ContactManager/src/Controller/AppController.php
namespace ContactManager\Controller;

use App\Controller\AppController as BaseController;

class AppController extends BaseController
{
}
```

Un `AppController` dédié à votre plugin peut contenir la logique commune à tous les controllers de votre plugin, et n'est pas obligatoire si vous ne souhaitez pas en utiliser.

Si vous souhaitez accéder à ce que nous avons fait jusqu'ici, visitez l'URL `/contact-manager/contacts`. Vous aurez une erreur « Missing Model » parce que nous n'avons pas encore défini de model Contact.

Si votre application inclut le routage par défaut fourni par CakePHP, vous serez en mesure d'accéder aux controllers de votre plugin en utilisant des URLs comme :

```
// Accéder à la route index d'un controller de plugin.
/contact-manager/contacts

// Toute action sur un controller de plugin.
/contact-manager/contacts/view/1
```

Si votre application définit des préfixes de routage, le routage par défaut de CakePHP connectera aussi les routes qui utilisent le modèle suivant :

```
/{prefix}/{plugin}/{controller}
/{prefix}/{plugin}/{controller}/{action}
```

Consultez la section sur *Configuration du Plugin* pour plus d'informations sur la façon de charger les fichiers de routes spécifiques à un plugin.

Pour les plugins que vous n'avez pas créés avec bake, vous devrez aussi modifier le fichier `composer.json` pour ajouter votre plugin aux classes d'autoload. Vous pouvez le faire en suivant la documentation *Autoload Manuel des Classes de Plugin*.

Models du Plugin

Les Models pour le plugin sont stockés dans **plugins/ContactManager/src/Model**. Nous avons déjà défini un ContactsController pour ce plugin, donc créons la table et l'entity pour ce controller :

```
// plugins/ContactManager/src/Model/Entity/Contact.php:
namespace ContactManager\Model\Entity;

use Cake\ORM\Entity;

class Contact extends Entity
{
}

// plugins/ContactManager/src/Model/Table/ContactsTable.php:
namespace ContactManager\Model\Table;

use Cake\ORM\Table;

class ContactsTable extends Table
{
}
```

Si vous avez besoin de faire référence à un model dans votre plugin lors de la construction des associations ou la définition de classes d'entity, vous devrez inclure le nom du plugin avec le nom de la classe, séparés par un point. Par exemple :

```
// plugins/ContactManager/src/Model/Table/ContactsTable.php:
namespace ContactManager\Model\Table;

use Cake\ORM\Table;

class ContactsTable extends Table
{
    public function initialize(array $config): void
    {
        $this->hasMany('ContactManager.AltName');
    }
}
```

Si vous préférez que les clés du tableau pour l'association n'aient pas le préfixe du plugin, utilisez la syntaxe alternative :

```
// plugins/ContactManager/src/Model/Table/ContactsTable.php:
namespace ContactManager\Model\Table;

use Cake\ORM\Table;

class ContactsTable extends Table
{
```

(suite sur la page suivante)

(suite de la page précédente)

```
    public function initialize(array $config): void
    {
        $this->hasMany('AltName', [
            'className' => 'ContactManager.AltName',
        ]);
    }
}
```

Vous pouvez utiliser `Cake\ORM\Locator\LocatorAwareTrait` pour charger les tables de votre plugin en utilisant l'habituelle *syntaxe de plugin* :

```
// Les controllers utilisent déjà LocatorAwareTrait, donc vous n'avez pas besoin d
↪'ajouter ceci.
use Cake\ORM\Locator\LocatorAwareTrait;

$contacts = $this->fetchTable('ContactManager.Contacts');
```

Vues du Plugin

Les Vues se comportent exactement comme elles le font dans les applications normales. Placez-les juste dans le bon dossier à l'intérieur du dossier `plugins/[PluginName]/templates/`. Pour notre plugin ContactManager, nous aurons besoin d'une vue pour notre action `ContactsController::index()`, donc ajoutons-y ceci :

```
// plugins/ContactManager/templates/Contacts/index.php:
<h1>Contacts</h1>
<p>Ce qui suit est une liste triable de vos contacts</p>
<!-- Une liste triable de contacts irait ici....-->
```

Les Plugins peuvent fournir leurs propres layouts. Pour ajouter des layouts de plugin, placez vos fichiers de template dans `plugins/[PluginName]/templates/layout`. Pour utiliser le layout d'un plugin dans votre controller, vous pouvez faire comme ceci :

```
$this->viewBuilder()->setLayout('ContactManager.admin');
```

Si le préfix de plugin n'est pas précisé, le fichier de vue/layout sera localisé normalement.

> **Note**
>
> Pour des informations sur la façon d'utiliser les elements à partir d'un plugin, consultez *Elements*.

Redéfinir des Templates de Plugin depuis l'Intérieur de votre Application

Vous pouvez redéfinir toutes les vues du plugin à partir de l'intérieur de votre app en utilisant des chemins spéciaux. Si vous avez un plugin appelé "ContactManager", vous pouvez redéfinir les fichiers de template du plugin avec une logique de vue spécifique à l'application, en créant des fichiers sur le modèle de **templates/plugin/[Plugin]/[Controller]/[view].php**. Pour le controller Contacts, vous pourriez écrire le fichier suivant :

```
templates/plugin/ContactManager/Contacts/index.php
```

La création de ce fichier vous permettra de redéfinir **plugins/ContactManager/templates/Contacts/index.php**.

Si votre plugin fait partie d'une dépendence de Composer (ex : "LeVendor/LePlugin"), le chemin vers la vue "index" du controller Contacts sera :

```
templates/plugin/LeVendor/LePlugin/Custom/index.php
```

La création de ce fichier vous permettra de redéfinir **vendor/levendor/leplugin/templates/Custom/index.php**.

Si le plugin implémente un préfixe de routing, vous devez inclure ce préfixe dans le template réécrit par votre application. Par exemple, si le plugin "ContactManager" implémente un préfixe "Admin", le chemin du template réécrit sera :

```
templates/plugin/ContactManager/Admin/ContactManager/index.php
```

Ressources de Plugin

Les ressources web du plugin (mais pas les fichiers PHP) peuvent être servies à travers le répertoire `webroot` du plugin, exactement comme les ressources de l'application principale :

```
/plugins/ContactManager/webroot/
                                 css/
                                 js/
                                 img/
                                 flash/
                                 pdf/
```

Vous pouvez mettre n'importe quel type de fichier dans tout répertoire, exactement comme un webroot habituel.

> **Avertissement**
>
> La gestion des ressources statiques comme les fichiers images, Javascript et CSS à travers le Dispatcher est très inefficace. Consultez *Améliorer les Performances de votre Application* pour plus d'informations.

Liens vers les Ressources dans des Plugins

Vous pouvez utiliser la *syntaxe de plugin* pour faire un lien vers les ressources d'un plugin en utilisant les méthodes script, image ou css de `HtmlHelper` :

```
// Génère une URL de /contact_manager/css/styles.css
echo $this->Html->css('ContactManager.styles');

// Génère une URL de /contact_manager/js/widget.js
echo $this->Html->script('ContactManager.widget');

// Génère une URL de /contact_manager/img/logo.jpg
echo $this->Html->image('ContactManager.logo');
```

Les ressources de plugins sont servies par défaut en utilisant le midlleware `AssetMiddleware`. Ce n'est recommandé que pour le développement. En production vous devriez *symlinker vos assets* pour améliorer la performance.

Si vous n'utilisez pas les helpers, vous pouvez préfixer l'URL par /plugin-name/ pour servir une ressource du plugin . Un lien vers "/contact_manager/js/some_file.js" renverrait la ressource **plugins/ContactManager/webroot/js/some_file.js**.

Components, Helpers et Behaviors

Un plugin peut avoir des Components, Helpers et Behaviors tout comme une application CakePHP classique. Vous pouvez soit créer des plugins qui sont composés seulement de Components, Helpers ou Behaviors, ce qui peut être une bonne façon de construire des Components réutilisables qui peuvent être facilement déplacés dans n'importe quel projet.

On construit ces components est exactement de la même manière qu'à l'intérieur d'une application habituelle, sans aucune convention spéciale de nommage.

Pour faire référence à votre component, que ce soit depuis l'intérieur ou l'extérieur de votre plugin, vous devez seulement préfixer le nom du component par le nom du plugin. Par exemple :

```
// Component défini dans le plugin 'ContactManager'
namespace ContactManager\Controller\Component;

use Cake\Controller\Component;

class ExampleComponent extends Component
{
}

// dans vos controllers:
public function initialize(): void
{
    parent::initialize();
    $this->loadComponent('ContactManager.Example');
}
```

La même technique s'applique aux Helpers et aux Behaviors.

Commands

Les plugins peuvent enregistrer leurs commandes dans le *hook* `console()`. Par défaut, tous les shells et commandes du plugin sont découverts automatiquement et ajoutés à la liste des commandes de l'application. Les commandes de plugin sont préfixées par le nom du plugin. Par exemple, la commande `UserCommand` fournie par le plugin `ContactManager` serait enregistrée à la fois comme `contact_manager.user` et `user`. Le nom non préfixé sera retenu par un plugin seulement s'il n'est pas déjà utilisé par l'application ou un autre plugin.

Vous pouvez personnaliser les noms de commandes au moment de définir chaque commande dans votre plugin :

```
public function console($commands)
{
    // Créez des commandes imbriquées
    $commands->add('bake model', ModelCommand::class);
    $commands->add('bake controller', ControllerCommand::class);

    return $commands;
}
```

Tester votre Plugin

Si vous testez des controllers ou si vous générez des URLs, assurez-vous que votre plugin connecte les routes `tests/bootstrap.php`.

Pour plus d'informations, consultez la page *testing plugins*.

Publier votre Plugin

Les plugins CakePHP devraient être publiés dans le packagist [157]. De cette façon, d'autres personnes pourraient les utiliser comme dépendances composer. Vous pouvez aussi proposer votre plugin dans la liste des outils formidables pour CakePHP [158].

Choisissez un nom qui ait du sens pour votre nom de package. Idéalement, il faudrait le préfixer du nom de la dépendance, au cas présent « cakephp » comme le nom du framework. Le nom de vendor sera généralement votre nom d'utilisateur GitHub. **n'utilisez pas** le namespace de CakePHP (cakephp) car il est réservé aux plugins appartenant à CakePHP. Par convention, on utilise des lettres en minuscules et des traits d'union comme séparateurs.

Donc si vous avez créé un plugin « Logging » avec votre compte GitHub « FooBar », *foo-bar/cakephp-logging* serait un nom judicieux. Et respectivement, le plugin « Localized » appartenant à CakePHP peut se trouver sous *cakephp/localized*.

-.. index : : vendor/cakephp-plugins.php

Fichier de Mappage de Plugin

Quand vous installez des plugins par Composer, vous noterez la création de **vendor/cakephp-plugins.php**. Ce fichier de configuration contient un mappage des noms de plugins et de leurs chemins sur le système de fichiers. Cela rend possible l'installation des plugins dans le répertoire standard vendor, qui est en-dehors de l'arborescence de recherche normale. La classe `Plugin` utilisera ce fichier pour localiser les plugins lorsqu'ils sont chargés avec `addPlugin()`. Vous n'aurez généralement pas besoin d'éditer ce fichier manuellement, dans la mesure où Composer et le package `plugin-installer` s'en chargeront pour vous.

Gérer Vos Plugins avec Mixer

Mixer [159] est un autre moyen de découvrir et gérer les plugins dans votre application CakePHP. C'est un plugin CakePHP qui aide à installer des plugins depuis Packagist. Il vous aide aussi à gérer les plugins existants.

> **Note**
>
> IMPORTANT : Ne l'utilisez pas en environnement de production.

157. https://packagist.org
158. https://github.com/FriendsOfCake/awesome-cakephp
159. https://github.com/CakeDC/mixer

CHAPITRE 25

REST

Beaucoup de programmeurs néophytes d'application réalisent qu'ils ont besoin d'ouvrir leurs fonctionnalités principales à un public plus important. Fournir un accès sans entrave à votre API du cœur peut aider à ce que votre plateforme soit acceptée, et permettre les mashups et une intégration facile avec les autres systèmes.

CakePHP propose des méthodes pour exposer les actions de votre controller via des méthodes HTTP et pour sérialiser les variables de vue en fonction de la négociation du type de contenu. La négociation du type de contenu permet aux clients de votre application d'envoyer des requêtes avec des données sérialisées et de recevoir des réponses avec des données sérialisées via les en-têtes `Accept` et `Content-Type`, ou des extensions d'URL.

Mise en place Simple

Pour commencer à ajouter une API REST à votre application, nous aurons d'abord besoin d'un controller contenant les actions que nous voulons exposer en tant qu'API. Un controller de base pourrait ressembler à ceci :

```
// src/Controller/RecipesController.php
use Cake\View\JsonView;

class RecipesController extends AppController
{
    public function viewClasses(): array
    {
        return [JsonView::class];
    }

    public function index()
    {
        $recipes = $this->Recipes->find('all')->all();
        $this->set('recipes', $recipes);
        $this->viewBuilder()->setOption('serialize', ['recipes']);
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }

    public function view($id)
    {
        $recipe = $this->Recipes->get($id);
        $this->set('recipe', $recipe);
        $this->viewBuilder()->setOption('serialize', ['recipe']);
    }

    public function add()
    {
        $this->request->allowMethod(['post', 'put']);
        $recipe = $this->Recipes->newEntity($this->request->getData());
        if ($this->Recipes->save($recipe)) {
            $message = 'Saved';
        } else {
            $message = 'Error';
        }
        $this->set([
            'message' => $message,
            'recipe' => $recipe,
        ]);
        $this->viewBuilder()->setOption('serialize', ['recipe', 'message']);
    }

    public function edit($id)
    {
        $this->request->allowMethod(['patch', 'post', 'put']);
        $recipe = $this->Recipes->get($id);
        $recipe = $this->Recipes->patchEntity($recipe, $this->request->getData());
        if ($this->Recipes->save($recipe)) {
            $message = 'Saved';
        } else {
            $message = 'Error';
        }
        $this->set([
            'message' => $message,
            'recipe' => $recipe,
        ]);
        $this->viewBuilder()->setOption('serialize', ['recipe', 'message']);
    }

    public function delete($id)
    {
        $this->request->allowMethod(['delete']);
        $recipe = $this->Recipes->get($id);
        $message = 'Deleted';
        if (!$this->Recipes->delete($recipe)) {
            $message = 'Error';
        }
        $this->set('message', $message);
        $this->viewBuilder()->setOption('serialize', ['message']);
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }
}
```

Dans notre `RecipesController`, nous avons plusieurs actions qui définissent la logique pour créer, modifier, visualiser et supprimer des recettes. Dans chacune de nos actions, nous utilisons l'option `serialize` pour indiquer à CakePHP quelles variables de vue doivent être sérialisées lors de la création des réponses API. Nous connecterons notre controller aux URL de l'application avec le *Créer des Routes RESTful* :

```
// in config/routes.php
$routes->scope('/', function (RouteBuilder $routes): void {
    $routes->setExtensions(['json']);
    $routes->resources('Recipes');
});
```

Ces routes permettront aux URL comme `/recipes.json` de renvoyer une réponse encodée en JSON. Les clients pourront également faire une requête à `/recipes` avec l'en-tête `Content-Type: application/json`.

Encodage des données de réponse

Dans le controlleur ci-dessus, nous définissons une méthode `viewClasses()`. Cette méthode définit les vues dont votre controller dispose pour la négociation de contenu. Nous incluons `JsonView` de CakePHP qui permet des réponses basées sur JSON. Pour en savoir plus à ce sujet et sur les vues basées sur XML, consultez *Vues JSON et XML*. Ceci est utilisé par CakePHP pour sélectionner une classe de vue avec laquelle restituer une réponse REST.

Ensuite, nous disposons de plusieurs méthodes qui exposent la logique de base pour créer, modifier, afficher et supprimer des recettes. Dans chacune de nos actions, nous utilisons l'option `serialize` pour indiquer à CakePHP quelles variables de vue doivent être sérialisées lors des réponses API.

Si nous souhaitons modifier les données avant qu'elles ne soient converties en JSON, nous ne devons pas définir l'option de `serialize`, mais plutôt utiliser des fichiers modèles. Nous placerions les modèles REST pour notre RecipesController dans **templates/Recipes/json**.

Voir *Négociation de Contenu* pour plus d'informations sur la fonctionnalité de négociation de réponse de CakePHP.

Parsing des corps de requête

La création de la logique de l'action de modification nécessite une autre étape. Parce que nos ressources sont sérialisées au format JSON, il serait ergonomique si nos requêtes contenaient également la représentation JSON.

Dans notre classe `Application`, assurez-vous que les éléments suivants sont présents :

```
$middlewareQueue->add(new BodyParserMiddleware());
```

Ce middleware utilisera l'en-tête `content-type` pour détecter le format des données de requête et analyser les formats activés. Par défaut, seule l'analyse `JSON` est activée par défaut. Vous pouvez activer la prise en charge XML en activant l'option du constructeur XML. Lorsqu'une requête est effectuée avec un `Content-Type application/json`, CakePHP décodera les données de la requête et mettra à jour la requête afin que `$request->getData()` contienne le corps analysé.

Vous pouvez également câbler des désérialiseurs supplémentaires pour des formats alternatifs si vous en avez besoin, en utilisant `BodyParserMiddleware::addParser()`.

CHAPITRE 26

Sécurité

CakePHP fournit quelques outils pour sécuriser votre application. Les sections suivantes traitent de ces outils :

Security

class Cake\Utility**Security**

La librairie security [160] gère les mesures basiques de sécurité telles que les méthodes fournies pour le hashage et les données chiffrées.

Chiffrer et Déchiffrer les Données

static Cake\Utility\Security::**encrypt**(*$text*, *$key*, *$hmacSalt = null*)

static Cake\Utility\Security::**decrypt**(*$cipher*, *$key*, *$hmacSalt = null*)

Chiffre `$text` en utilisant AES-256. La `$key` devrait être une valeur avec beaucoup de différence dans les données un peu comme un bon mot de passe. Le résultat retourné sera la valeur chiffrée avec un checksum HMAC.

Cette méthode va soit utiliser openssl [161] soit mcrypt [162] selon ce qui est disponible sur votre système. Les données cryptées dans une implémentation sont portables vers les autres implémentations.

Avertissement

L'extension mcrypt [163] a été dépréciée dans PHP7.1

160. https://api.cakephp.org/4.x/class-Cake.Utility.Security.html
161. https://php.net/openssl
162. https://php.net/mcrypt
163. https://php.net/mcrypt

Cette méthode **ne** devrait **jamais** être utilisée pour stocker des mots de passe. A la place, vous devriez utiliser la manière de hasher les mots de passe fournie par *hash()*. Un exemple d'utilisation serait :

```
// En supposant que la clé est stockée quelque part, elle peut être
// réutilisée pour le déchiffrement plus tard.
$key = 'wt1U5MACWJFTXGenFoZoiLwQGrLgdbHA';
$result = Security::encrypt($value, $key);
```

Si vous ne fournissez pas de sel HMAC, la valeur `Security.salt` sera utilisée. Les valeurs chiffrées peuvent être déchiffrées avec *Cake\Utility\Security::decrypt()*.

Déchiffre une valeur chiffrée au préalable. Les paramètres `$key` et `$hmacSalt` doivent correspondre aux valeurs utilisées pour chiffrer ou alors le déchiffrement sera un échec. Un exemple d'utilisation serait :

```
// En supposant que la clé est stockée quelque part, elle peut être
// réutilisée pour le déchiffrement plus tard.
$key = 'wt1U5MACWJFTXGenFoZoiLwQGrLgdbHA';

$cipher = $user->secrets;
$result = Security::decrypt($cipher, $key);
```

Si la valeur ne peut pas être déchiffrée à cause de changements dans la clé ou le sel HMAC à `false` sera retournée.

Choisir une Implémentation de Crypto Spécifique

Si vous mettez à jour une application à partir de CakePHP 2.x, les données cryptées dans 2.x ne sont pas compatibles avec openssl. Cela est dû au fait que les données cryptées ne sont pas complètement compatibles avec AES. Si vous ne voulez pas gérer les problèmes de rechiffrage de vos données, vous pouvez forcer CakePHP à utiliser `mcrypt` en utilisant la méthode `engine()` :

```
// Dans config/bootstrap.php
use Cake\Utility\Crypto\Mcrypt;

Security::engine(new Mcrypt());
```

L'exemple ci-dessus vous permet de lire les données de façon transparente des versions précédentes de CakePHP, et de chiffrer les nouvelles données pour être compatible avec OpenSSL.

Hashage des Données

static `Cake\Utility\Security::`**hash**(*$string*, *$type = NULL*, *$salt = false*)

Crée un hash à partir d'une chaîne en utilisant la méthode donnée. Le Fallback sur la prochaine méthode disponible. Si `$salt` est défini à `true`, la valeur de salt de l'application sera utilisée :

```
// Utilise la valeur du salt de l'application
$sha1 = Security::hash('CakePHP Framework', 'sha1', true);

// Utilise une valeur du salt personnalisée
$sha1 = Security::hash('CakePHP Framework', 'sha1', 'my-salt');

// Utilise l'algorithme de hashage par défaut
$hash = Security::hash('CakePHP Framework');
```

La méthode `hash()` a aussi les stratégies de hashage suivantes :

— md5

— sha1
— sha256

Et tout autre algorithme de hashage que la fonction `hash()` de PHP permet.

> **Avertissement**
>
> Vous ne devriez pas utiliser `hash()` pour les mots de passe dans les nouvelles applications. A la place, vous devez utiliser la classe `DefaultPasswordHasher` qui utilise bcrypt par défaut.

Getting Secure Random Data

static `Cake\Utility\Security::randomBytes(`*$length*`)`

Get `$length` number of bytes from a secure random source. This function draws data from one of the following sources :

— PHP's `random_bytes` function.
— `openssl_random_pseudo_bytes` from the SSL extension.

If neither source is available a warning will be emitted and an unsafe value will be used for backwards compatibility reasons.

Protection CSRF

Les Cross-Site Request Forgeries (CSRF) sont un type de vulnérabilité dans lequel des commandes non autorisées sont exécutées au nom d'un utilisateur authentifié à son insu ou sans son consentement.

CakePHP offre deux formes de protection CSRF :

— `SessionCsrfProtectionMiddleware` stocke les jetons CSRF en session. Cela nécessite que votre application ouvre la session à chaque requête, avec des effets collatéraux. L'avantage des jetons CSRF basés sur la session est qu'ils sont propres à un utilisateur, et valides seulement pendant la durée de la session.
— `CsrfProtectionMiddleware` stocke les jetons CSRF dans un cookie. L'utilisation d'un cookie permet de faire les vérifications CSRF indépendamment de l'état du serveur. Les valeurs des cookies sont vérifiées par un test HMAC. Toutefois, de par leur nature *stateless*, les jetons CSRF sont réutilisables d'un utilisateur à l'autre et d'une session à l'autre.

> **Note**
>
> Vous ne pouvez pas utiliser ces deux approches simultanément, vous devez en choisir une. Si vous utilisez les deux ensemble, une erreur de jeton CSRF invalide se produira à chaque requête *PUT* et *POST*.

Middleware Cross Site Request Forgery (CSRF)

La protection CSRF peut être appliqué à votre application complète ou à des "scopes" spécifiques. En ajoutant le middleware CSRF à la file des middlewares de votre Application, vous protégez toutes les actions de l'application :

```
// dans src/Application.php
// Pour les jetons CSRF basés sur un Cookie.
use Cake\Http\Middleware\CsrfProtectionMiddleware;

// Pour les jetons CSRF basés sur la session.
use Cake\Http\Middleware\SessionCsrfProtectionMiddleware;

public function middleware(MiddlewareQueue $middlewareQueue): MiddlewareQueue
```

(suite sur la page suivante)

(suite de la page précédente)

```
{
    $options = [
        // ...
    ];
    $csrf = new CsrfProtectionMiddleware($options);
    // ou
    $csrf = new SessionCsrfProtectionMiddleware($options);

    $middlewareQueue->add($csrf);

    return $middlewareQueue;
}
```

En ajoutant la protection CSRF à des scopes de routing, vous pouvez conditionner l'utilisation de CSRF à certains groupes de routes :

```
// dans src/Application.php
use Cake\Http\Middleware\CsrfProtectionMiddleware;
use Cake\Routing\RouteBuilder;

public function routes(RouteBuilder $routes) : void
{
    $options = [
        // ...
    ];
    $routes->registerMiddleware('csrf', new CsrfProtectionMiddleware($options));
    parent::routes($routes);
}

// dans config/routes.php
$routes->scope('/', function (RouteBuilder $routes) {
    $routes->applyMiddleware('csrf');
});
```

Options du middleware CSRF basés sur un Cookie

Les options de configuration disponibles sont :

- `cookieName` Le nom du cookie à envoyer. Par défaut `csrfToken`.
- `expiry` La durée de vie du jeton CSRF. Par défaut, le temps de la session.
- `secure` Selon que le cookie doit être défini avec le drapeau Secure ou pas. C'est-à-dire que le cookie sera défini seulement dans une connexion HTTPS et toute tentative à travers un HTTP normal échouera. Par défaut à `false`.
- `httponly` Selon que le cookie sera défini avec le drapeau HttpOnly ou pas. Par défaut à `false`. Avant 4.1.0, utilisez l'option `httpOnly`.
- `samesite` Vous permet de déclarer si le cookie doit être restreint à un contexte first-party ou same-site. Les valeurs possibles sont `Lax`, `Strict` et `None`. Par défaut à `null`.
- `field` Le champ de formulaire à vérifier. Par défaut `_csrfToken`. Changer ceci obligera à changer également la configuration de FormHelper.

Options du middleware CSRF basé sur la Session

Les options de configuration disponibles sont :
- `key` La clé de session à utiliser. Par défaut *csrfToken*.
- `field` Le champ de formulaire à vérifier. Par défaut `_csrfToken`. Changer ceci obligera à changer également la configuration de FormHelper.

Lorsqu'il est activé, vous pouvez accéder au jeton CSRF en cours sur l'objet requête :

```
$token = $this->request->getAttribute('csrfToken');
```

Ignorer les vérifications CSRF pour certaines actions

Les deux implémentations du middleware CSRF vous autorisent à ignorer les callbacks de vérification pour un contrôle plus fin selon l'URL pour laquelle la vérification était censée avoir lieu :

```
// dans src/Application.php
use Cake\Http\Middleware\CsrfProtectionMiddleware;

public function middleware(MiddlewareQueue $middlewareQueue): MiddlewareQueue
{
    $csrf = new CsrfProtectionMiddleware();

    // La vérification du jeton sera ignorée lorsque le callback renvoie `true`.
    $csrf->skipCheckCallback(function ($request) {
        // Ignore la vérification du jeton pour les URLs API.
        if ($request->getParam('prefix') === 'Api') {
            return true;
        }
    });

    // S'assure que le middleware de routing est ajouté à la file avant le middleware de␣
↪protection CSRF.
    $middlewareQueue->add($csrf);

    return $middlewareQueue;
}
```

> **Note**
>
> Vous devez appliquer le middleware de protection CSRF seulement pour les routes qui gèrent des requêtes stateful en utilisant des cookies/sessions. Par exemple, en développant une API, les requêtes stateless ne sont pas affectées par CSRF, donc le middleware n'a pas besoin d'être appliqué à ces routes.

Intégration avec le FormHelper

Le `CsrfProtectionMiddleware` s'intègre parfaitement avec le `FormHelper`. Chaque fois que vous créez un formulaire avec le `FormHelper`, cela créera un champ caché contenant le token CSRF.

> **Note**
>
> Lorsque vous utilisez la protection CSRF, vous devriez toujours commencer vos formulaires avec le `FormHelper`. Si vous ne le faites pas, vous allez devoir créer manuellement les champs cachés dans chaque formulaire.

Protection CSRF et Requêtes AJAX

En plus des données de la requête, les tokens CSRF peuvent être soumis *via* le header spécial `X-CSRF-Token`. Utiliser un header facilite généralement l'intégration du token CSRF dans les applications qui utilisent Javascript de manière intensive ou avec les applications API JSON / XML.

Le token CSRF peut être récupéré via le Cookie `csrfToken`, ou en PHP *via* l'attribut nommé `csrfToken` dans l'objet requête. Il est peut-être plus facile d'utiliser le cookie si votre code Javascript se trouve dans des fichiers séparés des templates de vue de CakePHP, ou si vous avez déjà une fonctionnalité qui vous permet de parser des cookies avec Javascript.

Si vous avez des fichiers Javascript séparés mais que vous ne voulez pas avoir à gérer des cookies, vous pouvez par exemple définir un token dans une variable Javascript globale dans votre layout, en définissant un bloc script comme ceci :

```
echo $this->Html->scriptBlock(sprintf(
    'var csrfToken = %s;',
    json_encode($this->request->getAttribute('csrfToken'))
));
```

Vous pouvez accéder au token par l'expression `csrfToken` ou `window.csrfToken` dans n'importe quel fichier de script qui sera chargé après ce bloc de script.

Une autre alternative serait de placer le token dans une balise meta personnalisée comme ceci :

```
echo $this->Html->meta('csrfToken', $this->request->getAttribute('csrfToken'));
```

ce qui le rendrait accessible dans vos scripts en recherchant l'élément `meta` nommé `csrfToken`. Avec jQuery, cela pourrait être aussi simple que ça :

```
var csrfToken = $('meta[name="csrfToken"]').attr('content');
```

Middleware Content Security Policy

Le `CspMiddleware` rend les choses plus simples pour ajouter des en-têtes Content-Security-Policy dans votre application. Avant de l'utiliser, vous devez installer `paragonie/csp-builder` :

```
composer require paragonie/csp-builder
```

Vous pouvez configurer le middleware en utilisant un tableau, ou en lui passant un objet `CSPBuilder` déjà construit :

```
use Cake\Http\Middleware\CspMiddleware;

$csp = new CspMiddleware([
    'script-src' => [
        'allow' => [
            'https://www.google-analytics.com',
        ],
        'self' => true,
        'unsafe-inline' => false,
        'unsafe-eval' => false,
    ],
]);

$middlewareQueue->add($csp);
```

Si vous voulez utiliser une configuration CSP plus stricte, vous pouvez activer des règles CSP basées sur le nonce avec les options `scriptNonce` et `styleNonce`. Lorsqu'elles sont activées, ces options vont modifier votre politique CSP et définir les attributs `cspScriptNonce` et `cspStyleNonce` dans la requête. Ces attributs sont appliqués à l'attribut `nonce` de tous les éléments scripts et liens CSS créés par `HtmlHelper`. Cela simplifie l'adoption de stratégies utilisant un nonce-base64 [164] et `strict-dynamic` pour un surcroît de sécurité et une maintenance plus facile :

```
$policy = [
    // Doivent exister, même vides, pour définir le nonce pour script-src
    'script-src' => [],
    'style-src' => [],
];
// Active l'ajout automatique du nonce aux tags script & liens CSS.
$csp = new CspMiddleware($policy, [
    'scriptNonce' => true,
    'styleNonce' => true,
]);
$middlewareQueue->add($csp);
```

Middleware des Headers de Sécurité

La couche `SecurityHeaderMiddleware` vous permet d'ajouter à votre application des headers liés à la sécurité. Une fois configuré, le middleware peut ajouter les headers suivants aux réponses :

- `X-Content-Type-Options`
- `X-Download-Options`
- `X-Frame-Options`
- `X-Permitted-Cross-Domain-Policies`
- `Referrer-Policy`

Ce middleware peut être configuré en utilisant l'interface fluide avant d'être appliqué au stack de middlewares :

```
use Cake\Http\Middleware\SecurityHeadersMiddleware;

$securityHeaders = new SecurityHeadersMiddleware();
$securityHeaders
    ->setCrossDomainPolicy()
    ->setReferrerPolicy()
    ->setXFrameOptions()
    ->setXssProtection()
    ->noOpen()
    ->noSniff();

$middlewareQueue->add($securityHeaders);
```

Middleware HTTPS Enforcer

Si vous voulez que votre application soit accessible uniquement par des connexions HTTPS, vous pouvez utiliser le `HttpsEnforcerMiddleware` :

```
use Cake\Http\Middleware\HttpsEnforcerMiddleware;

// Toujours soulever une exception et ne jamais rediriger.
```

(suite sur la page suivante)

164. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/script-src

(suite de la page précédente)

```
$https = new HttpsEnforcerMiddleware([
    'redirect' => false,
]);

// Envoyer un code de statut 302 en cas de redirection
$https = new HttpsEnforcerMiddleware([
    'redirect' => true,
    'statusCode' => 302,
]);

// Envoyer des headers supplémentaires dans la réponse de redirection.
$https = new HttpsEnforcerMiddleware([
    'headers' => ['X-Https-Upgrade' => 1],
]);

// Désactiver le HTTPs forcé quand ``debug`` est activé.
$https = new HttpsEnforcerMiddleware([
    'disableOnDebug' => true,
]);
```

À la réception d'une requête non-HTTP qui n'utilise pas GET, un `BadRequestException` sera soulevée.

Ajouter Strict-Transport-Security

Si votre application nécessite du SSL, une bonne idée serait de définir le header `Strict-Transport-Security`. La valeur de ce header est mise en cache par le navigateur, et informe les navigateurs qu'ils devraient toujours se connecter en HTTPS. Vous pouvez configurer ce header avec l'option `hsts` :

```
   $https = new HttpsEnforcerMiddleware([
       'hsts' => [
           // La durée pendant laquelle la valeur du header devrait être mise en cache
           'maxAge' => 60 * 60 * 24 * 365,
           // cette politique s'applique-t-elle aux sous-domaines ?
           'includeSubdomains' => true,
           // La valeur de ce header devrait-elle être mise en cache dans le service
           // HSTS preload de Google ? Bien que ne faisant pas partie de la␣
↪spécification, il est souvent implémenté.
           'preload' => true,
       ],
   ]);

.. meta::
    :title lang=fr: Middleware HTTPS Enforcer
```

CHAPITRE 27

Sessions

CakePHP fournit des fonctionnalités et un ensemble d'outils qui s'ajoutent à l'extension native `session` de PHP. Les Sessions vous permettent d'identifier les utilisateurs uniques pendant leurs requêtes et de stocker les données persistantes pour les utilisateurs spécifiques. Au contraire des Cookies, les données de session ne sont pas disponibles du côté client. L'utilisation de `$_SESSION` est généralement à éviter dans CakePHP, et à la place l'utilisation des classes de Session est préférable.

Configuration de Session

La configuration de Session est généralement définie dans `/config/app.php`. Les options suivantes sont disponibles :

- `Session.timeout` - Le nombre de *minutes* avant que le gestionnaire de session de CakePHP ne fasse expirer la session.
- `Session.defaults` - Vous permet d'utiliser les configurations de session intégrées par défaut comme une base pour votre configuration de session. Regardez ci-dessous les paramètres intégrés par défaut
- `Session.handler` - Vous permet de définir un gestionnaire de session personnalisé. La base de données du cœur et les gestionnaires de cache de session utilisent celui-ci. Regardez ci-dessous pour des informations supplémentaires sur les gestionnaires de Session.
- `Session.ini` - Vous permet de définir les configurations ini de session supplémentaire pour votre config. Ceci combiné avec `Session.handler` remplace les fonctionnalités de gestionnaire de session personnalisé des versions précédentes.
- `Session.cookie` - Le nom du cookie à utiliser. Le défaut est "CAKEPHP".
- `Session.cookiePath` - Le chemin URL pour lequel le cookie de session est défini. Correspond à la configuration `session.cookie_path` du php.ini. Le défaut est le chemin de base de l'application.

CakePHP met par défaut la configuration de `session.cookie_secure` à `true`, quand votre application est sur un protocole SSL. Si votre application utilise à la fois les SSL et non-SSL, alors vous aurez peut-être des problèmes avec de sessions perdues. Si vous avez besoin d'accéder à la session sur les deux domaines SSL et non-SSL, vous devrez désactiver cela :

```
Configure::write('Session', [
    'defaults' => 'php',
```

(suite sur la page suivante)

(suite de la page précédente)

```
    'ini' => [
        'session.cookie_secure' => false
    ]
]);
```

Le chemin du cookie de session est par défaut le chemin de base de l'application. Pour changer ceci, vous pouvez utiliser la valeur ini `session.cookie_path`. Par exemple, si vous voulez que votre session soit sauvegardée pour tous les sous-domaines, vous pouvez faire :

```
Configure::write('Session', [
    'defaults' => 'php',
    'ini' => [
        'session.cookie_path' => '/',
        'session.cookie_domain' => '.yourdomain.com'
    ]
]);
```

Par défaut PHP définit le cookie de session pour qu'il expire dès que le navigateur est fermé, quelque soit la valeur `Session.timeout` configurée. Le timeout du cookie est contrôlé par la valeur ini `session.cookie_lifetime` et peut être configuré en utilisant :

```
Configure::write('Session', [
    'defaults' => 'php',
    'ini' => [
        // Rend le cookie non valide après 30 minutes s'il n'y
        // a aucune visite d'aucune page sur le site.
        'session.cookie_lifetime' => 1800
    ]
]);
```

La différence entre les valeurs `Session.timeout` et `session.cookie_lifetime` est que la deuxième repose sur le fait que le client dit la vérité sur le cookie. Si vous devez vérifier plus strictement le timeout, sans que cela ne repose sur ce que dit le client, vous devez utiliser `Session.timeout`.

Merci de noter que `Session.timeout` correspond au temps total d'inactivité d'un utilisateur (par ex, le temps sans visite d'aucune page où la session est utilisée), et ne limite pas le nombre total de minutes pendant lesquelles un utilisateur peut rester sur le site.

Gestionnaires de Session intégrés & Configuration

CakePHP est fourni avec plusieurs configurations de session intégrées. Vous pouvez soit utiliser celles-ci comme base pour votre configuration de session, soit créer une solution complètement personnalisée. Pour utiliser les valeurs par défaut, définissez simplement la clé "defaults" avec le nom par défaut que vous voulez utiliser. Vous pouvez ensuite surcharger toute sous-configuration en la déclarant dans votre config Session :

```
Configure::write('Session', [
    'defaults' => 'php'
]);
```

Le code ci-dessus va utiliser la configuration de session intégrée dans "php". Vous pourriez augmenter tout ou partie de celle-ci en faisant ce qui suit :

```
Configure::write('Session', [
    'defaults' => 'php',
    'cookie' => 'my_app',
    'timeout' => 4320 // 3 days
]);
```

Le code ci-dessus surcharge le timeout et le nom du cookie pour la configuration de session "php". Les configurations intégrées sont :

- `php` - Sauvegarde les sessions avec les configurations standard dans votre fichier php.ini.
- `cake` - Sauvegarde les sessions en tant que fichiers à l'intérieur de `tmp/sessions`. Ceci est une bonne option lorsque les hôtes ne vous autorisent pas à écrire en dehors de votre propre répertoire home.
- `database` - Utilise les sessions de base de données intégrées. Regardez ci-dessous pour plus d'informations.
- `cache` - Utilise les sessions de cache intégrées. Regardez ci-dessous pour plus d'informations.

Gestionnaires de Session

Les gestionnaires peuvent aussi être définis dans le tableau de config de session. En définissant la clé de config "handler.engine", vous pouvez nommer le nom de la classe, ou fournir une instance de gestionnaire. La classe/objet doit implémenter le `SessionHandlerInterface` natif de PHP. Implémenter cette interface va permettre de faire le lien automatiquement de `Session` vers les méthodes du gestionnaire. Le Cache du cœur et les gestionnaires de session de la Base de Données utilisent tous les deux cette méthode pour sauvegarder les sessions. De plus, les configurations pour le gestionnaire doivent être placées dans le tableau du gestionnaire. Vous pouvez ensuite lire ces valeurs à partir de votre gestionnaire :

```
'Session' => [
    'handler' => [
        'engine' => 'DatabaseSession',
        'model' => 'CustomSessions'
    ]
]
```

Le code ci-dessus montre comment vous pouvez configurer le gestionnaire de session de la Base de Données avec un model de l'application. Lors de l'utilisation de noms de classe comme handler.engine, CakePHP va s'attendre à trouver votre classe dans le namespace `Network\Session`. Par exemple, si vous aviez une classe `AppSessionHandler`, le fichier doit être **src/Network/Session/AppSessionHandler.php**, et le nom de classe doit être `App\Network\Session\AppSessionHandler`. Vous pouvez aussi utiliser les gestionnaires de session à partir des plugins. En configurant le moteur avec `MyPlugin.PluginSessionHandler`.

Les Sessions de la Base de Données

Si vous devez utiliser une base de données pour stocker vos données de session, configurez comme suit :

```
'Session' => [
    'defaults' => 'database'
]
```

Cette configuration nécessitera une table ayant ce schema :

```
CREATE TABLE `sessions` (
  `id` char(40) CHARACTER SET ascii COLLATE ascii_bin NOT NULL,
  `created` datetime DEFAULT CURRENT_TIMESTAMP, -- Optional
  `modified` datetime DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP, -- Optional
  `data` blob DEFAULT NULL, -- for PostgreSQL use bytea instead of blob
  `expires` int(10) unsigned DEFAULT NULL,
```

(suite sur la page suivante)

(suite de la page précédente)

```
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Vous pouvez trouver une copie du schéma pour la table de sessions dans le squelette d'application.

Vous pouvez utiliser votre propre classe `Table` pour gérer la sauvegarde des sessions :

```
'Session' => [
    'defaults' => 'database',
    'handler' => [
        'engine' => 'DatabaseSession',
        'model' => 'CustomSessions'
    ]
]
```

Le code ci-dessus va dire à Session d'utiliser la base de donnée "database" intégrée par défaut, et spécifier qu'un model appelé `CustomSession` sera celui délégué pour la sauvegarde d'information de session dans la base de données.

Les Sessions de Cache

La classe Cache peut aussi être utilisée pour stocker les sessions. Cela vous permet de stocker les sessions dans un cache comme APC, Memcached, ou XCache. Il y a quelques bémols dans l'utilisation des sessions en cache, puisque si vous avez épuisé l'espace de cache, les sessions vont commencer à expirer tandis que les enregistrements sont supprimés.

Pour utiliser les sessions basées sur le Cache, vous pouvez configurer votre config Session comme ceci :

```
Configure::write('Session', [
    'defaults' => 'cache',
    'handler' => [
        'config' => 'session'
    ]
]);
```

Cela va configurer Session pour utiliser la classe `CacheSession` déléguée pour sauvegarder les sessions. Vous pouvez utiliser la clé "config" qui va mettre en cache la configuration à utiliser. La configuration par défaut de la mise en cache est `'default'`.

Configurer les Directives ini

Celui intégré par défaut tente de fournir une base commune pour la configuration de session. Vous aurez aussi besoin d'ajuster les flags ini spécifiques. CakePHP permet de personnaliser les configurations ini pour les deux configurations par défaut, ainsi que celles personnalisées. La clé `ini` dans les configurations de session vous permet de spécifier les valeurs de configuration individuelles. Par exemple vous pouvez l'utiliser pour contrôler les configurations comme `session.gc_divisor` :

```
Configure::write('Session', [
    'defaults' => 'php',
    'ini' => [
        'session.cookie_name' => 'MyCookie',
        'session.cookie_lifetime' => 1800, // Valide pour 30 minutes
        'session.gc_divisor' => 1000,
        'session.cookie_httponly' => true
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ]
]);
```

Créer un Gestionnaire de Session Personnalisé

Créer un gestionnaire de session personnalisé est simple dans CakePHP. Dans cet exemple, nous allons créer un gestionnaire de session qui stocke les sessions à la fois dans le Cache (APC) et la base de données. Cela nous donne le meilleur du IO rapide de APC, sans avoir à se soucier des sessions disparaissent quand le cache se remplit.

D'abord, nous aurons besoin de créer notre classe personnalisée et de la mettre dans **src/Network/Session/ComboSession.php**. La classe devrait ressembler à :

```
namespace App\Network\Session;

use Cake\Cache\Cache;
use Cake\Core\Configure;
use Cake\Network\Session\DatabaseSession;

class ComboSession extends DatabaseSession
{
    public $cacheKey;

    public function __construct()
    {
        $this->cacheKey = Configure::read('Session.handler.cache');
        parent::__construct();
    }

    // Lire des données de session.
    public function read($id)
    {
        $result = Cache::read($id, $this->cacheKey);
        if ($result) {
            return $result;
        }

        return parent::read($id);
    }

    // Ecrire des données dans session
    public function write($id, $data)
    {
        Cache::write($id, $data, $this->cacheKey);

        return parent::write($id, $data);
    }

    // Détruire une session.
    public function destroy($id)
    {
        Cache::delete($id, $this->cacheKey);
```

(suite sur la page suivante)

(suite de la page précédente)

```
        return parent::destroy($id);
    }

    // Retire des sessions expirées.
    public function gc($expires = null)
    {
        return Cache::gc($this->cacheKey) && parent::gc($expires);
    }
}
```

Notre classe étend la classe intégrée `DatabaseSession` donc nous ne devons pas dupliquer toute sa logique et son comportement. Nous entourons chaque opération avec une opération *Cake\Cache\Cache*. Cela nous permet de récupérer les sessions de la mise en cache rapide, et nous évite de nous inquiéter sur ce qui arrive quand nous remplissons le cache. Dans votre **config/app.php**, écrivez un block de session comme ceci :

```
'Session' => [
    'defaults' => 'database',
    'handler' => [
        'engine' => 'ComboSession',
        'model' => 'Session',
        'cache' => 'apc'
    ]
],
// Assurez-vous d'ajouter une config de cache apc
'Cache' => [
    'apc' => ['engine' => 'Apc']
]
```

Maintenant notre application va se lancer en utilisant notre gestionnaire de session personnalisé pour la lecture et l'écriture des données de session.

class Session

Accéder à l'Objet Session

Vous pouvez accéder aux données session à tous les endroits où vous avez accès à l'objet request. Cela signifie que la session est accessible via :

- Controllers
- Views
- Helpers
- Cells
- Components

En plus de l'objet basique session, vous pouvez aussi utiliser `Cake\View\Helper\SessionHelper` pour interagir avec la session dans vos views. Un exemple simple de l'utilisation de session serait :

```
$name = $this->request->session()->read('User.name');

// Si vous accédez à la session plusieurs fois,
// vous voudrez probablement une variable locale.
$session = $this->request->getSession();
$name = $session->read('User.name');
```

Dans les helpers, utilisez `$this->getView()->getRequest()` pour obtenir l'objet request. Dans les components, utilisez `$this->getController()->getRequest()`.

Lire & Ecrire les Données de Session

static `Session::read(`*$key*`)`

Vous pouvez lire les valeurs de session en utilisant la syntaxe compatible `Hash::extract()` :

```
$session->read('Config.language');
```

static `Session::write(`*$key*, *$value*`)`

`$key` devrait être le chemin séparé de point et `$value` sa valeur :

```
$session->write('Config.language', 'en');
```

Vous pouvez également spécifier un ou plusieurs hash de la manière suivante :

```
$session->write([
  'Config.theme' => 'blue',
  'Config.language' => 'en',
]);
```

static `Session::delete(`*$key*`)`

Quand vous avez besoin de supprimer des données de la session, vous pouvez utiliser `delete()` :

```
$session->delete('Some.value');
```

static `Session::consume(`*$key*`)`

Quand vous avez besoin de lire et supprimer des données de la session, vous pouvez utiliser `consume()` :

```
$session->consume('Some.value');
```

`Session::check(`*$key*`)`

Si vous souhaitez voir si des données existent dans la session, vous pouvez utiliser `check()` :

```
if ($session->check('Config.language')) {
    // Config.language existe et n'est pas null.
}
```

Détruire la Session

`Session::destroy()`

Détruire la session est utile quand les utilisateurs se déconnectent. Pour détruire une session, utilisez la méthode `destroy()` :

```
$session->destroy();
```

Détruire une session va retirer toutes les données sur le serveur dans la session, mais **ne va pas** retirer le cookie de session.

Faire une Rotation des Identificateurs de Session

`Session::renew()`

Alors que `AuthComponent` réactualise automatiquement l'id de session quand les utilisateurs se connectent et se déconnectent, vous aurez peut-être besoin de faire une rotation de l'id de session manuellement. Pour ce faire, utilisez la méthode `renew()` :

```
$session->renew();
```

Messages Flash

Les messages flash sont des messages courts à afficher aux utilisateurs une seule fois. Ils sont souvent utilisés pour afficher des messages d'erreur ou pour confirmer que les actions se font avec succès.

Pour définir et afficher les messages flash, vous devez utiliser *Flash* et *Flash*

CHAPITRE 28

Testing

CakePHP fournit un support de test intégré complet. CakePHP permet l'intégration de PHPUnit [165]. En plus de toutes les fonctionnalités offertes par PHPUnit, CakePHP offre quelques fonctionnalités supplémentaires pour faciliter le test. Cette section va couvrir l'installation de PHPUnit, comment démarrer avec les Tests Unitaires, et comment utiliser les extensions offertes par CakePHP.

Installer PHPUnit

CakePHP utilise PHPUnit comme framework de test sous-jacent. PHPUnit est *de facto* le standard des tests unitaires en PHP. Il offre un ensemble de fonctionnalités profondes et puissantes pour s'assurer que votre code fait ce que vous pensez qu'il fait. PHPUnit peut être installé soit avec le package PHAR [166], soit avec Composer [167].

Installer PHPUnit avec Composer

Pour installer PHPUnit avec Composer :

```
$ php composer.phar require --dev phpunit/phpunit:"^8.5"
```

Cela va ajouter la dépendance à la section `require-dev` de votre `composer.json`, et ensuite installer PHPUnit au même endroit que vos autres dépendances.

Vous pouvez maintenant lancer PHPUnit en utilisant :

```
$ vendor/bin/phpunit
```

165. https://phpunit.de
166. https://phpunit.de/#download
167. https://getcomposer.org

Utiliser le fichier PHAR

Après avoir téléchargé le fichier **phpunit.phar**, vous pouvez l'utiliser pour lancer vos tests :

```
php phpunit.phar
```

Astuce

Par souci de commodité vous pouvez rendre phpunit.phar disponible globalement sur Unix ou Linux via les commandes suivantes :

```
chmod +x phpunit.phar
sudo mv phpunit.phar /usr/local/bin/phpunit
phpunit --version
```

Référez vous à la documentation de PHPUnit pour les instructions concernant l'installation globale du PHAR PHPUnit sur Windows [168].

Tester la Configuration de la Base de Données

Pensez à activer le débogage dans votre fichier **config/app_local.php** avant de lancer des tests. Vous devrez aussi vous assurer d'avoir ajouté une configuration de base de données `test` dans **config/app_local.php**. Cette configuration est utilisée par CakePHP pour les tables de fixture et les données :

```
'Datasources' => [
    'test' => [
        'datasource' => 'Cake\Database\Driver\Mysql',
        'persistent' => false,
        'host' => 'dbhost',
        'username' => 'dblogin',
        'password' => 'dbpassword',
        'database' => 'test_database'
    ],
],
```

Note

C'est une bonne idée de faire une base de données de test différente de votre base de données actuelle. Cela évitera les erreurs embarrassantes qui ne manqueront pas d'arriver.

Vérifier la Configuration Test

Après avoir installé PHPUnit et configuré la configuration de la base de données de `test`, vous pouvez vous assurer que vous êtes prêt à écrire et lancer vos propres tests en lançant un de ceux présents dans le cœur :

```
# Pour phpunit.phar
$ php phpunit.phar

# Pour un PHPUnit installé avec Composer
$ vendor/bin/phpunit
```

168. https://phpunit.de/manual/current/en/installation.html#installation.phar.windows

Ceci va lancer tous les tests que vous avez, ou vous indiquer qu'aucun test n'a été lancé. Pour lancer un test spécifique, vous pouvez fournir le chemin du test en paramètre à PHPUnit. Par exemple, si vous aviez un cas de test pour la classe ArticlesTable, vous pourriez le lancer avec :

```
$ vendor/bin/phpunit tests/TestCase/Model/Table/ArticlesTableTest
```

Vous devriez voir une barre verte avec quelques informations supplémentaires sur les tests exécutés et le nombre de tests réussis.

> **Note**
>
> Si vous êtes sur un système Windows, vous ne verrez probablement pas les couleurs.

Conventions des Cas de Test (TestCase)

Comme beaucoup de choses dans CakePHP, les cas de test ont quelques conventions. En ce qui concerne les tests :

1. Les fichiers PHP contenant les tests doivent être dans votre répertoire `tests/TestCase/[Type]`.
2. Les noms de ces fichiers doivent finir par **Test.php** et pas seulement **.php**.
3. Les classes contenant les tests doivent étendre `Cake\TestSuite\TestCase`, `Cake\TestSuite\IntegrationTestCase` ou `\PHPUnit\Framework\TestCase`.
4. Comme les autres noms de classe, les noms de classe des cas de test doivent correspondre au nom de fichier. **RouterTest.php** doit contenir `class RouterTest extends TestCase`.
5. Le nom de toute méthode contenant un test (par ex : contenant une assertion) doit commencer par `test`, comme dans `testPublished()`. Vous pouvez aussi utiliser l'annotation `@test` pour marquer les méthodes en méthodes de test.

Créer Votre Premier Cas de Test

Dans l'exemple suivant, nous allons créer un cas de test pour une méthode de helper très simple. Le helper que nous allons tester fomatera une barre de progression HTML. Notre helper ressemblera à cela :

```
namespace App\View\Helper;

use Cake\View\Helper;

class ProgressHelper extends Helper
{
    public function bar($value)
    {
        $width = round($value / 100, 2) * 100;

        return sprintf(
            '<div class="progress-container">
                <div class="progress-bar" style="width: %s%%"></div>
            </div>', $width);
    }
}
```

C'est un exemple très simple, mais ce sera utile pour montrer comment créer un cas de test simple. Après avoir créé et sauvegardé notre helper, nous allons créer le fichier de cas de tests dans **tests/TestCase/View/Helper/ProgressHelperTest.php**. Dans ce fichier, nous allons commencer avec ceci :

```
namespace App\Test\TestCase\View\Helper;

use App\View\Helper\ProgressHelper;
use Cake\TestSuite\TestCase;
use Cake\View\View;

class ProgressHelperTest extends TestCase
{
    public function setUp(): void
    {

    }

    public function testBar(): void
    {

    }
}
```

Nous compléterons ce squelette dans une minute. Nous avons ajouté deux méthodes pour commencer. Tout d'abord `setUp()`. Cette méthode est appelée avant chaque méthode de *test* dans une classe de cas de test. Ces méthodes d'initialisation devraient initialiser les objets dont nous aurons besoin pour le test, et faire toute configuration nécessaire. Dans notre méthode d'initialisation, nous allons ajouter ce code :

```
public function setUp(): void
{
    parent::setUp();
    $View = new View();
    $this->Progress = new ProgressHelper($View);
}
```

Appeler la méthode parente est importante dans les cas de test, puisque `TestCase::setUp()` fait un certain nombre de choses comme sauvegarder les valeurs dans *Configure* et stocker les chemins dans *App*.

Ensuite, nous allons remplir la méthode de test. Nous utiliserons quelques assertions pour nous assurer que notre code crée la sortie que nous attendons :

```
public function testBar(): void
{
    $result = $this->Progress->bar(90);
    $this->assertStringContainsString('width: 90%', $result);
    $this->assertStringContainsString('progress-bar', $result);

    $result = $this->Progress->bar(33.3333333);
    $this->assertStringContainsString('width: 33%', $result);
}
```

C'est un test simple mais il montre le bénéfice potentiel de l'utilisation des cas de test. Nous utilisons `assertStringContainsString()` pour nous assurer que notre helper retourne une chaîne qui contient le contenu que nous attendons. Si le résultat ne contient pas le contenu attendu le test sera un échec, et nous savons que notre code est incorrect.

En utilisant les cas de test, vous pouvez décrire la relation entre un ensemble d'entrées connues et leur sortie attendue. Cela vous rend plus confiant dans le code que vous écrivez puisque vous pouvez vérifier que le code que vous avez déjà écrit remplit les attentes et les assertions vérifiées dans vos tests. De plus, puisque les tests sont du code, ils peuvent

être re-lancés à chaque changement. Cela évite la création de nouveaux bugs.

> **Note**
>
> L'EventManager est remis à blanc pour chaque méthode de test. Cela signifie que lorsque vous lancez plusieurs tests en une fois, vous perdez les écouteurs d'events qui ont été enregistrés dans config/bootstrap.php puisque le bootstrap n'est exécuté qu'une seule fois.

Lancer les Tests

Une fois que vous aurez installé PHPUnit et écrit quelques cas de tests, vous voudrez probablement lancer les cas de test très fréquemment. C'est une bonne idée de lancer les tests avant de committer chaque changement pour aider à vous assurer que vous n'avez rien cassé.

En utilisant `phpunit`, vous pouvez lancer les tests de votre application. Pour lancer vos tests d'application, vous pouvez simplement lancer :

```
vendor/bin/phpunit

php phpunit.phar
```

Si vous avez cloné le code source de CakePHP à partir de GitHub [169] et que vous souhaitez exécuter les tests unitaires de CakePHP, n'oubliez pas d'exécuter la commande suivante de `Composer` avant de lancer `phpunit` pour que toutes les dépendances soient installées :

```
composer install
```

À partir du répertoire racine de votre application. Pour lancer les tests pour un plugin qui fait partie de la source de votre application, appliquez d'abord la commande `cd` vers le répertoire du plugin, puis utilisez la commande `phpunit` qui correspond à la façon dont vous avez installé phpunit :

```
cd plugins

../vendor/bin/phpunit

php ../phpunit.phar
```

Pour lancer les tests sur un plugin séparé, vous devez d'abord installer le projet dans un répertoire séparé et installer ses dépendances :

```
git clone git://github.com/cakephp/debug_kit.git
cd debug_kit
php ~/composer.phar install
php ~/phpunit.phar
```

Filtrer les Cas de Test

Quand vous avez de nombreux cas de test et que vous travaillez sur un seul test qui échoue, vous préférerez lancer seulement une partie des méthodes de test. Avec l'exécuteur en ligne de commande vous pouvez utiliser une option pour filtrer les méthodes de test :

169. https://github.com/cakephp/cakephp

```
$ phpunit --filter testSave tests/TestCase/Model/Table/ArticlesTableTest
```

Le paramètre filter est utilisé comme une expression régulière sensible à la casse pour filtrer les méthodes de test à lancer.

Générer une Couverture de Code (Code Coverage)

Vous pouvez générer un rapport de couverture de code depuis la ligne de commande en utilisant les outils de couverture de code intégrés à PHPUnit. PHPUnit va générer un ensemble de fichiers statiques en HTML contenant les résultats de couverture. Vous pouvez générer un rapport de couverture pour un seul cas de test de la façon suivante :

```
$ phpunit --coverage-html webroot/coverage tests/TestCase/Model/Table/ArticlesTableTest
```

Cela placera le résultat de couverture de code dans le répertoire webroot de votre application. Vous pourrez voir les résultats en consultant `http://localhost/votre_app/coverage`.

Vous pouvez aussi utiliser `phpdbg` pour générer la couverture des résultats à la place de xdebug. `phpdbg` est généralement plus rapide dans la génération des rapports de couverture :

```
$ phpdbg -qrr phpunit --coverage-html webroot/coverage tests/TestCase/Model/Table/
↪ArticlesTableTest
```

Combiner les Suites de Test pour les Plugins

Souvent, votre application sera composée de plusieurs plugins. Dans ces situations, il peut être assez fastidieux d'effectuer des tests pour chaque plugin. Vous pouvez faire lancer des tests pour chaque plugin qui compose votre application en ajoutant une section `<testsuite>` supplémentaire dans le fichier `phpunit.xml.dist` de votre application :

```
<testsuites>
    <testsuite name="app">
        <directory>./tests/TestCase</directory>
    </testsuite>

    <!-- Ajouter vos plugins -->
    <testsuite name="forum">
        <directory>./plugins/Forum/tests/TestCase/</directory>
    </testsuite>
</testsuites>
```

Les suites de tests supplémentaires ajoutées à l'élément `<testsuites>` seront exécutées automatiquement quand vous utiliserez `phpunit`.

Si vous vous servez de `<testsuites>` pour utiliser des fixtures à partir de plugins que vous avez installés avec composer, le fichier `composer.json` du plugin doit ajouter le namespace de la fixture dans la section autoload. Par exemple :

```
"autoload-dev": {
    "psr-4": {
        "PluginName\\Test\\Fixture\\": "tests/Fixture/"
    }
},
```

Callbacks du Cycle de Vie des Cas de Test

Les cas de test de nombreux callbacks que vous pouvez utiliser quand pendant les tests :

- `setUp` est appelé avant chaque méthode de test. Doit être utilisé pour créer les objets qui vont être testés, et initialiser les données pour le test. Rappelez-vous de toujours appeler `parent::setUp()`.
- `tearDown` est appelé après chaque méthode de test. Doit être utilisé pour tout nettoyer une fois que le test est terminé. Rappelez-vous de toujours appeler `parent::tearDown()`.
- `setupBeforeClass` est appelé une fois dans chaque cas de test avant que les méthodes de test aient démarré. Cette méthode doit être *statique*.
- `tearDownAfterClass` est appelé une fois dans chaque cas de test après que les méthodes de test aient démarré. Cette méthode doit être *statique*.

Fixtures

Pour tester du code qui dépend de models et d'une base de données, il est possible d'utiliser les **fixtures** comme une façon de créer un état initial pour les tests de votre application. En utilisant des données de fixture, vous réduisez des étapes de configuration répétitives dans vos tests. Les fixtures sont bien adaptées pour des données qui sont communes, ou partagées entre de nombreux tests, voire tous. Les données qui ne sont utiles que dans quelques tests devraient plutôt être créées dans les tests qui en ont besoin.

CakePHP utilise la connexion nommée `test` dans votre fichier de configuration **config/app.php**. Si cette connexion n'est pas utilisable, une exception sera levée et vous ne pourrez pas utiliser les fixtures de base de données.

CakePHP accomplit les étapes suivantes pendant le déroulement d'un test :

1. Création des tables pour chacune des fixtures nécessaires.
2. Remplissage des tables avec des données.
3. Lancement des méthodes de test.
4. Vidage des tables de fixture.

Le schéma pour les fixtures est créé au début d'un test par des migrations ou un dump SQL.

Connexions de Test

Par défaut, CakePHP va faire un alias pour chaque connexion de votre application. Pour chaque connexion définie dans le bootstrap de votre application qui ne commence pas par `test_`, un alias va être créé avec le préfixe `test_`. Le fait d'ajouter des alias de connexions garantit que vous n'utiliserez pas accidentellement la mauvaise connexion dans les cas de test. Les alias de connexions sont transparents pour le reste de votre application. Par exemple, si vous utilisez la connexion "default", vous obtiendrez à la place la connexion `test` dans les cas de test. Si vous utilisez la connexion "replica", la suite de tests tentera d'utiliser "test_replica".

Configuration de PHPUnit

Avant d'utiliser les fixtures vous devez vous assurer que votre `phpunit.xml` contienne l'extension fixture :

```
<!-- dans phpunit.xml -->
<!-- Configurer l'extension pour les fixtures -->
<extensions>
    <extension class="\Cake\TestSuite\Fixture\PHPUnitExtension" />
</extensions>
```

L'extension est incluse par défaut dans votre application et vos plugins générés par `bake`.

Avant 4.3.0, CakePHP utilisait un listener PHPUnit au lieu d'une extension PHPUnit et votre fichier `phpunit.xml` devait contenir :

```
<!-- dans phpunit.xml -->
<!-- Définir un listener pour les fixtures -->
<listeners>
    <listener
    class="\Cake\TestSuite\Fixture\FixtureInjector">
        <arguments>
            <object class="\Cake\TestSuite\Fixture\FixtureManager" />
        </arguments>
    </listener>
</listeners>
```

Le listener est déprécié et vous devriez *mettre à niveau votre configuration de fixture*.

Créer un Schéma de Base de Données de Test

Vous pouvez générer un schéma de base de données de test soit par des migrations de CakePHP, soit en chargeant un fichier de dump SQL, soit en utilisant un autre outil externe de gestion de schéma. Vous devez créer votre schéma dans le fichier `tests/bootstrap.php` de votre application.

Si vous utilisez le *plugin de migrations <https ://book.cakephp.org/migrations>* de CakePHP pour gérer les schémas de votre application, vous pouvez tout aussi bien réutiliser ces migrations pour générer le schéma de votre base de données de test :

```
// dans tests/bootstrap.php
use Migrations\TestSuite\Migrator;

$migrator = new Migrator();

// Configuration simple sans plugin
$migrator->run();

// Lancer les migrations pour plusieurs plugins
$migrator->run(['plugin' => 'Contacts']);

// Lancer les migrations Documents sur la connexion test_docs.
$migrator->run(['plugin' => 'Documents', 'connection' => 'test_docs']);
```

Si vous avez besoin de lancer plusieurs ensembles de migrations, vous pouvez le faire comme ceci :

```
$migrator->runMany([
    // Lancer les migrations de l'application sur la connexion test
    ['connection' => 'test'],
    // Lancer les migrations du plugin Contacts sur la connexion test
    ['plugin' => 'Contacts'],
    // Lancer les migrations du plugin Documents sur la connexion test_docs
    ['plugin' => 'Documents', 'connection' => 'test_docs']
]);
```

L'utilisation de `runMany()` vous garantit que les plugins qui partagent une même base de données ne risquent pas de supprimer des tables quand chaque ensemble de migrations est lancé.

Le plugin de migrations lancera uniquement les migrations qui n'ont pas été appliquées, et réinitialisera les migrations si l'en-tête de votre migration actuelle est différente des migrations appliquées.

Vous pouvez aussi configurer dans vos datasources la façon dont les migrations doivent être lancées dans les tests. Consultez la *documentation des migrations* pour plus d'information.

Pour charger un fichier de dump SQL, vous pouvez faire ceci :

```
// dans tests/bootstrap.php
use Cake\TestSuite\Fixture\SchemaLoader;

// Charger un ou plusieurs fichiers SQL.
(new SchemaLoader())->loadSqlFiles('chemin/vers/le/schema.sql', 'test');
```

Au début du lancement de chaque test, `SchemaLoader` supprimera toutes les tables dans la connexion et les reconstruira à partir du fichier de schéma fourni.

Gestionnaires d'Etat des Fixtures

Par défaut, CakePHP réinitialise l'état des fixtures à la fin de chaque test en tronquant toutes les tables dans la base de données. Cette opération peut devenir coûteuse quand votre application grossit. Si vous utilisez `TransactionStrategy`, chaque méthode de test sera lancée à l'intérieur d'une transaction suivie d'un rollback à la fin du test. Cela peut améliorer vos performances mais nécessite que vos tests ne dépendent pas trop de données statiques de fixtures, car les valeurs des auto-incréments ne sont pas réinitialisées avant chaque test.

La stratégie de gestion de l'état des fixtures peut être définie à l'intérieur du test :

```
use Cake\TestSuite\TestCase;
use Cake\TestSuite\Fixture\FixtureStrategyInterface;
use Cake\TestSuite\Fixture\TransactionStrategy;

class ArticlesTableTest extends TestCase
{
    /**
     * Crée la stratégie de fixtures utilisée pour ce cas de test.
     * Vous pouvez utiliser une classe/un trait pour modifier plusieurs classes.
     */
    protected function getFixtureStrategy(): FixtureStrategyInterface
    {
        return new TransactionStrategy();
    }
}
```

Créer les Fixtures

Les fixtures définissent les enregistrements qui seront insérés dans la base de données au démarrage de chaque test. Créons notre première fixture, qui sera utilisée pour tester notre propre model Article. Créez un fichier nommé **ArticlesFixture.php** dans votre répertoire **tests/Fixture** avec le contenu suivant :

```
namespace App\Test\Fixture;

use Cake\TestSuite\Fixture\TestFixture;

class ArticlesFixture extends TestFixture
{

      // Facultatif. Définissez cette variable pour charger des fixtures avec
      // une base de données de test différente.
      public $connection = 'test';
```

(suite sur la page suivante)

(suite de la page précédente)

```
    public $records = [
        [
            'title' => 'Premier Article',
            'body' => 'Contenu du premier Article',
            'published' => '1',
            'created' => '2007-03-18 10:39:23',
            'modified' => '2007-03-18 10:41:31'
        ],
        [
            'title' => 'Deuxième Article',
            'body' => 'Contenu du deuxième Article',
            'published' => '1',
            'created' => '2007-03-18 10:41:23',
            'modified' => '2007-03-18 10:43:31'
        ],
        [
            'title' => 'Troisième Article',
            'body' => 'Contenu du troisième Article',
            'published' => '1',
            'created' => '2007-03-18 10:43:23',
            'modified' => '2007-03-18 10:45:31'
        ]
    ];
}
```

Note

Il est recommandé de ne pas ajouter manuellement des valeurs dans les colonnesavec autoincrément car cela interfère avec la génération de séquence dans PostgreSQL et SQLServer.

La propriété `$connection` définit la source de données que la fixture va utiliser. Si votre application utilise plusieurs sources de données, vous devriez faire correspondre les fixtures avec les sources de données du model, en ajoutant le préfixe `test_`. Par exemple, si votre model utilise la source de données `mydb`, votre fixture devra utiliser la source de données `test_mydb`. Si la connexion `test_mydb` n'existe pas, vos models utiliseront la source de données `test` par défaut. Les sources de données des fixtures doivent être préfixées par `test` pour réduire la possibilité de tronquer accidentellement toutes les données de votre application en lançant vos tests.

Nous pouvons définir un ensemble d'enregistrements qui seront insérés après la création de la table de fixture. Le format parle de lui-même. `$records` est un tableau d'enregistrements. Chaque item dans `$records` correspond à un enregistrement (une seule ligne). À l'intérieur de chaque ligne, il doit y avoir un tableau associatif des colonnes et valeurs pour la ligne. Gardez juste à l'esprit que tous les enregistrements dans le tableau `$records` doivent avoir les mêmes clés car les lignes sont insérées en une seule requête SQL.

Les Données Dynamiques

Pour utiliser des fonctions ou d'autres données dynamiques dans les enregistrements de vos fixtures, vous pouvez définir vos enregistrements dans la méthode `init()` des fixtures :

```
namespace App\Test\Fixture;

use Cake\TestSuite\Fixture\TestFixture;
```

(suite sur la page suivante)

(suite de la page précédente)

```
class ArticlesFixture extends TestFixture
{
    public function init(): void
    {
        $this->records = [
            [
                'title' => 'Premier Article',
                'body' => 'Contenu du premier Article',
                'published' => '1',
                'created' => date('Y-m-d H:i:s'),
                'modified' => date('Y-m-d H:i:s'),
            ],
        ];
        parent::init();
    }
}
```

Note

Quand vous surchargez `init()`, rappelez-vous juste de toujours appeler `parent::init()`.

Charger les Fixtures dans vos Tests

Une fois que vous avez créé vos fixtures, vous pouvez les utiliser dans vos cas de test. Vous devez charger dans chaque cas de test les fixtures dont vous aurez besoin. Vous devriez charger une fixture pour chaque model sur lequel une requête sera exécutée. Pour charger les fixtures, définissez la propriété `$fixtures` dans votre model :

```
class ArticleTest extends TestCase
{
    protected $fixtures = ['app.Articles', 'app.Comments'];
}
```

À partir de 4.1.0 vous pouvez utiliser `getFixtures()` pour définir votre liste de fixtures depuis une méthode :

```
public function getFixtures(): array
{
    return [
        'app.Articles',
        'app.Comments',
    ];
}
```

Ceci va charger les fixtures d'Article et de Comment à partir du répertoire Fixture de votre application. Vous pouvez aussi charger des fixtures du cœur de CakePHP ou des plugins :

```
class ArticlesTest extends TestCase
{
    protected $fixtures = [
        'plugin.DebugKit.Articles',
        'plugin.MonNomDeVendor/MonPlugin.Messages',
```

(suite sur la page suivante)

(suite de la page précédente)

```
        'core.Comments'
    ];
}
```

Utiliser le préfixe `core` va charger des fixtures de CakePHP, et utiliser un nom de plugin en préfixe chargera la fixture à partir de ce plugin.

Vous pouvez charger les fixtures dans des sous-répertoires. Si votre application est volumineuse, il est plus facile d'organiser vos fixtures en utilisant plusieurs répertoires. Pour charger des fixtures dans des sous-répertoires, incluez simplement le nom du sous-répertoire dans le nom de la fixture :

```
class ArticlesTest extends CakeTestCase
{
    protected $fixtures = ['app.Blog/Articles', 'app.Blog/Comments'];
}
```

Dans l'exemple ci-dessus, les deux fixtures seront chargées à partir de `tests/Fixture/Blog/`.

Fixture Factories

Le nombre et la taille de vos fixtures vont croissant avec la taille votre application. Il est possible qu'à un certain point, vous ne soyez plus en mesure de les maintenir et de suivre leur contenu.

Le plugin fixture factories [170] propose une alternative efficace pour des applications de grande taille.

Le plugin utilise le plugin test suite light [171] pour tronquer avant chaque test toutes les tables modifiées.

Cette commande bake vous aidera créer vos factories :

```
bin/cake bake fixture_factory -h
```

Une fois vos factories mises en place [172], vous voilà équipés pour créer vos fixtures de test à une vitesse folle.

Les interactions inutiles avec la base de donnée vont ralentir les tests ainsi que votre application. Il est possible de créer des fixtures sans les insérer. Ceci est utile lorsque vous testez des méthodes qui n'interagissent pas avec la base de donnée :

```
$article = ArticleFactory::make()->getEntity();
```

Pour insérer dans la base de donnée :

```
$article = ArticleFactory::make()->persist();
```

Les factories peuvent aussi aider à créer des fixtures associées. En supposant que les *Articles belongs to many Authors*, il est possible de créer 5 articles ayant chacun 2 auteurs de la manière suivante :

```
$articles = ArticleFactory::make(5)->with('Authors', 2)->getEntities();
```

Notez que bien que les factories ne nécessitent ni création, ni déclaration de fixtures. Elles sont toujours parfaitement compatibles avec les fixtures qui viennent de CakePHP. Pour plus de détails, rendez-vous ici [173].

170. https://github.com/vierge-noire/cakephp-fixture-factories
171. https://github.com/vierge-noire/cakephp-test-suite-light
172. https://github.com/vierge-noire/cakephp-fixture-factories/blob/main/docs/factories.md
173. https://github.com/vierge-noire/cakephp-fixture-factories

Charger des Routes dans les Tests

Si vous testez des mailers, des components de controllers ou d'autres classes qui ont besoin de routes et de résoudre des URLs, vous aurez besoin de charger des routes. Pendant le `setUp()` d'une classe ou pendant les méthodes de tests individuelles vous pouvez utiliser `loadRoutes()` pour vous assurer que les routes de votre application sont chargées :

```
public function setUp(): void
{
    parent::setUp();
    $this->loadRoutes();
}
```

Cette méthode construira une instance de votre `Application` et appellera la méthode `routes()` dessus. Si votre classe `Application` a besoin de paramètres spécialisés dans le constructeur, vous pouvez les fournir dans `loadRoutes($constructorArgs)`.

Création de routes dans les tests

Parfois, il peut être nécessaire d'ajouter dynamiquement des routes dans les tests, par exemple lors du développement de plugins, ou d'applications qui sont extensibles.

Tout comme le chargement de routes d'applications existantes, ceci peut être fait pendant `setup()` d'une méthode de test, et/ou dans les méthodes de test individuelles elles-mêmes :

```
use Cake\Routing\Route\DashedRoute;
use Cake\Routing\RouteBuilder;
use Cake\Routing\Router;
use Cake\TestSuite\TestCase;

class PluginHelperTest extends TestCase
{
    protected RouteBuilder $routeBuilder;

    public function setUp(): void
    {
        parent::setUp();

        $this->routeBuilder = Router::createRouteBuilder('/');
        $this->routeBuilder->scope('/', function (RouteBuilder $routes) {
            $routes->setRouteClass(DashedRoute::class);
            $routes->get(
                '/test/view/{id}',
                ['controller' => 'Tests', 'action' => 'view']
            );
            // ...
        });

        // ...
    }
}
```

Ceci créera une nouvelle instance de route builder qui fusionnera les routes connectées dans la même collection de routes utilisée par toutes les autres instances de route builder qui qui peuvent déjà exister, ou qui doivent encore être créées dans l'environnement.

Charger des Plugins dans les Tests

Si votre application est censée charger des plugins dynamiquement, vous pouvez utiliser `loadPlugins()` pour charger un ou plusieurs plugins pendant les tests :

```
public function testMethodUsingPluginResources()
{
    $this->loadPlugins(['Company/Cms']);
    // Tester la logique qui nécessite d'avoir chargé Company/Cms.
}
```

Tester les Classes De Tables

Supposons que nous avons déjà notre table Articles définie dans **src/Model/Table/ArticlesTable.php**, et qu'elle ressemble à ceci :

```
namespace App\Model\Table;

use Cake\ORM\Table;
use Cake\ORM\Query;

class ArticlesTable extends Table
{

    public function findPublished(Query $query, array $options): Query
    {
        $query->where([
            $this->alias() . '.published' => 1
        ]);

        return $query;
    }
}
```

Nous voulons maintenant configurer un test qui va tester cette classe de table. Créons un fichier nommé **ArticlesTableTest.php** dans notre répertoire **tests/TestCase/Model/Table**, avec le contenu suivant :

```
namespace App\Test\TestCase\Model\Table;

use App\Model\Table\ArticlesTable;
use Cake\TestSuite\TestCase;

class ArticlesTableTest extends TestCase
{
    protected $fixtures = ['app.Articles'];
}
```

Dans la variable `$fixtures` de notre cas de test, nous définissons l'ensemble des fixtures que nous utiliserons. Vous devriez vous rappeler d'inclure tous les fixtures sur lesquelles des requêtes vont être lancées.

Créer une Méthode de Test

Ajoutons maintenant une méthode pour tester la fonction `published()` dans la table Articles. Modifions le fichier **tests/TestCase/Model/Table/ArticlesTableTest.php** afin qu'il ressemble maintenant à ceci :

```
namespace App\Test\TestCase\Model\Table;

use App\Model\Table\ArticlesTable;
use Cake\TestSuite\TestCase;

class ArticlesTableTest extends TestCase
{
    protected $fixtures = ['app.Articles'];

    public function setUp(): void
    {
        parent::setUp();
        $this->Articles = $this->getTableLocator()->get('Articles');
    }

    public function testFindPublished(): void
    {
        $query = $this->Articles->find('published')->all();
        $this->assertInstanceOf('Cake\ORM\Query', $query);
        $result = $query->enableHydration(false)->toArray();
        $expected = [
            ['id' => 1, 'title' => 'Premier Article'],
            ['id' => 2, 'title' => 'Deuxième Article'],
            ['id' => 3, 'title' => 'Troisième Article']
        ];

        $this->assertEquals($expected, $result);
    }
}
```

Vous pouvez voir que nous avons ajouté une méthode appelée `testFindPublished()`. Nous commençons par créer une instance de notre classe `ArticleTable`, et lançons ensuite notre méthode `find('published')`. Dans `$expected`, nous définissons ce qui devrait être le résultat approprié (que nous connaissons puisque nous avons défini les enregistrements qui sont remplis initialement dans la table articles). Nous testons si les résultats correspondent à nos attentes en utilisant la méthode `assertEquals()`. Lisez la section sur les *Lancer les Tests* pour plus d'informations sur la façon de lancer les cas de test.

En utilisant les fixture factories, le test se présenterait ainsi :

```
namespace App\Test\TestCase\Model\Table;

use App\Test\Factory\ArticleFactory;
use Cake\TestSuite\TestCase;

class ArticlesTableTest extends TestCase
{
    public function testFindPublished(): void
    {
        // Insérer 3 articles publiés
        $articles = ArticleFactory::make(['published' => 1], 3)->persist();
```

(suite sur la page suivante)

(suite de la page précédente)

```
        // Insérer 2 articles non publiés
        ArticleFactory::make(['published' => 0], 2)->persist();

        $result = ArticleFactory::find('published')->find('list')->toArray();

        $expected = [
            $articles[0]->id => $articles[0]->title,
            $articles[1]->id => $articles[1]->title,
            $articles[2]->id => $articles[2]->title,
        ];

        $this->assertEquals($expected, $result);
    }
}
```

Aucune fixture n'a besoin d'être déclarée. Les 5 articles créés n'existeront que dans ce test. La méthode statique `::find()` va requêter la base de donnée sans utiliser la table `ArticlesTable` ni ses évènements.

Méthodes de Mocking des Models

Il y aura des fois où vous voudrez mocker les méthodes des models quand vous les testez. Vous devrez utiliser `getMockForModel` pour créer des mocks de test des classes de tables. Cela évite des problèmes avec les propriétés réfléchies qu'ont les mocks normaux :

```
public function testSendingEmails(): void
{
    $model = $this->getMockForModel('EmailVerification', ['send']);
    $model->expects($this->once())
        ->method('send')
        ->will($this->returnValue(true));

    $model->verifyEmail('test@example.com');
}
```

Dans votre méthode `tearDown()`, assurez-vous de retirer le mock avec ceci :

```
$this->getTableLocator()->clear();
```

Tests d'Intégration des Controllers

Alors que vous pouvez tester les controllers de la même manière que les Helpers, Models et Components, CakePHP offre une classe spécialisée `IntegrationTestCase`. L'utilisation de ce trait dans vos cas de test de vos controllers vous offre une interface de test de haut niveau.

Si vous n'êtes pas familier avec les tests d'intégration, dites-vous qu'il s'agit d'une approche de test qui permet de test de plusieurs éléments qui fonctionnent de concert. Les fonctionnalités de test d'intégration dans CakePHP simulent une requête HTTP à traiter par votre application. Par exemple, tester vos controllers impactera également vos components, models et helpers qui auraient été invoqués pour traiter la requête HTTP. Cela vous donne un test de plus haut niveau sur votre application et tous ses composants.

Supposons que vous ayez un controller typique ArticlesController, et son model correspondant. Le code du controller ressemble à ceci :

```
namespace App\Controller;

use App\Controller\AppController;

class ArticlesController extends AppController
{
    public function index($short = null)
    {
        if ($this->request->is('post')) {
            $article = $this->Articles->newEntity($this->request->getData());
            if ($this->Articles->save($article)) {
                // Redirige selon le pattern PRG
                return $this->redirect(['action' => 'index']);
            }
        }
        if (!empty($short)) {
            $result = $this->Articles->find('all', [
                    'fields' => ['id', 'title']
                ])
                ->all();
        } else {
            $result = $this->Articles->find()->all();
        }

        $this->set([
            'title' => 'Articles',
            'articles' => $result
        ]);
    }
}
```

Créez un fichier nommé **ArticlesControllerTest.php** dans votre répertoire **tests/TestCase/Controller** et mettez-y ce qui suit :

```
namespace App\Test\TestCase\Controller;

use Cake\TestSuite\IntegrationTestTrait;
use Cake\TestSuite\TestCase;

class ArticlesControllerTest extends TestCase
{
    use IntegrationTestTrait;

    protected $fixtures = ['app.Articles'];

    public function testIndex(): void
    {
        $this->get('/articles');

        $this->assertResponseOk();
        // D'autres assertions.
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    public function testIndexQueryData(): void
    {
        $this->get('/articles?page=1');

        $this->assertResponseOk();
        // D'autres assertions.
    }

    public function testIndexShort(): void
    {
        $this->get('/articles/index/short');

        $this->assertResponseOk();
        $this->assertResponseContains('Articles');
        // D'autres assertions.
    }

    public function testIndexPostData(): void
    {
        $data = [
            'user_id' => 1,
            'published' => 1,
            'slug' => 'nouvel-article',
            'title' => 'Nouvel Article',
            'body' => 'Nouveau Contenu'
        ];
        $this->post('/articles', $data);

        $this->assertResponseSuccess();
        $articles = $this->getTableLocator()->get('Articles');
        $query = $articles->find()->where(['title' => $data['title']]);
        $this->assertEquals(1, $query->count());
    }
}
```

Cet exemple montre quelques exemples de méthodes qui envoient des requêtes et quelques assertions fournies par `IntegrationTestTrait`. Avant de pouvoir utiliser les assertions, vous aurez besoin de simuler une requête. Vous pouvez utiliser l'une des méthodes suivantes pour envoyer une requête :

- `get()` Envoie une requête GET.
- `post()` Envoie une requête POST.
- `put()` Envoie une requête PUT.
- `delete()` Envoie une requête DELETE.
- `patch()` Envoie une requête PATCH.
- `options()` Envoie une requête OPTIONS.
- `head()` Envoie une requête HEAD.

Toutes les méthodes exceptées `get()` et `delete()` acceptent un second paramètre qui vous permet d'envoyer le corps d'une requête. Après avoir émis une requête, vous pouvez utiliser les différentes assertions fournies par `IntegrationTestTrait` ou PHPUnit afin de vous assurer que votre requête a les effets attendus.

Configurer la Requête

Le trait `IntegrationTestTrait` comporte de nombreux helpers pour configurer les requêtes que vous allez envoyer à l'application testée :

```
// Définit des cookies
$this->cookie('name', 'Oncle Bob');

// Définit des données de session
$this->session(['Auth.User.id' => 1]);

// Configure les en-têtes
$this->configRequest([
    'headers' => ['Accept' => 'application/json']
]);
```

Les états définis par les méthodes de cet utilitaire sont remis à zéro dans la méthode `tearDown()`.

Tester Des Actions Qui Nécessitent Une Authentification

Si vous utilisez `AuthComponent`, vous aurez besoin de simuler les données de session utilisées par AuthComponent pour valider l'identité d'un utilisateur. Pour ce faire, vous pouvez utiliser les méthodes utilitaires fournies par `IntegrationTestTrait`. En admettant que vous ayez un `ArticlesController` qui contient une méthode add, et que cette méthode nécessite une authentification, vous pourriez écrire les tests suivants :

```
public function testAddUnauthenticatedFails(): void
{
    // Pas de données de session définies.
    $this->get('/articles/add');

    $this->assertRedirect(['controller' => 'Users', 'action' => 'login']);
}

public function testAddAuthenticated(): void
{
    // Définit des données de session
    $this->session([
        'Auth' => [
            'User' => [
                'id' => 1,
                'username' => 'testing',
                // autres clés.
            ]
        ]
    ]);
    $this->get('/articles/add');

    $this->assertResponseOk();
    // Autres assertions.
}
```

Test de l'Authentification Stateless (sans état) et des APIs

Pour tester les APIs qui utilisent une authentification stateless, telles que l'authentification Basic, vous pouvez configurer la requête de manière à y injecter des variables d'environnement et des en-têtes qui vont simuler de vraies en-têtes d'authentification.

Lorsque vous testez les authentifications Basic ou Digest, vous pouvez ajouter les variables d'environnement *créées automatiquement par PHP <https ://php.net/manual/fr/features.http-auth.php> `_. Ces variables d'environnement utilisées dans l'adaptateur d'authentification sont décrites dans : ref : `basic-authentication*

```
public function testBasicAuthentication(): void
{
    $this->configRequest([
        'environment' => [
            'PHP_AUTH_USER' => 'username',
            'PHP_AUTH_PW' => 'password',
        ]
    ]);

    $this->get('/api/posts');
    $this->assertResponseOk();
}
```

Si vous testez d'autres types d'authentification, tel que OAuth2, vous pouvez définir l'en-tête d'autorisation directement :

```
public function testOauthToken(): void
{
    $this->configRequest([
        'headers' => [
            'authorization' => 'Bearer: oauth-token'
        ]
    ]);

    $this->get('/api/posts');
    $this->assertResponseOk();
}
```

Vous pouvez utiliser la clé `headers` dans `configRequest()` pour configurer n'importe quelle autre en-tête HTTP dont vous auriez besoin pour cette action.

Tester les Actions Protégées par CsrfProtectionMiddleware ou FormProtectionComponent

Quand vous testez des actions protégées par `CsrfProtectionMiddleware` ou `FormProtectionComponent`, vous pouvez activer la génération automatique de token pour vous assurer que vos tests ne vont pas échoué à cause d'un problème de token :

```
public function testAdd(): void
{
    $this->enableCsrfToken();
    $this->enableSecurityToken();
    $this->post('/posts/add', ['title' => 'News excitante!']);
}
```

Il est aussi important d'activer le débogage dans les tests qui utilisent des tokens pour éviter que le `FormProtectionComponent` ne pense que le token de débogage est utilisé dans un environnement non-debug. Quand vous faites des tests avec d'autres méthodes comme `requireSecure()`, vous pouvez utiliser `configRequest()` pour définir les bonnes variables d'environnement :

```
// Fake out SSL connections.
$this->configRequest([
    'environment' => ['HTTPS' => 'on']
]);
```

Si votre action a besoin de champs déverrouillés vous pouvez les déclarer avec `setUnlockedFields()` :

```
$this->setUnlockedFields(['dynamic_field']);
```

Test d'intégration sur les middlewares PSR-7

Les tests d'intégration peuvent aussi être utilisés pour tester entièrement vos applications PSR-7 et les /controllers/middleware. Par défaut, `IntegrationTestTrait` détectera automatiquement la présence d'une classe `App\Application` et activera automatiquement les tests d'intégration sur votre Application.

Vous pouvez personnaliser le nom de la classe Application utilisée ainsi que les arguments du contructeur, en utilisant la méthode `configApplication()` :

```
public function setUp(): void
{
    $this->configApplication('App\App', [CONFIG]);
}
```

Vous devriez également faire en sorte d'utiliser application-bootstrap pour charger les plugins qui contiennent des événements ou des routes. De cette manière, vous vous assurez que les événements et les routes seront connectés pour tous vos *test cases*.

Tester avec des Cookies Cryptés

Si vous utilisez le encrypted-cookie-middleware dans votre application, il y a des méthodes pratiques pour définir des cookies chiffrés dans vos *test cases* :

```
// Définit un cookie en utilisant AES et la clé par défaut.
$this->cookieEncrypted('mon_cookie', 'Des valeurs secrètes');

// Partons du principe que cette action modifie le cookie.
$this->get('/bookmarks/index');

$this->assertCookieEncrypted('Une nouvelle valeur', 'mon_cookie');
```

Tester les Messages Flash

Si vous souhaitez faire une assertion sur la présence de messages Flash en session et pas sur le rendu du HTML, vous pouvez utiliser `enableRetainFlashMessages()` dans vos tests pour que les messages Flash soient conservés dans la session et que vous puissez ainsi effectuer vos assertions :

```
// Active la rétention des messages flash plutôt que leur consommation
$this->enableRetainFlashMessages();
$this->get('/bookmarks/delete/9999');
```

(suite sur la page suivante)

(suite de la page précédente)

```
$this->assertSession("Ce marque-page n'existe pas", 'Flash.flash.0.message');

// Vérifie un message flash sous la clé 'flash'.
$this->assertFlashMessage('Marque-page supprimé', 'flash');

// Vérifie le deuxième message flash, également sous la clé 'flash'.
$this->assertFlashMessageAt(1, 'Marque-page vraiment supprimé');

// Vérifie un message flash en première position sous la clé 'auth'
$this->assertFlashMessageAt(0, "Vous n'avez pas le droit d'entrer dans ce donjon !",
↪'auth');

// Vérifie qu'un message flash utilise l'élément error
$this->assertFlashElement('Flash/error');

// Vérifie l'élément du deuxième message flash
$this->assertFlashElementAt(1, 'Flash/error')
```

Tester un Controller Retournant du JSON

JSON est un format commun et agréable à utiliser lors de la conception d'un service web. Avec CakePHP, il est très simple de tester les terminaux de votre service web. Commençons avec un exemple simple de controller qui renvoie du JSON :

```
use Cake\View\JsonView;

class MarkersController extends AppController
{
    public function viewClasses(): array
    {
        return [JsonView::class];
    }

    public function view($id)
    {
        $marker = $this->Markers->get($id);
        $this->set('marker', $marker);
        $this->viewBuilder()->setOption('serialize', ['marker']);
    }
}
```

Créons maintenant le fichier **tests/TestCase/Controller/MarkersControllerTest.php** et assurons-nous que le web service répond correctement :

```
class MarkersControllerTest extends IntegrationTestCase
{
    public function testGet(): void
    {
        $this->configRequest([
            'headers' => ['Accept' => 'application/json']
        ]);
```

(suite sur la page suivante)

(suite de la page précédente)

```
        $result = $this->get('/markers/view/1.json');

        // Vérification que la réponse était bien une 200
        $this->assertResponseOk();

        $expected = [
            ['id' => 1, 'lng' => 66, 'lat' => 45],
        ];
        $expected = json_encode($expected, JSON_PRETTY_PRINT);
        $this->assertEquals($expected, (string)$this->_response->getBody());
    }
}
```

Nous utilisons l'option `JSON_PRETTY_PRINT` car la vue JsonView intégrée à CakePHP utilise cette option quand le mode `debug` est activé.

Test avec Téléversement de Fichiers

La simulation d'un téléversement de fichiers est enfantine lorsque vous utilisez le mode par défaut de « *téléversement de fichiers en tant qu'objets* ». Vous pouvez créer tout simplement des instances qui implémentent \Psr\Http\Message\UploadedFileInterface [174] (l'implémentation par défaut actuellement utilisée par CakePHP est `\Laminas\Diactoros\UploadedFile`), et les passer dans les données de vos requêtes de test. Dans l'environnement CLI, ces objets passeront par défaut les contrôles de validation qui testent si le fichier a été téléversé via HTTP. Il n'en va pas de même pour les données de type tableau comme celles que l'on trouve dans `$_FILES` ; ce contrôle échouerait.

Afin de simuler exactement la façon dont les objets de fichiers téléversés seraient présents dans une requête normale, vous devez non seulement les passer dans les données de la requête, mais aussi les passer dans la configuration de la requête de test via l'option `files`. Ce n'est toutefois pas techniquement nécessaire, sauf si votre code accède aux fichiers téléversés via les méthodes *`Cake\Http\ServerRequest::getUploadedFile()`* ou *`Cake\Http\ServerRequest::getUploadedFiles()`*.

Supposons que les articles aient une image d'accroche, et une association `Articles hasMany Attachments`. Le formulaire ressemblerait à quelque chose comme ceci en conséquence, où un fichier image et plusieurs fichiers ou pièces jointes seraient acceptés :

```
<?= $this->Form->create($article, ['type' => 'file']) ?>
<?= $this->Form->control('title') ?>
<?= $this->Form->control('teaser_image', ['type' => 'file']) ?>
<?= $this->Form->control('attachments.0.attachment', ['type' => 'file']) ?>
<?= $this->Form->control('attachments.0.description']) ?>
<?= $this->Form->control('attachments.1.attachment', ['type' => 'file']) ?>
<?= $this->Form->control('attachments.1.description']) ?>
<?= $this->Form->button('Submit') ?>
<?= $this->Form->end() ?>
```

Le test qui simulerait la requête correspondante pourrait ressembler à ceci :

```
public function testAddWithUploads(): void
{
    $teaserImage = new \Laminas\Diactoros\UploadedFile(
        '/path/to/test/file.jpg', // flux ou chemin d'accès au fichier représentant le␣
↪fichier temporaire
```

(suite sur la page suivante)

174. https://www.php-fig.org/psr/psr-7/#16-uploaded-files

(suite de la page précédente)

```
        12345,                      // la taille du fichier en octets
        \UPLOAD_ERR_OK,             // le statut de téléversement ou d'erreur
        'teaser.jpg',               // le nom du fichier tel qu'il aurait été envoyé par le␣
↪client
        'image/jpeg'                // le type mime tel qu'il aurait été envoyé par le␣
↪client
    );

    $textAttachment = new \Laminas\Diactoros\UploadedFile(
        '/path/to/test/file.txt',
        12345,
        \UPLOAD_ERR_OK,
        'attachment.txt',
        'text/plain'
    );

    $pdfAttachment = new \Laminas\Diactoros\UploadedFile(
        '/path/to/test/file.pdf',
        12345,
        \UPLOAD_ERR_OK,
        'attachment.pdf',
        'application/pdf'
    );

    // Voici les données accessibles via `$this->request->getUploadedFile()`
    // et `$this->request->getUploadedFiles()`.
    $this->configRequest([
        'files' => [
            'teaser_image' => $teaserImage,
            'attachments' => [
                0 => [
                    'attachment' => $textAttachment,
                ],
                1 => [
                    'attachment' => $pdfAttachment,
                ],
            ],
        ],
    ]);

    // Voici les données accessibles via `$this->request->getData()`.
    $postData = [
        'title' => 'Nouvel Article',
        'teaser_image' => $teaserImage,
        'attachments' => [
            0 => [
                'attachment' => $textAttachment,
                'description' => 'Fichier texte',
            ],
            1 => [
                'attachment' => $pdfAttachment,
                'description' => 'Fichier PDF',
```

(suite sur la page suivante)

(suite de la page précédente)

```
            ],
        ],
    ];
    $this->post('/articles/add', $postData);

    $this->assertResponseOk();
    $this->assertFlashMessage("L'article a été sauvegardé avec succès");
    $this->assertFileExists('/path/to/uploads/teaser.jpg');
    $this->assertFileExists('/path/to/uploads/attachment.txt');
    $this->assertFileExists('/path/to/uploads/attachment.pdf');
}
```

Astuce

Si vous configurez la requête de test avec des fichiers, alors elle *doit* correspondre à la structure de vos données POST (mais n'inclure que les objets de fichiers téléversés) !

De même, vous pouvez simuler des erreurs de téléversement [175] ou d'autres fichiers invalides qui ne passent pas la validation :

```
public function testAddWithInvalidUploads(): void
{
    $missingTeaserImageUpload = new \Laminas\Diactoros\UploadedFile(
        '',
        0,
        \UPLOAD_ERR_NO_FILE,
        '',
        ''
    );

    $uploadFailureAttachment = new \Laminas\Diactoros\UploadedFile(
        '/path/to/test/file.txt',
        1234567890,
        \UPLOAD_ERR_INI_SIZE,
        'attachment.txt',
        'text/plain'
    );

    $invalidTypeAttachment = new \Laminas\Diactoros\UploadedFile(
        '/path/to/test/file.exe',
        12345,
        \UPLOAD_ERR_OK,
        'attachment.exe',
        'application/vnd.microsoft.portable-executable'
    );

    $this->configRequest([
        'files' => [
            'teaser_image' => $missingTeaserImageUpload,
```

(suite sur la page suivante)

175. https://www.php.net/manual/fr/features.file-upload.errors.php

(suite de la page précédente)

```
            'attachments' => [
                0 => [
                    'file' => $uploadFailureAttachment,
                ],
                1 => [
                    'file' => $invalidTypeAttachment,
                ],
            ],
        ],
    ]);

    $postData = [
        'title' => 'Nouvel Article',
        'teaser_image' => $missingTeaserImageUpload,
        'attachments' => [
            0 => [
                'file' => $uploadFailureAttachment,
                'description' => 'Pièce jointe en échec de téléversement',
            ],
            1 => [
                'file' => $invalidTypeAttachment,
                'description' => 'Pièce jointe de type invalide',
            ],
        ],
    ];
    $this->post('/articles/add', $postData);

    $this->assertResponseOk();
    $this->assertFlashMessage("L'article n'a pas pu être sauvegardé");
    $this->assertResponseContains("Une image d'accroche est nécessaire");
    $this->assertResponseContains('Dépassement de la taille maximale autorisée des␣
↪fichiers');
    $this->assertResponseContains('Type de fichier non supporté');
    $this->assertFileNotExists('/path/to/uploads/teaser.jpg');
    $this->assertFileNotExists('/path/to/uploads/attachment.txt');
    $this->assertFileNotExists('/path/to/uploads/attachment.exe');
}
```

Désactiver le Middleware de Gestion d'Erreurs dans les Tests

Quand vous déboguez des tests qui échouent parce que votre application rencontre des erreurs, il peut être utile de désactiver temporairement le middleware de gestion des erreurs pour permettre aux erreurs de remonter. Pour cela, vous pouvez utiliser la méthode `disableErrorHandlerMiddleware()` :

```
public function testGetMissing(): void
{
    $this->disableErrorHandlerMiddleware();
    $this->get('/markers/not-there');
    $this->assertResponseCode(404);
}
```

Dans l'exemple ci-dessus, le test échouera et le message d'exception et la stack-trace seront affichés à la place de la page d'erreur de l'application.

Méthodes d'Assertion

Le trait `IntegrationTestTrait` fournit de nombreuses méthodes d'assertions afin de tester les réponses plus simplement. Quelques exemples :

```
// Vérifie un code de réponse 2xx
$this->assertResponseOk();

// Vérifie un code de réponse 2xx/3xx
$this->assertResponseSuccess();

// Vérifie un code de réponse 4xx
$this->assertResponseError();

// Vérifie un code de réponse 5xx
$this->assertResponseFailure();

// Vérifie un code de réponse spécifique, par exemple 200
$this->assertResponseCode(200);

// Vérifie l'en-tête Location
$this->assertRedirect(['controller' => 'Articles', 'action' => 'index']);

// Vérifie qu'aucun en-tête Location n'a été envoyé
$this->assertNoRedirect();

// Vérifie une partie de l'en-tête Location
$this->assertRedirectContains('/articles/edit/');

// Vérifie que l'en-tête location ne contient pas...
$this->assertRedirectNotContains('/articles/edit/');

// Vérifie que le contenu de la réponse n'est pas vide
$this->assertResponseNotEmpty();

// Vérifie que le contenu de la réponse est vide
$this->assertResponseEmpty();

// Vérifie le contenu de la réponse
$this->assertResponseEquals('Ouais !');

// Vérifie que le contenu de la réponse n'est pas égal à...
$this->assertResponseNotEquals('Non !');

// Vérifie un contenu partiel de la réponse
$this->assertResponseContains("C'est gagné !");
$this->assertResponseNotContains("Encore raté !");

// Vérifie un fichier renvoyé
$this->assertFileResponse('/chemin/absolu/vers/le/fichier.ext');

// Vérifie le layout
$this->assertLayout('default');
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Vérifie quel Template a été rendu (s'il y en a un)
$this->assertTemplate('index');

// Vérifie les données de la session
$this->assertSession(1, 'Auth.User.id');

// Vérifie l'en-tête de la réponse.
$this->assertHeader('Content-Type', 'application/json');
$this->assertHeaderContains('Content-Type', 'html');

// Vérifie que l'en-tête de la réponse ne contient pas de xml
$this->assertHeaderNotContains('Content-Type', 'xml');

// Vérifie le contenu d'une variable.
$user =  $this->viewVariable('user');
$this->assertEquals('jose', $user->username);

// Vérifie les cookies.
$this->assertCookie('1', 'thingid');

// Vérifie le type de contenu
$this->assertContentType('application/json');
```

En plus des méthodes d'assertion ci-dessus, vous pouvez également utiliser toutes les assertions de TestSuite [176] et celles de PHPUnit [177].

Comparer les Résultats du Test avec un Fichier

Pour certains types de test, il peut être plus simple de comparer les résultats d'un test avec le contenu d'un fichier - par exemple, quand vous testez le rendu d'une vue. `StringCompareTrait` ajoute une méthode d'assertion simple pour cela.

Pour l'utiliser, vous devez inclure le trait, définir le répertoire contenant le fichier servant de base de comparaison et appeler `assertSameAsFile` :

```
use Cake\TestSuite\StringCompareTrait;
use Cake\TestSuite\TestCase;

class SomeTest extends TestCase
{
    use StringCompareTrait;

    public function setUp(): void
    {
        $this->_compareBasePath = APP . 'tests' . DS . 'comparisons' . DS;
        parent::setUp();
    }

    public function testExample(): void
    {
        $result = ...;
```

(suite sur la page suivante)

176. https://api.cakephp.org/4.x/class-Cake.TestSuite.TestCase.html
177. https://phpunit.de/manual/current/fr/appendixes.assertions.html

(suite de la page précédente)

```
        $this->assertSameAsFile('example.php', $result);
    }
}
```

Cet exemple va comparer `$result` au contenu du fichier `APP/tests/comparisons/example.php`.

Il existe un mécanisme pour écrire/mettre à jour les fichiers de test, en définissant la variable d'environment `UPDATE_TEST_COMPARISON_FILES`, ce qui va créer et/ou mettre à jour les fichiers de comparaison de test dès qu'ils seront référencés :

```
phpunit
...
FAILURES!
Tests: 6, Assertions: 7, Failures: 1

UPDATE_TEST_COMPARISON_FILES=1 phpunit
...
OK (6 tests, 7 assertions)

git status
...
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#   modified:   tests/comparisons/example.php
```

Tests d'Intégration en Console

Voir la *Tester les Commandes* pour savoir comment tester les commandes.

Mocker les Injections de Dépendances

Voir mocking-services-in-tests pour savoir comment remplacer des services injectés avec le conteneur d'injection de dépendances dans vos tests d'intégration.

Mocker les Réponses du Client HTTP

Voir *Tests* pour savoir comment créer des mocks de réponses vers des API externes.

Tester les Views

Généralement, la plupart des applications ne vont pas directement tester leur code HTML. Le faire donne souvent des suites de tests fragiles, difficiles à maintenir et sujettes à se casser. En écrivant des tests fonctionnels avec `IntegrationTestTrait`, vous pouvez inspecter le contenu de la vue rendue en configurant l'option `return` à "view". Bien qu'il soit possible de tester le contenu de la vue en utilisant `IntegrationTestTrait`, il est aussi possible de réaliser des tests d'intégration/de vue plus robustes plus maintenables en utilisant des outils comme Selenium webdriver [178].

178. https://www.selenium.dev/

Tester les Components

Imaginons que nous ayons dans notre application un component appelé PagematronComponent. Ce component nous aide à fixer la valeur limite de pagination dans tous les controllers qui l'utilisent. Voici notre exemple de component situé dans **src/Controller/Component/PagematronComponent.php** :

```
class PagematronComponent extends Component
{
    public $controller = null;

    public function setController($controller)
    {
        $this->controller = $controller;
        // Assurez-vous que le controller utilise la pagination.
        if (!isset($this->controller->paginate)) {
            $this->controller->paginate = [];
        }
    }

    public function startup(EventInterface $event)
    {
        $this->setController($event->getSubject());
    }

    public function adjust($length = 'short'): void
    {
        switch ($length) {
            case 'long':
                $this->controller->paginate['limit'] = 100;
            break;
            case 'medium':
                $this->controller->paginate['limit'] = 50;
            break;
            default:
                $this->controller->paginate['limit'] = 20;
            break;
        }
    }
}
```

Maintenant nous pouvons écrire des tests pour nous assurer que notre paramètre de pagination `limit` est défini correctement par la méthode `adjust()` dans notre component. Nous créons le fichier **tests/TestCase/Controller/Component/PagematronComponentTest.php** :

```
namespace App\Test\TestCase\Controller\Component;

use App\Controller\Component\PagematronComponent;
use Cake\Controller\Controller;
use Cake\Controller\ComponentRegistry;
use Cake\Event\Event;
use Cake\Http\ServerRequest;
use Cake\Http\Response;
use Cake\TestSuite\TestCase;
```

(suite sur la page suivante)

(suite de la page précédente)

```
class PagematronComponentTest extends TestCase
{
    protected $component;
    protected $controller;

    public function setUp(): void
    {
        parent::setUp();
        // Configuration de notre component et de notre faux controller de test.
        $request = new ServerRequest();
        $response = new Response();
        $this->controller = $this->getMockBuilder('Cake\Controller\Controller')
            ->setConstructorArgs([$request, $response])
            ->setMethods(null)
            ->getMock();
        $registry = new ComponentRegistry($this->controller);
        $this->component = new PagematronComponent($registry);
        $event = new Event('Controller.startup', $this->controller);
        $this->component->startup($event);
    }

    public function testAdjust(): void
    {
        // Test de notre méthode avec différents paramètres.
        $this->component->adjust();
        $this->assertEquals(20, $this->controller->paginate['limit']);

        $this->component->adjust('medium');
        $this->assertEquals(50, $this->controller->paginate['limit']);

        $this->component->adjust('long');
        $this->assertEquals(100, $this->controller->paginate['limit']);
    }

    public function tearDown(): void
    {
        parent::tearDown();
        // Nettoie les variables quand les tests sont finis.
        unset($this->component, $this->controller);
    }
}
```

Tester les Helpers

Puisqu'une bonne quantité de logique se situe dans les classes de helpers, il est important de s'assurer que ces classes sont couvertes par des tests.

Tout d'abord, créons un exemple de helper à tester. Le `CurrencyRendererHelper` va nous aider à afficher les monnaies dans nos vues et pour simplifier, il ne va avoir qu'une méthode `usd()` :

```
// src/View/Helper/CurrencyRendererHelper.php
namespace App\View\Helper;

use Cake\View\Helper;

class CurrencyRendererHelper extends Helper
{
    public function usd($amount): string
    {
        return 'USD ' . number_format($amount, 2, '.', ',');
    }
}
```

Ici nous configurons deux décimales après la virgule, le séparateur décimal est un point, le séparateur de milliers est une virgule, et nous plaçons la chaîne "USD" en préfixe.

Maintenant, créons nos tests :

```
// tests/TestCase/View/Helper/CurrencyRendererHelperTest.php

namespace App\Test\TestCase\View\Helper;

use App\View\Helper\CurrencyRendererHelper;
use Cake\TestSuite\TestCase;
use Cake\View\View;

class CurrencyRendererHelperTest extends TestCase
{

    public $helper = null;

    // Ici nous instancions notre helper
    public function setUp(): void
    {
        parent::setUp();
        $View = new View();
        $this->helper = new CurrencyRendererHelper($View);
    }

    // Test de la fonction usd()
    public function testUsd(): void
    {
        $this->assertEquals('USD 5.30', $this->helper->usd(5.30));

        // Nous devrions toujours avoir 2 chiffres après la virgule
        $this->assertEquals('USD 1.00', $this->helper->usd(1));
        $this->assertEquals('USD 2.05', $this->helper->usd(2.05));

        // Test du séparateur de milliers
        $this->assertEquals(
          'USD 12,000.70',
          $this->helper->usd(12000.70)
        );
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }
}
```

Ici nous appelons `usd()` avec différents paramètres et demandons à notre test de vérifier si les valeurs retournées sont égales à ce que nous attendons.

Sauvegardez cela et exécutez le test. Vous devriez voir une barre verte et un message indiquant 1 test passé et 4 assertions.

Lorsque vous testez un Helper qui utilise d'autres helpers, assurez-vous de mocker la méthode `loadHelpers` de la classe View.

Tester les Events

Les *Événements système* sont un bon moyen pour découpler le code de votre application. Mais parfois quand vous les testez, vous aurez tendance à tester les événements dans les cas de test qui exécutent ces événements. C'est une forme supplémentaire de couplage qui peut être évitée en utilisant à la place `assertEventFired` et `assertEventFiredWith`.

En poursuivant l'exemple sur les Orders, supposons que nous avons les tables suivantes :

```
class OrdersTable extends Table
{
    public function place($order): bool
    {
        if ($this->save($order)) {
            // la suppression du panier a été déplacée dans CartsTable
            $event = new Event('Model.Order.afterPlace', $this, [
                'order' => $order
            ]);
            $this->getEventManager()->dispatch($event);

            return true;
        }

        return false;
    }
}

class CartsTable extends Table
{
    public function implementedEvents(): array
    {
        return [
            'Model.Order.afterPlace' => 'removeFromCart'
        ];
    }

    public function removeFromCart(EventInterface $event): void
    {
        $order = $event->getData('order');
        $this->delete($order->cart_id);
    }
}
```

> **Note**
>
> Pour faire des assertions sur le fait que des événements sont déclenchés, vous devez d'abord activer *Suivre la Trace des Événements* sur le gestionnaire d'événements sur lequel vous souhaitez faire des assertions.

Pour tester la `OrdersTable` ci-dessus, nous devons activer le tracking dans la méthode `setUp()` puis vérifier que l'événement a été déclenché, puis que l'entity `$order` a été passée dans les données de l'événement :

```
namespace App\Test\TestCase\Model\Table;

use App\Model\Table\OrdersTable;
use Cake\Event\EventList;
use Cake\TestSuite\TestCase;

class OrdersTableTest extends TestCase
{
    protected $fixtures = ['app.Orders'];

    public function setUp(): void
    {
        parent::setUp();

        $this->Orders = $this->getTableLocator()->get('Orders');

        // activer le suivi de la trace des événements
        $this->Orders->getEventManager()->setEventList(new EventList());
    }

    public function testPlace(): void
    {
        $order = new Order([
            'user_id' => 1,
            'item' => 'Cake',
            'quantity' => 42,
        ]);

        $this->assertTrue($this->Orders->place($order));

        $this->assertEventFired('Model.Order.afterPlace', $this->Orders->
↪getEventManager());
        $this->assertEventFiredWith('Model.Order.afterPlace', 'order', $order, $this->
↪Orders->getEventManager());
    }
}
```

Par défaut, les assertions utilisent l'`EventManager` global, donc il n'est pas nécessaire de passer le gestionnaire d'événements pour tester les événements globaux :

```
$this->assertEventFired('Mon.Event.Global');
$this->assertEventFiredWith('Mon.Event.Global', 'user', 1);
```

Testing Email

Consultez *Tester les Mailers* pour savoir comment tester les emails.

Créer des Suites de Test (Test Suites)

Si vous voulez que plusieurs de vos tests s'exécutent en même temps, vous pouvez créer une suite de tests. Une suite de test est composée de plusieurs cas de test. Vous pouvez créer des suites de tests dans le fichier `phpunit.xml` de votre application. Un exemple simple serait :

```
<testsuites>
  <testsuite name="Models">
    <directory>src/Model</directory>
    <file>src/Service/UserServiceTest.php</file>
    <exclude>src/Model/Cloud/ImagesTest.php</exclude>
  </testsuite>
</testsuites>
```

Créer des Tests pour les Plugins

Les Tests pour les plugins sont créés dans leur propre répertoire dans le dossier des plugins :

```
/src
/plugins
    /Blog
        /tests
            /TestCase
            /Fixture
```

Ils fonctionnent comme des tests normaux mais vous devrez vous souvenir d'utiliser les conventions de nommage pour les plugins quand vous importez des classes. Ceci est un exemple d'un cas de test pour le model `BlogPost` tiré du chapitre des plugins de ce manuel. Une différence par rapport aux autres tests se situe dans la première ligne, où on importe "Blog.BlogPost". Vous devrez aussi préfixer les fixtures de votre plugin avec `plugin.Blog.BlogPosts` :

```
namespace Blog\Test\TestCase\Model\Table;

use Blog\Model\Table\BlogPostsTable;
use Cake\TestSuite\TestCase;

class BlogPostsTableTest extends TestCase
{

    // Fixtures de plugin se trouvant dans /plugins/Blog/tests/Fixture/
    protected $fixtures = ['plugin.Blog.BlogPosts'];

    public function testSomething(): void
    {
        // Teste quelque chose.
    }
}
```

Si vous voulez utiliser des fixtures de plugin dans les tests de l'application, vous pouvez y faire référence en utilisant la syntaxe `plugin.pluginName.fixtureName` dans le tableau `$fixtures`.

Avant d'utiliser des fixtures, assurez-vous que le *listener de fixture* soit configuré dans votre fichier `phpunit.xml`. Vous devez également vous assurer que vos fixtures sont chargeables. Vérifiez que le code suivant est présent dans votre fichier **composer.json** :

```
"autoload-dev": {
    "psr-4": {
        "MonPlugin\\Test\\": "plugins/MonPlugin/tests"
    }
}
```

Note

N'oubliez pas de lancer `composer.phar dumpautoload` lorsque vous modifiez le mapping de l'autoloader.

Générer des Tests avec Bake

Si vous utilisez bake pour générer votre code, il va également générer le squelette de vos fichiers de tests. Si vous avez besoin de re-générer le squelette de vos fichiers de tests, ou si vous souhaitez générer le squelette de test pour du code que vous avez écrit, vous pouvez utiliser `bake` :

```
bin/cake bake test <type> <name>
```

`<type>` doit être une de ces options :

1. Entity
2. Table
3. Controller
4. Component
5. Behavior
6. Helper
7. Shell
8. Task
9. ShellHelper
10. Cell
11. Form
12. Mailer
13. Command

Où `<name>` est le nom de l'objet dont vous voulez générer le squelette de tests.

CHAPITRE 29

Validation

Le package de validation dans CakePHP fournit des fonctionnalités pour construire des validators qui peuvent valider des tableaux arbitraires de données avec simplicité. Vous pouvez trouver une liste des règles de validation dans l'API [179].

Créer les Validators

class Cake\Validation**Validator**

Les objets Validator définissent les valeurs qui s'appliquent à un ensemble de champs. Les objets Validator contiennent un mapping entre les champs et les ensembles de validation. A son tour l'ensemble de validation contient une collection de règles qui s'appliquent au champ auquel elles sont attachées. Créer un validator est simple :

```
use Cake\Validation\Validator;

$validator = new Validator();
```

Une fois créé, vous pouvez commencer à définir des ensembles de règle pour les champs que vous souhaitez valider :

```
$validator
    ->requirePresence('title')
    ->notEmptyString('title', 'Please fill this field')
    ->add('title', [
        'length' => [
            'rule' => ['minLength', 10],
            'message' => 'Titles need to be at least 10 characters long',
        ]
    ])
    ->allowEmptyString('published')
    ->add('published', 'boolean', [
```

(suite sur la page suivante)

179. https://api.cakephp.org/4.x/class-Cake.Validation.Validation.html

(suite de la page précédente)

```
        'rule' => 'boolean'
    ])
    ->requirePresence('body')
    ->add('body', 'length', [
        'rule' => ['minLength', 50],
        'message' => 'Articles must have a substantial body.'
    ]);
```

Comme vu dans l'exemple ci-dessus, les validators sont construits avec une interface facile qui vous permet de définir les règles pour chaque champ que vous souhaitez valider.

Il y a quelques méthodes appelées dans l'exemple ci-dessus, alors regardons les différentes fonctionnalités. La méthode `add()` vous permet d'ajouter les nouvelles règles au validator. Vous pouvez soit ajouter des règles individuellement, soit dans en groupe comme vu ci-dessus.

Valider la Présence d'un champ

La méthode `requirePresence()` oblige le champ à être présent dans tout tableau de validation. Si le champ est absent, la validation va échouer. La méthode `requirePresence()` a 4 modes :

- `true` La présence du champ est toujours requise.
- `false` La présence du champ n'est pas requise.
- `create` La présence du champ est requise lorsque vous validez une opération **create**.
- `update` La présence du champ est requise lorsque vous validez une opération **update**.

Par défaut `true` est utilisée. La présence de la clé est vérifiée pour l'utilisation de `array_key_exists()` donc les valeurs null vont être comptabilisées comme étant présentes. Vous pouvez définir le mode en utilisant le deuxième paramètre :

```
$validator->requirePresence('author_id', 'create');
```

Si vous avez plusieurs champs dont la présence est requise, vous pouvez les définir en liste :

```
// Définir plusieurs champs pour create
$validator->requirePresence(['author_id', 'title'], 'create');

// Définir plusieurs champs pour divers modes
$validator->requirePresence([
    'author_id' => [
        'mode' => 'create',
        'message' => "L'auteur est obligatoire.",
    ],
    'published' => [
        'mode' => 'update',
        'message' => 'Le statut de publication est obligatoire.',
    ]
]);
```

Permettre aux Champs d'être Vides

Les méthodes `allowEmpty()` et `notEmpty()` vous permettent de contrôler les champs autorisés à être "vide". En utilisant la méthode `notEmpty()`, le champ donné sera noté comme invalide quand il est vide. Vous pouvez utiliser `allowEmpty()` pour permettre à un champ d'être vide. Les deux méthodes `allowEmpty()` et `notEmpty()` ont un paramètre mode qui vous permet de contrôler quand un champ peut ou ne peut pas être vide :

- `true` Le champ peut être vide.

— `false` Le champ ne peut pas être vide.
— `create` La présence du champ est nécessaire mais il peut être vide lors de la validation d'une opération **create**.
— `update` La présence du champ est nécessaire mais il peut être vide lors de la validation d'une opération **update**.

Les valeurs `''`, `null` et `[]` (tableau vide) vont entraîner des erreurs de validation quand les champs n'ont pas l'autorisation d'être vide. Quand les champs ont l'autorisation d'être vide, les valeurs `''`, `null`, `false`, `[]`, `0`, `'0'` sont acceptées.

Un exemple de ces méthodes est le suivant :

```
$validator->allowEmpty('published')
    ->notEmpty('title', 'Le titre ne peut être vide')
    ->notEmpty('body', 'Le body ne peut être vide', 'create')
    ->allowEmpty('header_image', 'update');
```

Marquer les Règles comme étant les Dernières à être exécutées

Quand les champs ont plusieurs règles, chaque règle de validation sera exécutée même si la précédente a échouée. Cela vous permet de recueillir autant d'erreurs de validation que vous le pouvez en un seul passage. Si toutefois, vous voulez stopper l'exécution après qu'une règle spécifique a échoué, vous pouvez définir l'option `last` à `true` :

```
$validator = new Validator();
$validator
    ->add('body', [
        'minLength' => [
            'rule' => ['minLength', 10],
            'last' => true,
            'message' => 'Les commentaires doivent avoir un contenu un peu fourni.'
        ],
        'maxLength' => [
            'rule' => ['maxLength', 250],
            'message' => 'Les commentaires ne peuvent pas être trop longs.'
        ]
    ]);
```

Dans l'exemple ci-dessus, si la règle minLength (longueur minimale) échoue, la règle maxLength ne sera pas exécutée.

Méthodes de Validation Moins Verbeuses

Depuis la version 3.2, l'objet Validator accepte de nombreuses nouvelles méthodes qui rendent la construction de validateurs moins verbeux. Par exemple, ajouter des règles de validation à un champ username peut maintenant ressembler à ceci :

```
$validator = new Validator();
$validator
    ->email('username')
    ->ascii('username')
    ->lengthBetween('username', [4, 8]);
```

Ajouter des Providers de Validation

Les classes `Validator`, `ValidationSet` et `ValidationRule` ne fournissent elles-mêmes aucune méthode de validation. Les règles de validation viennent de "providers". Vous pouvez lier tout nombre de providers à un objet Validator. Les instances de Validator sont automatiquement fournies avec une configuration de provider à "default". Le provider par défaut est mappé à la classe `Validation`. Cela facilite l'utilisation des méthodes de cette classe en règles de validation. Lors de l'utilisation conjointe de Validators et de l'ORM, des providers supplémentaires sont configurés pour

la table et les objets entity. Vous pouvez utiliser la méthode `setProvider()` pour ajouter un provider supplémentaire que votre application a besoin d'utiliser :

```
$validator = new Validator();

// Utilise une instance de l'object.
$validator->setProvider('custom', $myObject);

// Utilise un nom de classe. Les méthodes doivent être static.
$validator->setProvider('custom', 'App\Model\Validation');
```

Les providers de Validation peuvent être des objets, ou des noms de classe. Si un nom de classe est utilisé, les méthodes doivent être static. Pour utiliser un provider autre que "default", assurez-vous de définir la clé `setProvider()` dans votre règle :

```
// Utilise une règle à partir du provider de la table
$validator->add('title', 'custom', [
    'rule' => 'customTableMethod',
    'provider' => 'table'
]);
```

Si vous souhaitez ajouter un `provider` à tous les objets `Validator` créés plus tard, vous pouvez utiliser la méthode `addDefaultProvider()` :

```
use Cake\Validation\Validator;

// En utilisant une instance d'objet.
Validator::addDefaultProvider('custom', $myObject);

// En utilisant un nom de classe. Les méthodes devront être static.
Validator::addDefaultProvider('custom', 'App\Model\Validation');
```

Note

Les DefaultProviders doivent être ajoutés avant que l'objet `Validator` ne soit créé. Par conséquent **config/bootstrap.php** est le meilleur endroit pour définir vos providers par défaut.

Vous pouvez utiliser le plugin Localized [180] pour fournir des providers basés sur les pays. Avec ce plugin, vous pourrez valider les champs de models selon un pays, par exemple :

```
namespace App\Model\Table;

use Cake\ORM\Table;
use Cake\Validation\Validator;

class PostsTable extends Table
{
    public function validationDefault(Validator $validator)
    {
        // Ajoute le provider au validator
        $validator->setProvider('fr', 'Localized\Validation\FrValidation');
```

(suite sur la page suivante)

180. https://github.com/cakephp/localized

(suite de la page précédente)

```
        // utilise le provider dans une règle de validation de champ
        $validator->add('phoneField', 'myCustomRuleNameForPhone', [
            'rule' => 'phone',
            'provider' => 'fr'
        ]);

        return $validator;
    }
}
```

Le plugin localized utilise le code ISO à 2 lettres des pays pour la validation, par exemple en, fr, de.

Il y a quelques méthodes qui sont communes à toutes les classes, définies par l'interface ValidationInterface [181] :

```
phone() pour vérifier un numéro de téléphone
postal() pour vérifier un code postal
personId() pour vérifier un ID d'une personne d'un pays
```

Règles de Validation Personnalisées

En plus de l'utilisation des méthodes venant des providers, vous pouvez aussi utiliser toute fonction appellable inclue de façon anonyme en règle de validation :

```
// Utilise une fonction globale
$validator->add('title', 'custom', [
    'rule' => 'validate_title',
    'message' => 'Le titre est invalide'
]);

// Utilise un tableau appelable qui n'est pas un provider
$validator->add('title', 'custom', [
    'rule' => [$this, 'method'],
    'message' => 'Le titre est invalide'
]);

// Utilise une closure
$extra = 'Des valeurs supplémentaires dont vous avez besoin dans la closure';
$validator->add('title', 'custom', [
    'rule' => function ($value, $context) use ($extra) {
        // Logique personnalisée qui retourne true/false
    },
    'message' => 'Le titre est invalide'
]);

// Utilisez une règle à partir d'un provider personnalisé
$validator->add('title', 'custom', [
    'rule' => 'customRule',
    'provider' => 'custom',
    'message' => 'Le titre n'est pas suffisamment unique'
]);
```

181. https://github.com/cakephp/localized/blob/master/src/Validation/ValidationInterface.php

Les Closures ou les méthodes appelables vont recevoir 2 arguments lors de leur appel. Le premier va être la valeur pour le champ étant validé. Le second est un tableau contextuel contenant des données liées au processus de validation :

- **data** : Les données originelles passées à la méthode de validation, utile si vous planifiez de créer les règles comparant les valeurs.
- **providers** : La liste complète de règle des objets provider, utile si vous avez besoin de créer des règles complexes en appelant plusieurs providers.
- **newRecord** : Selon si l'appel de la validation est pour un nouvel enregistrement ou pour un enregistrement existant.

Si vous devez passer des données supplémentaires à vos méthodes de validation comme pour les ids des users, vous pouvez utiliser un provider dynamique personnalisé dans votre controller :

```
$this->Examples->validator('default')->provider('passed', [
    'count' => $countFromController,
    'userid' => $this->Auth->user('id')
]);
```

Ensuite assurez-vous que votre méthode de validation ait le deuxième paramètre de contexte :

```
public function customValidationMethod($check, array $context)
{
    $userid = $context['providers']['passed']['userid'];
}
```

Validation Conditionnelle

Lors de la définition des règles de validation, vous pouvez utiliser la clé `on` pour définir quand une règle de validation doit être appliquée. Si elle est laissée non définie, la règle va toujours être appliquée. Les autres valeurs valides sont `create` et `update`. L'utilisation d'une de ces valeurs va faire que la règle va s'appliquer seulement pour les opérations create ou update.

En plus, vous pouvez fournir une fonction appelable qui va déterminer si oui ou non, une règle particulière doit être appliquée :

```
$validator->add('picture', 'file', [
    'rule' => ['mimeType', ['image/jpeg', 'image/png']],
    'on' => function ($context) {
        return !empty($context['data']['show_profile_picture']);
    }
]);
```

Vous pouvez accéder aux autres données soumises depuis le formulaire via le tableau `$context['data']`. L'exemple ci-dessus va rendre la règle pour "picture" optionnelle selon si la valeur pour `show_profile_picture` est vide. Vous pouvez également utiliser la règle de validation `uploadedFile` pour créer des inputs optionnelles d'upload de fichiers :

```
$validator->add('picture', 'file', [
    'rule' => ['uploadedFile', ['optional' => true]],
]);
```

Les méthodes de validation `allowEmpty()`, `notEmpty()` et `requirePresence()` prennent également une fonction appelable en dernier argument, ce qui determine si oui ou non la règle doit être appliquée. Par exemple on peut autoriser parfois à un champ à être vide :

```
$validator->allowEmpty('tax', function ($context) {
    return !$context['data']['is_taxable'];
});
```

De la même façon, on peut vouloir qu'un champ soit peuplé quand certaines conditions sont vérifiées :

```
$validator->notEmpty('email_frequency', 'This field is required', function ($context) {
    return !empty($context['data']['wants_newsletter']);
});
```

Dans l'exemple ci-dessus, le champ `email_frequency` ne peut être laissé vide si l'utilisateur veut recevoir la newsletter.

De plus il est aussi possible de demander à ce qu'un champ soit présent sous certaines conditions seulement :

```
$validator->requirePresence('full_name', function ($context) {
    if (isset($context['data']['action'])) {
        return $context['data']['action'] === 'subscribe';
    }

    return false;
});
$validator->requirePresence('email');
```

Ceci demanderait à ce que le champ `full_name` soit présent seulement dans le cas où l'utilisateur veut créer une inscription, alors que le champ `email` est toujours requis puisqu'il serait aussi demandée lors de l'annulation d'une inscription.

Imbriquer des Validators

Lorsque vous validez des *Formulaires Sans Models* avec des données imbriquées, ou lorsque vous travaillez avec des modèles qui contiennent des données de type tableau, il est nécessaire de valider les données imbriquées dont vous disposez. CakePHP permet d'ajouter des validators sur des attributs spécifiques. Par exemple, imaginez que vous travailliez avec une base de données non relationnelle et que vous ayez besoin d'enregistrer un article et ses commentaires :

```
$data = [
    'title' => 'Meilleur article',
    'comments' => [
        ['comment' => '']
    ]
];
```

Pour valider les commentaires, vous utiliseriez un validator imbriqué :

```
$validator = new Validator();
$validator->add('title', 'not-blank', ['rule' => 'notBlank']);

$commentValidator = new Validator();
$commentValidator->add('comment', 'not-blank', ['rule' => 'notBlank']);

// Connecte les validators imbriqués.
$validator->addNestedMany('comments', $commentValidator);

// Prior to 3.9 use $validator->errors()
// Récupère toutes erreurs y compris celles des validators imbriqués.
$validator->validate($data);
```

Vous pouvez créer des "relations" 1 :1 avec `addNested()` et des "relations" 1 :N avec `addNestedMany()`. Avec ces deux méthodes, les erreurs des validators contribuerons aux erreurs du validator parent et influeront sur le résultat final.

Créer des Validators Ré-utilisables

Bien que définir des validators inline, là où ils sont utilisés, permet de donner un bon exemple de code, cela ne conduit pas à avoir des applications facilement maintenable. A la place, vous devriez créer des sous-classes de `Validator` pour votre logique de validation réutilisable :

```
// Dans src/Model/Validation/ContactValidator.php
namespace App\Model\Validation;

use Cake\Validation\Validator;

class ContactValidator extends Validator
{
    public function __construct()
    {
        parent::__construct();
        // Add validation rules here.
    }
}
```

Valider les Données

Maintenant que vous avez créé un validator et que vous lui avez ajouté les règles que vous souhaitiez, vous pouvez commencer à l'utiliser pour valider les données. Les Validators sont capables de valider un tableau de données. Par exemple, si vous voulez valider un formulaire de contact avant de créer et d'envoyer un email, vous pouvez faire ce qui suit :

```
use Cake\Validation\Validator;

$validator = new Validator();
$validator
    ->requirePresence('email')
    ->add('email', 'validFormat', [
        'rule' => 'email',
        'message' => 'E-mail must be valid'
    ])
    ->requirePresence('name')
    ->allowEmpty('name', false, 'We need your name.')
    ->requirePresence('comment')
    ->allowEmpty('comment', false, 'You need to give a comment.');

// Prior to 3.9 use $validator->errors()
$errors = $validator->validate($this->request->getData());
if (empty($errors)) {
    // Envoi d'un email.
}
```

La méthode `errors()` va retourner un tableau non-vide quand il y a des échecs de validation. Le tableau retourné d'erreurs sera structuré comme ceci :

```
$errors = [
    'email' => ['E-mail doit être valide']
];
```

Si vous avez plusieurs erreurs pour un seul champ, un tableau de messages d'erreur va être retourné par champ. Par défaut la méthode `errors()` applique les règles pour le mode "create" mode. Si vous voulez appliquer les règles "update" vous pouvez faire ce qui suit :

```
// Prior to 3.9 use $validator->errors()
$errors = $validator->validate($this->request->getData(), false);
if (empty($errors)) {
    // Envoi d'un email.
}
```

Note

Si vous avez besoin de valider les entities, vous devez utiliser les méthodes comme `newEntity()`, `newEntities()`, `patchEntity()`, `patchEntities()` or *save()* puisqu'elles ont été créées pour cela.

Valider les Entities

Alors que les entities sont validées quand elles sont sauvegardées, vous pouvez aussi vouloir valider les entities avant d'essayer de faire toute sauvegarde. La validation des entities avant la sauvegarde est faite automatiquement quand on utilise `newEntity()`, `newEntities()`, `patchEntity()` ou `patchEntities()` :

```
// Dans la classe ArticlesController
$article = $this->Articles->newEntity($this->request->getData());
if ($article->errors()) {
    // Afficher les messages d'erreur ici.
}
```

De la même manière, quand vous avez besoin de pré-valider plusieurs entities en une fois, vous pouvez utiliser la méthode `newEntities()` :

```
// Dans la classe ArticlesController
$entities = $this->Articles->newEntities($this->request->getData());
foreach ($entities as $entity) {
    if (!$entity->errors()) {
            $this->Articles->save($entity);
    }
}
```

Les méthodes `newEntity()`, `patchEntity()`, `newEntities()` et `patchEntities()` vous permettent de spécifier les associations à valider, et les ensembles de validation à appliquer en utilisant le paramètre `options` :

```
$valid = $this->Articles->newEntity($article, [
  'associated' => [
    'Comments' => [
      'associated' => ['User'],
      'validate' => 'special',
    ]
  ]
]);
```

La validation est habituellement utilisée pour les formulaires ou les interfaces utilisateur, et ainsi elle n'est pas limitée seulement à la validation des colonnes dans le schéma de la table. Cependant maintenir l'intégrité des données

selon d'où elles viennent est important. Pour résoudre ce problème, CakePHP dispose d'un deuxième niveau de validation qui est appelé « règles d'application ». Vous pouvez en savoir plus en consultant la section *Appliquer les Règles d'Application*.

Règles de Validation du Cœur

CakePHP fournit une suite basique de méthodes de validation dans la classe `Validation`. La classe Validation contient un ensemble de méthodes static qui fournissent des validators pour plusieurs situations de validation habituelles.

La documentation de l'API [182] pour la classe `Validation` fournit une bonne liste de règles de validation qui sont disponibles, et leur utilisation basique.

Certaines des méthodes de validation acceptent des paramètres supplémentaires pour définir des conditions limites ou des options valides. Vous pouvez fournir ces conditions limite et options comme suit :

```
$validator = new Validator();
$validator
    ->add('title', 'minLength', [
        'rule' => ['minLength', 10]
    ])
    ->add('rating', 'validValue', [
        'rule' => ['range', 1, 5]
    ]);
```

Les règles du Cœur qui prennent des paramètres supplémentaires doivent avoir un tableau pour la clé `rule` qui contient la règle comme premier élément, et les paramètres supplémentaires en paramètres restants.

182. https://api.cakephp.org/4.x/class-Cake.Validation.Validation.html

CHAPITRE 30

Classe App

class Cake\Core**App**

La classe App est responsable de la localisation des ressources et de la gestion des chemins.

Trouver les Classes

static Cake\Core\App::**classname**(*$name*, *$type* = '', *$suffix* = '')

Cette méthode est utilisée pour trouver les noms de classe dans CakePHP. Elle retrouve les noms courts que CakePHP utilise et retourne le nom de classe entier :

```
// Retourne un nom de classe court avec le namespace + suffixe
App::classname('Auth', 'Controller/Component', 'Component');
// Retourne Cake\Controller\Component\AuthComponent

// Retourne un nom de plugin.
App::classname('DebugKit.Toolbar', 'Controller/Component', 'Component');
// Retourne DebugKit\Controller\Component\ToolbarComponent

// Noms contenant \ seront retournés non modifiés.
App::classname('App\Cache\ComboCache');
// Retourne App\Cache\ComboCache
```

Quand vous retrouvez les classes, le namespace `App` sera essayé, et si la classe n'existe pas, le namespace `Cake` sera tenté. Si les deux noms de classe n'existent pas, `false` sera retourné.

Trouver les Chemins vers les Namespaces

static Cake\Core\App::**path**(*string $package*, *string $plugin = null*)

Utilisée pour obtenir les localisations pour les chemins basés sur les conventions :

```
// Obtenir le chemin vers Controller/ dans votre application
App::path('Controller');
```

Ceci peut être fait pour tous les namespaces qui font parti de votre application. Vous pouvez aussi récupérer les chemins pour un plugin :

```
// retourne les chemins de component dans DebugKit
App::path('Component', 'DebugKit');
```

`App::path()` va seulement retourner le chemin par défaut, et ne sera pas capable de fournir toutes les informations sur les chemins supplémentaires pour lesquels l'autoloader est configuré.

static Cake\Core\App::**core**(*string $package*)

Utilisée pour trouver le chemin vers un package dans CakePHP :

```
// Obtenir le chemin des moteurs de Cache.
App::core('Cache/Engine');
```

Localiser les Plugins

Les plugins peuvent être localisés avec Plugin. En utilisant `Plugin::path('DebugKit');` par exemple, cela vous donnera le chemin complet vers le plugin DebugKit :

```
$path = Plugin::path('DebugKit');
```

Localiser les Themes

Puisque les themes sont les plugins, vous pouvez utiliser les méthodes ci-dessus pour récupérer le chemin vers un theme.

Charger les Fichiers de Vendor

Idéalement les fichiers de vendor devront être auto-chargés avec `Composer`, si vous avez des fichiers de vendor qui ne peuvent pas être auto-chargés ou installés avec Composer, vous devrez utiliser `require` pour les charger.

Si vous ne pouvez pas installer une librairie avec Composer, il est mieux d'installer chaque librairie dans un répertoire en suivant les conventions de Composer de `vendor/$author/$package`. Si vous avez une librairie appelée AcmeLib, vous pouvez l'installer dans `vendor/Acme/AcmeLib`. En supposant qu'il n'utilise pas des noms de classe compatible avec PSR-0, vous pouvez auto-charger les classes qu'il contient en utilisant `classmap` dans le `composer.json` de votre application :

```
"autoload": {
    "psr-4": {
        "App\\": "App",
        "App\\Test\\": "Test",
```

(suite sur la page suivante)

(suite de la page précédente)

```
        "": "./Plugin"
    },
    "classmap": [
        "vendor/Acme/AcmeLib"
    ]
}
```

Si votre librairie de vendor n'utilise pas de classes, et fournit plutôt des fonctions, vous pouvez configurer Composer pour charger ces fichiers au début de chaque requête en utilisant la stratégie d'auto-chargement `files` :

```
"autoload": {
    "psr-4": {
        "App\\": "App",
        "App\\Test\\": "Test",
        "": "./Plugin"
    },
    "files": [
        "vendor/Acme/AcmeLib/functions.php"
    ]
}
```

Après avoir configuré les librairies de vendor, vous devrez regénérer l'autoloader de votre application en utilisant :

```
$ php composer.phar dump-autoload
```

Si vous n'utilisez pas Composer dans votre application, vous devrez manuellement charger toutes les librairies de vendor vous-même.

CHAPITRE 31

Collections

class Cake\Collection**Collection**

Les classes collection fournissent un ensemble d'outils pour manipuler les tableaux ou les objets `Traversable`. Si vous avez déjà utilisé underscore.js, vous avez une idée de ce que vous pouvez attendre des classes collection.

Les instances Collection sont immutables, modifier une collection va plutôt générer une nouvelle collection. Cela rend le travail avec les objets collection plus prévisible puisque les opérations sont sans effets collatéraux.

Exemple Rapide

Les Collections peuvent être créées en utilisant un tableau ou un objet Traversable. Vous allez aussi interagir avec les collections à chaque fois que vous faites une interaction avec l'ORM de CakePHP. Une utilisation simple de Collection serait :

```
use Cake\Collection\Collection;

$items = ['a' => 1, 'b' => 2, 'c' => 3];
$collection = new Collection($items);

// Crée une nouvelle collection contenant des éléments
// avec une valeur supérieure à un.
$overOne = $collection->filter(function ($value, $key, $iterator) {
    return $value > 1;
});
```

Vous pouvez aussi utiliser la fonction `collection()` à la place de `new Collection()` :

```
$items = ['a' => 1, 'b' => 2, 'c' => 3];

// Les deux créent une instance de Collection.
```

(suite sur la page suivante)

(suite de la page précédente)

```
$collectionA = new Collection($items);
$collectionB = collection($items);
```

Le bénéfice de cette méthode est qu'il est plus facile de chaîner par rapport à `(new Collection($items))`.

`CollectionTrait` vous permet également d'intégrer des fonctionnalités semblables aux Collections pour tout objet `Traversable` de votre application.

Liste des Méthodes

append	*avg*	*buffered*	*chunk*
chunkWithKeys	*combine*	*compile*	*contains*
countBy	*each*	*every*	*extract*
filter	*first*	*groupBy*	*indexBy*
insert	*isEmpty*	*last*	*listNested*
map	*match*	*max*	*median*
min	*nest*	*reduce*	*reject*
sample	*shuffle*	*skip*	*some*
sortBy	*stopWhen*	*sumOf*	*take*
through	*transpose*	*unfold*	*zip*

Faire une Itération

`Cake\Collection\Collection::`**`each`**`(`*$callback*`)`

Les Collections peuvent être itérées et/ou transformées en nouvelles collections avec les méthodes `each()` et `map()`. La méthode `each()` ne va pas créer une nouvelle collection, mais va vous permettre de modifier tout objet dans la collection :

```
$collection = new Collection($items);
$collection = $collection->each(function ($value, $key) {
    echo "Element $key: $value";
});
```

Le retour de `each()` sera un objet collection. Each va itérer la collection en appliquant immédiatement le callback pour chaque valeur de la collection.

`Cake\Collection\Collection::`**`map`**`(`*$callback*`)`

La méthode `map()` va créer une nouvelle collection basée sur la sortie du callback étant appliqué à chaque objet dans la collection originelle :

```
$items = ['a' => 1, 'b' => 2, 'c' => 3];
$collection = new Collection($items);

$new = $collection->map(function ($value, $key) {
    return $value * 2;
});

// $result contient ['a' => 2, 'b' => 4, 'c' => 6];
$result = $new->toArray();
```

La méthode `map()` va créer un nouvel itérateur, qui va créer automatiquement les objets résultants quand ils sont itérés.

`Cake\Collection\Collection::extract($path)`

Une des utilisations les plus courantes de la fonction `map()` est l'extraction d'une colonne unique d'une collection. Si vous souhaitez construire une liste d'éléments contenant les valeurs pour une propriété en particulier, vous pouvez utiliser la méthode `extract()` :

```
$collection = new Collection($people);
$names = $collection->extract('name');

// $result contient ['mark', 'jose', 'barbara'];
$result = $names->toArray();
```

Comme plusieurs autres fonctions dans la classe `Collection`, vous pouvez spécifier un chemin séparé de points pour extraire les colonnes. Cet exemple va retourner une collection contenant les noms d'auteurs à partir d'une liste d'articles :

```
$collection = new Collection($articles);
$names = $collection->extract('author.name');

// $result contient ['Maria', 'Stacy', 'Larry'];
$result = $names->toArray();
```

Finalement, si la propriété que vous recherchez ne peut être exprimée en chemin, vous pouvez utiliser une fonction de callback pour la retourner :

```
$collection = new Collection($articles);
$names = $collection->extract(function ($article) {
    return $article->author->name . ', ' . $article->author->last_name;
});
```

Vous aurez souvent besoin d'extraire une clé commune présente dans plusieurs tableaux ou objets qui sont imbriqués profondément dans d'autres structures. Dans ces cas-là, vous pouvez utilisez le matcher `{*}` dans la clé du chemin. Ce matcher est souvent utile quand vous faîtes correspondre des données d'association HasMany et BelongsToMany :

```
$data = [
    [
        'name' => 'James',
        'phone_numbers' => [
            ['number' => 'number-1'],
            ['number' => 'number-2'],
            ['number' => 'number-3'],
        ]
    ],
    [
        'name' => 'James',
        'phone_numbers' => [
            ['number' => 'number-4'],
            ['number' => 'number-5'],
        ]
    ]
];
```

(suite sur la page suivante)

(suite de la page précédente)

```
$numbers = (new Collection($data))->extract('phone_numbers.{*}.number');
$numbers->toList();
// Retourne ['number-1', 'number-2', 'number-3', 'number-4', 'number-5']
```

Ce dernier exemple utilise `toList()` au contraire des autres exemples, ce qui est important quand vous récupérez des résultats avec de possibles clés dupliquées. En utilisant `toList()`, nous aurons la garantie de récupérer toutes les valeurs même s'il y a des clés dupliquées.

`Cake\Collection\Collection::combine`(*$keyPath*, *$valuePath*, *$groupPath = null*)

Les collections vous permettent de créer une nouvelle collection à partir des clés et des valeurs d'une collection existante. Les chemins de clé et de valeur peuvent être spécifiés avec la notation par point des chemins :

```
$items = [
    ['id' => 1, 'name' => 'foo', 'parent' => 'a'],
    ['id' => 2, 'name' => 'bar', 'parent' => 'b'],
    ['id' => 3, 'name' => 'baz', 'parent' => 'a'],
];
$combined = (new Collection($items))->combine('id', 'name');

// Le résultat ressemble à ceci quand il est converti en tableau
[
    1 => 'foo',
    2 => 'bar',
    3 => 'baz',
];
```

Vous pouvez aussi utiliser `groupPath` en option pour grouper les résultats basés sur un chemin :

```
$combined = (new Collection($items))->combine('id', 'name', 'parent');

// Le résultat ressemble à ceci quand il est converti en tableau
[
    'a' => [1 => 'foo', 3 => 'baz'],
    'b' => [2 => 'bar']
];
```

Finalement vous pouvez utiliser les *closures* pour construire les chemins des clés/valeurs/groupes de façon dynamique, par exemple quand vous travaillez avec les entities et les dates (convertis en instances `Cake/Time` par l'ORM) vous pourriez grouper les résultats par date :

```
$combined = (new Collection($entities))->combine(
    'id',
    function ($entity) { return $entity; },
    function ($entity) { return $entity->date->toDateString(); }
);

// Le résultat va ressembler à ceci quand il sera converti en tableau
[
    'date string like 2015-05-01' => ['entity1->id' => entity1, 'entity2->id' => entity2,
↪ ..., 'entityN->id' => entityN]
    'date string like 2015-06-01' => ['entity1->id' => entity1, 'entity2->id' => entity2,
↪ ..., 'entityN->id' => entityN]
]
```

`Cake\Collection\Collection::stopWhen(callable $c)`

Vous pouvez stopper l'itération à n'importe quel point en utilisant la méthode `stopWhen()`. L'appeler dans une collection va en créer une qui va stopper le retour des résultats si le callable passé retourne false pour l'un des éléments :

```
$items = [10, 20, 50, 1, 2];
$collection = new Collection($items);

$new = $collection->stopWhen(function ($value, $key) {
    // Stop on the first value bigger than 30
    return $value > 30;
});

// $result contient [10, 20];
$result = $new->toArray();
```

`Cake\Collection\Collection::unfold(callable $callback)`

Parfois les items internes d'une collection vont contenir des tableaux ou des itérateurs avec plus d'items. Si vous souhaitez aplatir la structure interne pour itérer une fois tous les éléments, vous pouvez utiliser la méthode `unfold()`. Cela va créer une nouvelle collection qui va produire l'élément unique imbriqué dans la collection :

```
$items = [[1, 2, 3], [4, 5]];
$collection = new Collection($items);
$new = $collection->unfold();

// $result contient [1, 2, 3, 4, 5];
$result = $new->toList();
```

Quand vous passez un callable à `unfold()`, vous pouvez contrôler les éléments qui vont être révélés à partir de chaque item dans la collection originale. C'est utile pour retourner les données à partir des services paginés :

```
$pages = [1, 2, 3, 4];
$collection = new Collection($pages);
$items = $collection->unfold(function ($page, $key) {
    // Un service web imaginaire qui retourne une page de résultats
    return MyService::fetchPage($page)->toArray();
});

$allPagesItems = $items->toList();
```

Si vous utilisez PHP 5.5+, vous pouvez utiliser le mot clé `yield` à l'intérieur de `unfold()` pour renvoyer autant d'éléments pour chaque item dans la collection que besoin :

```
$oddNumbers = [1, 3, 5, 7];
$collection = new Collection($oddNumbers);
$new = $collection->unfold(function ($oddNumber) {
    yield $oddNumber;
    yield $oddNumber + 1;
});

// $result contient [1, 2, 3, 4, 5, 6, 7, 8];
$result = $new->toList();
```

`Cake\Collection\Collection::chunk($chunkSize)`

Quand vous gérez des grandes quantités d'items dans une collection, il peut paraître sensé d'agir sur les éléments en lots plutôt qu'un par un. Pour séparer une collection en plusieurs tableaux d'une certaine taille, vous pouvez utiliser la fonction `chunk()` :

```
$items = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11];
$collection = new Collection($items);
$chunked = $collection->chunk(2);
$chunked->toList(); // [[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11]]
```

La fonction `chunk` est particulièrement utile quand vous faîtes des opérations en lots, par exemple avec les résultats d'une base de données :

```
$collection = new Collection($articles);
$collection->map(function ($article) {
        // Change une propriété de l'article
        $article->property = 'changed';
    })
    ->chunk(20)
    ->each(function ($batch) {
        myBulkSave($batch); // Cette fonction sera appelée pour chaque lot
    });
```

`Cake\Collection\Collection::chunkWithKeys(`*$chunkSize*`)`

Tout comme *chunk()*, `chunkWithKeys()` vous permet de découper une collection en plusieurs tableaux plus petits mais en préservant les clés. Ceci est particulièrement utile quand vous avez besoin de découper des tableaux associatifs :

```
$collection = new Collection([
    'a' => 1,
    'b' => 2,
    'c' => 3,
    'd' => [4, 5]
]);
$chunked = $collection->chunkWithKeys(2)->toList();
// Va créer
[
    ['a' => 1, 'b' => 2],
    ['c' => 3, 'd' => [4, 5]]
]
```

Filtrer

`Cake\Collection\Collection::filter(`*$callback*`)`

Les collections permettent de filtrer et de créer les nouvelles collections basées sur le résultat de fonctions callback. Vous pouvez utiliser `filter()` pour créer une nouvelle collection d'éléments qui matchent un critère callback :

```
$collection = new Collection($people);
$ladies = $collection->filter(function ($person, $key) {
    return $person->gender === 'female';
});
$guys = $collection->filter(function ($person, $key) {
    return $person->gender === 'male';
});
```

`Cake\Collection\Collection::reject(callable $c)`

L'inverse de `filter()` est `reject()`. Cette méthode fait un filtre négatif, retirant les éléments qui matchent la fonction filter :

```
$collection = new Collection($people);
$ladies = $collection->reject(function ($person, $key) {
    return $person->gender === 'male';
});
```

`Cake\Collection\Collection::every($callback)`

Vous pouvez faire des tests de vérité avec les fonctions filter. Pour voir si chaque élément dans une collection matche un test, vous pouvez utiliser `every()` :

```
$collection = new Collection($people);
$allYoungPeople = $collection->every(function ($person) {
    return $person->age < 21;
});
```

`Cake\Collection\Collection::some($callback)`

Vous pouvez regarder si la collection contient au moins un élément matchant une fonction filter en utilisant la méthode `some()` :

```
$collection = new Collection($people);
$hasYoungPeople = $collection->some(function ($person) {
    return $person->age < 21;
});
```

`Cake\Collection\Collection::match($conditions)`

Si vous avez besoin d'extraire une nouvelle collection contenant seulement les éléments qui contiennent un ensemble donné de propriétés, vous devez utiliser la méthode `match()` :

```
$collection = new Collection($comments);
$commentsFromMark = $collection->match(['user.name' => 'Mark']);
```

`Cake\Collection\Collection::firstMatch($conditions)`

Le nom de la propriété peut être un chemin séparé par des points. Vous pouvez traverser des entities imbriquées et matcher les valeurs qu'elles contiennent. Quand vous avez besoin de seulement matcher le premier élément d'une collection, vous pouvez utiliser `firstMatch()` :

```
$collection = new Collection($comments);
$comment = $collection->firstMatch([
    'user.name' => 'Mark',
    'active' => true
]);
```

Comme vous pouvez le voir ci-dessus, les méthodes `match()` et `firstMatch()` vous permettent de fournir plusieurs conditions à matcher. De plus, les conditions peuvent être utilisées sur des chemins différents, vous permettant d'exprimer des conditions complexes à faire correspondre.

Agrégation

`Cake\Collection\Collection::reduce($callback)`

La contrepartie de l'opération `map()` est habituellement un `reduce`. Cette fonction va vous aider à construire un résultat unique à partir de tous les éléments d'une collection :

```
$totalPrice = $collection->reduce(function ($accumulated, $orderLine, $index) {
    return $accumulated + $orderLine->price;
}, 0);
```

Dans l'exemple ci-dessus, `$totalPrice` va être la somme de tous les prix uniques qui se trouvent dans la collection. Remarquez le deuxième argument pour la fonction `reduce()`, il prend la valeur initiale pour l'opération `reduce` que vous souhaitez faire :

```
$allTags = $collection->reduce(function ($accumulated, $article, $index) {
    return array_merge($accumulated, $article->tags);
}, []);
```

`Cake\Collection\Collection::min(string|$callback, $type = SORT_NUMERIC)`

Pour extraire la valeur minimum pour une collection basée sur une propriété, utilisez juste la fonction `min()`. Celle-ci va retourner l'élément complet à partir de la collection et pas seulement la plus petite valeur trouvée :

```
$collection = new Collection($people);
$youngest = $collection->min('age');

echo $youngest->name;
```

Vous pouvez aussi exprimer la propriété à comparer en fournissant un chemin ou une fonction callback :

```
$collection = new Collection($people);
$personYoungestChild = $collection->min(function ($person) {
    return $person->child->age;
});

$personWithYoungestDad = $collection->min('dad.age');
```

`Cake\Collection\Collection::max($callback, $type = SORT_NUMERIC)`

La même chose peut être appliquée à la fonction `max()`, qui retourne un élément unique à partir de la collection ayant la valeur de propriété la plus élevée :

```
$collection = new Collection($people);
$oldest = $collection->max('age');

$personOldestChild = $collection->max(function ($person) {
    return $person->child->age;
});

$personWithOldestDad = $collection->max('dad.age');
```

`Cake\Collection\Collection::sumOf($path = null)`

Pour finir, la méthode `sumOf()` va retourner la somme d'une propriété de tous les éléments :

```
$collection = new Collection($people);
$sumOfAges =  $collection->sumOf('age');

$sumOfChildrenAges = $collection->sumOf(function ($person) {
    return $person->child->age;
});

$sumOfDadAges = $collection->sumOf('dad.age');
```

`Cake\Collection\Collection::avg($path = null)`

Calcule la moyenne des éléments de la collection. Vous pouvez passer, en option, un « path » à matcher ou une fonction pour extraire les valeurs pour lesquelles vous souhaitez générer la moyenne :

```
$items = [
   ['invoice' => ['total' => 100]],
   ['invoice' => ['total' => 200]],
];

// Moyenne : 150
$average = (new Collection($items))->avg('invoice.total');
```

`Cake\Collection\Collection::median($path = null)`

Calcule la valeur médianne d'un jeu d'élément. Vous pouvez passer, en option, un « path » à matcher ou une fonction pour extraire les valeurs pour lesquelles vous souhaitez calculer la valeur médianne :

```
$items = [
  ['invoice' => ['total' => 400]],
  ['invoice' => ['total' => 500]],
  ['invoice' => ['total' => 100]],
  ['invoice' => ['total' => 333]],
  ['invoice' => ['total' => 200]],
];

// Valeur médiane : 333
$median = (new Collection($items))->median('invoice.total');
```

Grouper et Compter

`Cake\Collection\Collection::groupBy($callback)`

Les valeurs d'une collection peuvent être groupées avec des clés différentes dans une nouvelle collection quand elles partagent la même valeur pour une propriété :

```
$students = [
    ['name' => 'Mark', 'grade' => 9],
    ['name' => 'Andrew', 'grade' => 10],
    ['name' => 'Stacy', 'grade' => 10],
    ['name' => 'Barbara', 'grade' => 9]
];
$collection = new Collection($students);
$studentsByGrade = $collection->groupBy('grade');
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Le résultat ressemble à ceci quand il est converti en tableau:
[
  10 => [
    ['name' => 'Andrew', 'grade' => 10],
    ['name' => 'Stacy', 'grade' => 10]
  ],
  9 => [
    ['name' => 'Mark', 'grade' => 9],
    ['name' => 'Barbara', 'grade' => 9]
  ]
]
```

Comme d'habitude, il est possible de fournir soit un chemin séparé de points pour les propriétés imbriquées ou votre propre fonction de callback pour générer les groupes dynamiquement :

```
$commentsByUserId = $comments->groupBy('user.id');

$classResults = $students->groupBy(function ($student) {
    return $student->grade > 6 ? 'approved' : 'denied';
});
```

`Cake\Collection\Collection::countBy($callback)`

Si vous souhaitez seulement connaître le nombre d'occurrences par groupe, vous pouvez le faire en utilisant la méthode `countBy()`. Elle prend les mêmes arguments que `groupBy` donc cela devrait vous être déjà familier :

```
$classResults = $students->countBy(function ($student) {
    return $student->grade > 6 ? 'approved' : 'denied';
});
```

Result could look like this when converted to array :

["approved" => 70, "denied" => 20]

`Cake\Collection\Collection::indexBy($callback)`

Il y aura des cas où vous savez qu'un élément est unique pour la propriété selon laquelle vous souhaitez faire un `groupBy()`. Si vous souhaitez un unique résultat par groupe, vous pouvez utiliser la fonction `indexBy()` :

```
$usersById = $users->indexBy('id');

// Quand il est converti en tableau, le résultat pourrait ressembler à ceci
[
    1 => 'markstory',
    3 => 'jose_zap',
    4 => 'jrbasso'
]
```

Comme avec la fonction `groupBy()`, vous pouvez aussi utiliser un chemin de propriété ou un callback :

```
$articlesByAuthorId = $articles->indexBy('author.id');

$filesByHash = $files->indexBy(function ($file) {
    return md5($file);
});
```

`Cake\Collection\Collection::zip($items)`

Les éléments de différentes collections peuvent être groupés ensemble en utilisant la méthode `zip()`. Elle retournera une nouvelle collection contenant un tableau regroupant les éléments de chaque collection qui sont placés à la même position :

```
$odds = new Collection([1, 3, 5]);
$pairs = new Collection([2, 4, 6]);
$combined = $odds->zip($pairs)->toList(); // [[1, 2], [3, 4], [5, 6]]
```

Vous pouvez également zipper des cllections multiples d'un coup :

```
$years = new Collection([2013, 2014, 2015, 2016]);
$salaries = [1000, 1500, 2000, 2300];
$increments = [0, 500, 500, 300];

$rows = $years->zip($salaries, $increments)->toList();
// Retourne:
[
    [2013, 1000, 0],
    [2014, 1500, 500],
    [2015, 2000, 500],
    [2016, 2300, 300]
]
```

Comme vous avez pu le voir, la méthode `zip()` est très utile pour transposer des tableaux multidimensionnels :

```
$data = [
    2014 => ['jan' => 100, 'feb' => 200],
    2015 => ['jan' => 300, 'feb' => 500],
    2016 => ['jan' => 400, 'feb' => 600],
]

// Récupérer les données de jan et fev ensemble

$firstYear = new Collection(array_shift($data));
$firstYear->zip($data[0], $data[1])->toList();

// Ou $firstYear->zip(...$data) in PHP >= 5.6

// Retourne
[
    [100, 300, 400],
    [200, 500, 600]
]
```

Trier

`Cake\Collection\Collection::sortBy($callback, $order = SORT_DESC, $sort = SORT_NUMERIC)`

Les valeurs de collection peuvent être triées par ordre croissant ou décroissant basé sur une colonne ou une fonction personnalisée. Pour créer une nouvelle collection triée à partir de valeurs d'une autre, vous pouvez utiliser `sortBy` :

```
$collection = new Collection($people);
$sorted = $collection->sortBy('age');
```

Comme vu ci-dessus, vous pouvez trier en passant le nom d'une colonne ou d'une propriété qui est présente dans les valeurs de la collection. Vous pouvez aussi spécifier un chemin de propriété à la place de la notation par point. L'exemple suivant va trier les articles par leur nom d'auteur :

```
$collection = new Collection($articles);
$sorted = $collection->sortBy('author.name');
```

La méthode `sortBy()` est assez flexible pour vous laisser spécifier une fonction d'extracteur qui vous laisse sélectionner dynamiquement la valeur à utiliser pour comparer deux valeurs différentes dans la collection :

```
$collection = new Collection($articles);
$sorted = $collection->sortBy(function ($article) {
    return $article->author->name . '-' . $article->title;
});
```

Afin de spécifier la direction dans laquelle la collection doit être triée, vous devez fournir soit `SORT_ASC` soit `SORT_DESC` en deuxième paramètre pour trier respectivement par ordre croissant ou décroissant. Par défaut, les collections sont triées par ordre décroissant :

```
$collection = new Collection($people);
$sorted = $collection->sortBy('age', SORT_ASC);
```

Parfois vous devez spécifier le type de données que vous essayez de comparer pour avoir des résultats cohérents. A cet effet, vous devez fournir un troisième argument dans la fonction `sortBy()` avec une des constantes suivantes :

- **SORT_NUMERIC** : Pour comparer les nombres
- **SORT_STRING** : Pour comparer les valeurs de chaîne
- **SORT_NATURAL** : Pour trier une chaîne contenant des nombres que vous souhaitez trier de façon naturelle. Par exemple, afficher « 10 » après « 2 ».
- **SORT_LOCALE_STRING** : Pour comparer les chaînes basées sur la locale courante.

Par défaut, `SORT_NUMERIC` est utilisée :

```
$collection = new Collection($articles);
$sorted = $collection->sortBy('title', SORT_ASC, SORT_NATURAL);
```

> **Avertissement**
>
> Il est souvent coûteux d'itérer les collections triées plus d'une fois. Si vous voulez le faire, pensez à convertir la collection en tableau ou utilisez simplement la méthode `compile()` dessus.

Utiliser des Données en Arbre

`Cake\Collection\Collection::nest(`*$idPath*, *$parentPath*`)`

Toutes les données ne sont pas destinées à être représentées de façon linéaire. Les collections facilitent la construction et l'aplatissement de structures hiérarchiques ou imbriquées. Avec la méthode `nest()`, il devient possible de créer une structure imbriquée où les enfants sont groupés selon une propriété identifier parente.

Deux paramètres sont requis pour cette fonction. La première est la propriété représentant l'identifier de l'item. Le second paramètre est le nom de la propriété représentant l'identifier pour l'item parent :

```
$items new Collection([
    ['id' => 1, 'parent_id' => null, 'name' => 'Birds'],
    ['id' => 2, 'parent_id' => 1, 'name' => 'Land Birds'],
    ['id' => 3, 'parent_id' => 1, 'name' => 'Eagle'],
    ['id' => 4, 'parent_id' => 1, 'name' => 'Seagull'],
    ['id' => 5, 'parent_id' => 6, 'name' => 'Clown Fish'],
    ['id' => 6, 'parent_id' => null], 'name' => 'Fish'],
]);

$collection->nest('id', 'parent_id')->toArray();
// Retourne
[
    [
        'id' => 1,
        'parent_id' => null,
        'name' => 'Bird',
        'children' => [
            [
                'id' => 2,
                'parent_id' => 1,
                'name' => 'Land Birds',
                'children' => [
                    ['id' => 3, 'name' => 'Eagle', 'parent_id' => 2]
                ]
            ],
            ['id' => 4, 'parent_id' => 1, 'name' => 'Seagull',  'children' => []],
        ]
    ],
    [
        'id' => 6,
        'parent_id' => null,
        'name' => 'Fish',
        'children' => [
            ['id' => 5, 'parent_id' => 6, 'name' => 'Clown Fish', 'children' => []],
        ]
    ]
];
```

Les éléments enfants sont imbriqués dans la propriété `children` à l'intérieur de chacun des items dans la collection. Cette représentation de type de données aide à rendre les menus ou à traverser les éléments vers le haut à un certain niveau dans l'arbre.

`Cake\Collection\Collection::listNested`(*$sort = 'desc'*, *$nestingKey = 'children'*)

L'inverse de `nest()` est `listNested()`. Cette méthode vous permet d'aplatir une structure en arbre en structure linéaire. Elle prend deux paramètres, le premier est le mode de traversement (asc, desc ou leaves), et le deuxième est le nom de la propriété contenant les enfants pour chaque élément dans la collection.

Considérons la collection imbriquée intégrée dans l'exemple précédent, nous pouvons l'aplatir :

```
$nested->listNested()->toList();

// Retourne
[
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ['id' => 1, 'parent_id' => null, 'name' => 'Birds', 'children' => [...]],
    ['id' => 2, 'parent_id' => 1, 'name' => 'Land Birds'],
    ['id' => 3, 'parent_id' => 1, 'name' => 'Eagle'],
    ['id' => 4, 'parent_id' => 1, 'name' => 'Seagull'],
    ['id' => 6, 'parent_id' => null, 'name' => 'Fish', 'children' => [...]],
    ['id' => 5, 'parent_id' => 6, 'name' => 'Clown Fish']
]
```

Par défaut, l'arbre est traversé de la racine vers les feuilles. Vous pouvez également demander à retourner seulement les éléments feuilles de l'arbre :

```
$nested->listNested()->toArray();

// Retourne
[
    ['id' => 3, 'parent_id' => 1, 'name' => 'Eagle'],
    ['id' => 4, 'parent_id' => 1, 'name' => 'Seagull'],
    ['id' => 5, 'parent_id' => 6, 'name' => 'Clown Fish']
]
```

Once you have converted a tree into a nested list, you can use the `printer()` method to configure how the list output should be formatted :

```
$nested->listNested()->printer('name', 'id', '--')->toArray();

// Returns
[
    3 => 'Eagle',
    4 => 'Seagull',
    5 -> '--Clown Fish',
]
```

The `printer()` method also lets you use a callback to generate the keys and or values :

```
$nested->listNested()->printer(
    function ($el) {
        return $el->name;
    },
    function ($el) {
        return $el->id;
    }
);
```

Autres Méthodes

`Cake\Collection\Collection::isEmpty()`

Vous permet de voir si une collection contient un élément :

```
$collection = new Collection([]);
// Returns true
$collection->isEmpty();
```

(suite sur la page suivante)

(suite de la page précédente)

```
$collection = new Collection([1]);
// Returns false
$collection->isEmpty();
```

`Cake\Collection\Collection::contains($value)`

Les collections vous permettent de vérifier rapidement si elles contiennent une valeur particulière : en utilisant la méthode `contains()` :

```
$items = ['a' => 1, 'b' => 2, 'c' => 3];
$collection = new Collection($items);
$hasThree = $collection->contains(3);
```

Les comparaisons sont effectuées en utilisant l'opérateur ===. Si vous souhaitez faire des types de comparaison non stricte, vous pouvez utiliser la méthode `some()`.

`Cake\Collection\Collection::shuffle()`

Parfois vous pouvez souhaiter montrer une collection de valeurs dans un ordre au hasard. Afin de créer une nouvelle collection qui va retourner chaque valeur dans une position au hasard, utilisez `shuffle` :

```
$collection = new Collection(['a' => 1, 'b' => 2, 'c' => 3]);

// Ceci pourrait retourner [2, 3, 1]
$collection->shuffle()->toArray();
```

`Cake\Collection\Collection::transpose()`

Quand vous transposez une collection, vous récupérez une nouvelle collection contenant une colonne avec chacune des colonnes originales :

```
$items = [
   ['Products', '2012', '2013', '2014'],
   ['Product A', '200', '100', '50'],
   ['Product B', '300', '200', '100'],
   ['Product C', '400', '300', '200'],
]
$transpose = (new Collection($items))->transpose()->toList();

// Returns
[
    ['Products', 'Product A', 'Product B', 'Product C'],
    ['2012', '200', '300', '400'],
    ['2013', '100', '200', '300'],
    ['2014', '50', '100', '200'],
]
```

Retrait d'Eléments

`Cake\Collection\Collection::sample($length = 10)`

Remanier une collection est souvent utile quand vous faites des statistiques d'analyse rapides. Une autre opération habituelle quand vous faites ce type de tâches est d'extraire quelques valeurs au hasard en dehors de la collection pour

que plus de tests puissent être effectués dessus. Par exemple, si vous souhaitez sélectionner 5 utilisateurs au hasard auxquels vous voulez appliquer des tests A/B, vous pouvez utiliser la fonction `sample()` :

```
$collection = new Collection($people);

// Extrait au maximum 20 utilisateurs au hasard de la collection
$testSubjects = $collection->sample(20);
```

`sample()` va prendre au moins le nombre de valeurs que vous spécifiez dans le premier argument. S'il n'y a pas assez d'éléments dans la collection qui satisfont le sample, la collection sera retournée en entier dans un ordre au hasard.

`Cake\Collection\Collection::take(`*$length*, *$offset*`)`

Quand vous souhaitez prendre une partie d'une collection, utilisez la fonction `take()`, cela va créer une nouvelle collection avec au moins le nombre de valeurs que vous spécifiez dans le premier argument, en commençant par la position passée dans le second argument :

```
$topFive = $collection->sortBy('age')->take(5);

// Prenons 5 personnes d'une collection en commençant par la position 4
$nextTopFive = $collection->sortBy('age')->take(5, 4);
```

Les positions sont basées sur zéro, donc le premier nombre de la position est `0`.

`Cake\Collection\Collection::skip(`*$length*`)`

Alors que le second argument de `take()` peut vous aider à exclure quelques éléments avant de les récupérer depuis une collection, vous pouvez également utiliser `skip()` pour récupérer le reste des éléments après une certaine position :

```
$collection = new Collection([1, 2, 3, 4]);
$allExceptFirstTwo = $collection->skip(2)->toList(); // [3, 4]
```

`Cake\Collection\Collection::first()`

Un des cas d'utilisation les plus courant de `take()` est de récupérer le premier élément d'un collection. Une moyen plus rapide d'arriver au même résultat est d'utiliser la méthode `first()` :

```
$collection = new Collection([5, 4, 3, 2]);
$collection->first(); // Retourne 5
```

`Cake\Collection\Collection::last()`

De la même manière, vous pouvez récupérer le dernier élément d'une collection en utilisant la méthode `last()` :

```
$collection = new Collection([5, 4, 3, 2]);
$collection->last(); // Returns 2
```

Agrandir les Collections

`Cake\Collection\Collection::append(`*array|Traversable $items*`)`

Vous pouvez regrouper plusieurs collections en une collection unique. Ceci vous permet de recueillir des données provenant de diverses sources, de concaténer et de lui appliquer d'autres fonctions de collection très en douceur. La méthode `append()` va retourner une nouvelle collection contenant les valeurs à partir des deux sources :

```
$cakephpTweets = new Collection($tweets);
$myTimeline = $cakephpTweets->append($phpTweets);

// Tweets contenant `cakefest` à partir des deux sources
$myTimeline->filter(function ($tweet) {
    return strpos($tweet, 'cakefest');
});
```

Avertissement

Quand vous ajoutez différentes sources, vous pouvez avoir certaines clés des deux collections qui sont les mêmes. Par exemple, quand vous ajoutez deux tableaux unidimensionnels. Ceci peut entraîner un problème quand vous convertissez une collection en un tableau en utilisant `toArray()`. Si vous ne voulez pas que des valeurs d'une collection surchargent les autres dans la précédente basée sur leur clé, assurez-vous que vous appelez `toList()` afin de supprimer les clés et de préserver toutes les valeurs.

Modification d'Eléments

`Cake\Collection\Collection::insert(`*$path*, *$items*`)`

A certains moments, vous pourriez avoir à séparer des ensembles de données que vous souhaiteriez, pour insérer les éléments d'un ensemble dans chacun des éléments de l'autre ensemble. C'est un cas très courant quand vous récupérez les données à partir d'une source de données qui ne supporte pas la fusion de données ou les jointures nativement.

Les collections ont une méthode `insert()` qui vous permet d'insérer chacun des éléments dans une collection dans une propriété dans chacun des éléments d'une autre collection :

```
$users = [
    ['username' => 'mark'],
    ['username' => 'juan'],
    ['username' => 'jose']
];

$languages = [
    ['PHP', 'Python', 'Ruby'],
    ['Bash', 'PHP', 'Javascript'],
    ['Javascript', 'Prolog']
];

$merged = (new Collection($users))->insert('skills', $languages);
```

Une fois convertie en un tableau, la collection `$merged` va ressembler à ceci :

```
[
    ['username' => 'mark', 'skills' => ['PHP', 'Python', 'Ruby']],
    ['username' => 'juan', 'skills' => ['Bash', 'PHP', 'Javascript']],
    ['username' => 'jose', 'skills' => ['Javascript', 'Prolog']]
];
```

Le premier paramètre de la méthode `insert()` est un chemin séparé par des points des propriétés à suivre pour que les éléments puissent être insérés à cette position. Le second argument est tout ce qui peut être converti en objets collection.

Veuillez noter que les éléments sont insérés par la position dans laquelle ils sont trouvés, ainsi le premier élément de la deuxième collection est fusionné dans le premier élément de la première collection.

S'il y a assez d'éléments de la seconde collection à insérer dans la première, alors la propriété cible va être remplie avec les valeurs `null` :

```
$languages = [
    ['PHP', 'Python', 'Ruby'],
    ['Bash', 'PHP', 'Javascript']
];

$merged = (new Collection($users))->insert('skills', $languages);

// Va retourner
[
    ['username' => 'mark', 'skills' => ['PHP', 'Python', 'Ruby']],
    ['username' => 'juan', 'skills' => ['Bash', 'PHP', 'Javascript']],
    ['username' => 'jose', 'skills' => null]
];
```

La méthode `insert()` peut opérer sur des éléments tableau ou des objets qui implémentent l'interface `ArrayAccess`.

Créer des Méthodes de Collection Réutilisables

Utiliser une `Closure` pour les méthodes de Collection est optimal lorsque le travail à accomplir est petit et ciblé, mais cela peut devenir gênant très rapidement. Cela devient plus évident quand beaucoup de méthodes différentes doivent être appelées ou lorsque la longueur des méthodes de la `Closure` est de plus de quelques lignes.

Il y a aussi des cas où la logique utilisée pour les méthodes de Collection peut être réutilisée dans plusieurs parties de votre application. Il est préférable d'envisager d'éclater la logique d'ensemble complexe dans des classes séparées. Par exemple, imaginez une longue restriction comme celle-ci :

```
$collection
        ->map(function ($row, $key) {
            if (!empty($row['items'])) {
                $row['total'] = collection($row['items'])->sumOf('price');
            }

            if (!empty($row['total'])) {
                $row['tax_amount'] = $row['total'] * 0.25;
            }

            // More code here...

            return $modifiedRow;
        });
```

Cela peut être remodeler en créant une autre classe :

```
class TotalOrderCalculator
{

        public function __invoke($row, $key)
        {
            if (!empty($row['items'])) {
                $row['total'] = collection($row['items'])->sumOf('price');
            }
```

(suite sur la page suivante)

(suite de la page précédente)

```
            if (!empty($row['total'])) {
                $row['tax_amount'] = $row['total'] * 0.25;
            }

            // More code here...

            return $modifiedRow;
        }
}

// Use the logic in your map() call
$collection->map(new TotalOrderCalculator)
```

`Cake\Collection\Collection::through($callback)`

Parfois une suite d'appels de méthodes de Collection peut devenir réutilisable dans d'autres parties de votre application, mais seulement si elles sont appelées dans cet ordre précis. Dans ces cas, vous pouvez utiliser les `through()` en combinaison avec une classe implémentant `__invoke` pour répartir vos traitements de données :

```
$collection
        ->map(new ShippingCostCalculator)
        ->map(new TotalOrderCalculator)
        ->map(new GiftCardPriceReducer)
        ->buffered()
       ...
```

Les appels aux méthodes ci-dessus, peuvent être regroupés dans une nouvelle classe permettant de ne pas être répétés à chaque fois :

```
class FinalCheckOutRowProcessor
{

        public function __invoke($collection)
        {
                return $collection
                        ->map(new ShippingCostCalculator)
                        ->map(new TotalOrderCalculator)
                        ->map(new GiftCardPriceReducer)
                        ->buffered()
                       ...
        }
}

// Maintenant vous pouvez utiliser la méthode through() pour appeler toutes les méthodes␣
↪en une fois
$collection->through(new FinalCheckOutRowProcessor);
```

Optimiser les Collections

`Cake\Collection\Collection::buffered()`

Les collections effectuent souvent la plupart des opérations que vous créez en utilisant ses fonctions de façon lazy. Ceci signifie que même si vous pouvez appeler une fonction, cela ne signifie pas qu'elle est exécutée de la bonne manière. C'est vrai pour une grande quantité de fonctions de cette classe. L'évaluation lazy vous permet de gagner des ressources dans des situations où vous n'utilisez pas toutes les valeurs d'une collection. Vous pouvez ne pas utiliser toutes les valeurs quand l'itération stoppe rapidement, ou quand une exception/un échec se produit rapidement.

De plus, l'évaluation lazy aide à accélérer certaines operations. Considérez l'exemple suivant :

```
$collection = new Collection($oneMillionItems);
$collection = $collection->map(function ($item) {
    return $item * 2;
});
$itemsToShow = $collection->take(30);
```

Si nous avions des collections non lazy, nous aurions dû executer un million d'opérations, même si nous voulions seulement montrer 30 éléments. A la place, notre opération map a été seulement appliquée aux 30 éléments que nous avons utilisés. Nous pouvons aussi tirer des bénéfices de l'évaluation lazy pour des collections plus petites quand nous faisons plus qu'une opération sur elles. Par exemple : appeler `map()` deux fois et ensuite `filter()`.

L'évaluation lazy a aussi ses inconvénients. Vous pourriez faire les mêmes opérations plus d'une fois si vous optimisiez une collection prématurément. Considérons cet exemple :

```
$ages = $collection->extract('age');

$youngerThan30 = $ages->filter(function ($item) {
    return $item < 30;
});

$olderThan30 = $ages->filter(function ($item) {
    return $item > 30;
});
```

Si nous itérons `youngerThan30` et `olderThan30`, la collection exécuterait malheureusement l'opération `extract()` deux fois. C'est parce que les collections sont immutables et l'opération d'extraction lazy serait fait pour les deux filtres.

Heureusement, nous pouvons passer outre ce problème avec une simple fonction. Si vous planifiez de réutiliser les valeurs à partir de certaines opérations plus d'une fois, vous pouvez compiler les résultats dans une autre collection en utilisant la fonction `buffered()` :

```
$ages = $collection->extract('age')->buffered();
$youngerThan30 = ...
$olderThan30 = ...
```

Maintenant quand les deux collections sont itérées, elles vont seulement appeler l'opération d'extraction en une fois.

Rendre les Collections Rembobinables

La méthode `buffered()` est aussi utile pour convertir des itérateurs non-rembobinables dans des collections qui peuvent être itérées plus d'une fois :

```
// Dans PHP 5.5+
public function results()
```

(suite sur la page suivante)

(suite de la page précédente)

```
{
    ...
    foreach ($transientElements as $e) {
        yield $e;
    }
}
$rewindable = (new Collection(results()))->buffered();
```

Clonage de Collection

`Cake\Collection\Collection::compile($preserveKeys = true)`

Parfois vous devez cloner un des éléments à partir d'une collection. C'est utile quand vous avez besoin d'itérer le même ensemble à partir d'endroits différents au même moment. Afin de cloner une collection à partir d'une autre, utilisez la méthode `compile()` :

```
$ages = $collection->extract('age')->compile();

foreach ($ages as $age) {
    foreach ($collection as $element) {
        echo h($element->name) . ' - ' . $age;
    }
}
```

CHAPITRE 32

Folder & File

Les utilitaires Folder et File sont des classes pratiques pour la lecture, l'écriture/l'ajout de fichiers, lister les fichiers d'un dossier et toute autre tâche habituelle liée aux répertoires.

Obsolète depuis la version 4.0 : Les classes `File` et `Folder` vont être retirées dans la version 5.0. Utilisez plutôt les classes `SplFileInfo` ou `SplFileObject` et les classes d'itérateurs comme `RecursiveDirectoryIterator`, `RecursiveRegexIterator` etc.

Utilisation Basique

Assurez-vous que les classes sont chargées :

```
use Cake\Filesystem\Folder;
use Cake\Filesystem\File;
```

Ensuite nous pouvons configurer une nouvelle instance de dossier :

```
$dir = new Folder('/path/to/folder');
```

et chercher tous les fichiers *.php* à l'intérieur de ce dossier en utilisant les regex :

```
$files = $dir->find('.*\.php');
```

Maintenant nous pouvons faire une boucle sur les fichiers et les lire, écrire/ajouter aux contenus, ou simplement supprimer le fichier :

```
foreach ($files as $file) {
    $file = new File($dir->pwd() . DS . $file);
    $contents = $file->read();
    // $file->write('J'écris dans ce fichier');
    // $file->append('J'ajoute à la fin de ce fichier.');
```

(suite sur la page suivante)

(suite de la page précédente)

```
    // $file->delete(); // Je supprime ce fichier
    $file->close(); // Assurez-vous de fermer le fichier quand c'est fini
}
```

API de Folder

class Cake\Filesystem**Folder**(*string $path = false*, *boolean $create = false*, *string|boolean $mode = false*)

```
// Crée un nouveau dossier avec les permissions à 0755
$dir = new Folder('/path/to/folder', true, 0755);
```

property Cake\Filesystem\Folder::$**path**

Le chemin pour le dossier courant. *Folder::pwd()* retournera la même information.

property Cake\Filesystem\Folder::$**sort**

Dit si la liste des résultats doit être oui ou non rangée par ordre alphabétique.

property Cake\Filesystem\Folder::$**mode**

Mode à utiliser pour la création de dossiers. par défaut à `0755`. Ne fait rien sur les machines Windows.

static Cake\Filesystem\Folder::**addPathElement**(*string $path*, *string $element*)

Retourne $path avec $element ajouté, avec le bon slash entre-deux :

```
$path = Folder::addPathElement('/un/chemin/pour', 'tester');
// $path égal /un/chemin/pour/tester
```

$element peut aussi être un tableau :

```
$path = Folder::addPathElement('/un/chemin/pour', ['un-autre', 'test']);
// $path égal à /un/chemin/pour/un-autre/test
```

Cake\Filesystem\Folder::**cd**(*$path*)

Change le répertoire en $path. Retourne `false` en cas d'échec :

```
$folder = new Folder('/foo');
echo $folder->path; // Affiche /foo
$folder->cd('/bar');
echo $folder->path; // Affiche /bar
$false = $folder->cd('/repertoire-inexistant');
```

Cake\Filesystem\Folder::**chmod**(*string $path*, *integer $mode = false*, *boolean $recursive = true*, *array $exceptions = []*)

Change le mode sur la structure de répertoire de façon récursive. Ceci inclut aussi le changement du mode des fichiers :

```
$dir = new Folder();
$dir->chmod('/path/to/folder', 0755, true, ['skip_me.php']);
```

Cake\Filesystem\Folder::**copy**(*array|string $options = []*)

Copie de façon récursive un répertoire. Le seul paramètre $options peut être soit un chemin à copier soit un tableau d'options :

```
$folder1 = new Folder('/path/to/folder1');
$folder1->copy('/path/to/folder2');
// mettra le folder1 et tout son contenu dans folder2

$folder = new Folder('chemin/vers/repertoire');
$folder->copy([
    'to' => '/chemin/vers/nouveau/repertoire',
    'from' => '/chemin/de/depart', // Va provoquer un cd()
    'mode' => 0755,
    'skip' => ['bon-a-ignorer.php', '.git'],
    'scheme' => Folder::SKIP  // Ne crée pas les répertoires/fichiers qui existent␣
↪déjà.
]);
```

Il y a 3 schémas supportés :

— `Folder::SKIP` échapper la copie/déplacement des fichiers & répertoires qui existent dans le répertoire de destination.
— `Folder::MERGE` fusionne les répertoires source/destination. Les fichiers dans le répertoire source vont remplacer les fichiers dans le répertoire de cible. Les contenus du répertoire seront fusionnés.
— `Folder::OVERWRITE` écrase les fichiers & répertoires existant dans le répertoire cible avec ceux dans le répertoire source. Si la source et la destination contiennent le même sous-répertoire, les contenus du répertoire de cible vont être retirés et remplacés avec celui de la source.

static `Cake\Filesystem\Folder::`**correctSlashFor**(*string $path*)

Retourne un ensemble correct de slashes pour un $path donné. ("\" pour les chemins Windows et "/" pour les autres chemins).

`Cake\Filesystem\Folder::`**create**(*string $pathname*, *integer $mode = false*)

Crée une structure de répertoire de façon récursive. Peut être utilisée pour créer des structures de chemin profond comme */foo/bar/baz/shoe/horn* :

```
$folder = new Folder();
if ($folder->create('foo' . DS . 'bar' . DS . 'baz' . DS . 'shoe' . DS . 'horn')) {
    // Répertoires imbriqués créés avec succèse
}
```

`Cake\Filesystem\Folder::`**delete**(*string $path = null*)

Efface de façon récursive les répertoires si le système le permet :

```
$folder = new Folder('foo');
if ($folder->delete()) {
    // Supprime foo et ses dossiers imbriqués avec succès
}
```

`Cake\Filesystem\Folder::`**dirsize**()

Retourne la taille en octets de ce dossier et ses contenus.

`Cake\Filesystem\Folder::`**errors**()

Récupère l'erreur de la dernière méthode.

`Cake\Filesystem\Folder::`**find**(*string $regexpPattern = '.*'*, *boolean $sort = false*)

Retourne un tableau de tous les fichiers correspondants dans le répertoire courant :

```
// Trouve tous les .png dans votre dossier webroot/img/ et range les résultats
$dir = new Folder(WWW_ROOT . 'img');
$files = $dir->find('.*\.png', true);
/*
Array
(
    [0] => cake.icon.png
    [1] => test-error-icon.png
    [2] => test-fail-icon.png
    [3] => test-pass-icon.png
    [4] => test-skip-icon.png
)
*/
```

Note

Les méthodes find et findRecursive de folder ne trouvent seulement que des fichiers. Si vous voulez obtenir des dossiers et fichiers, regardez `Folder::read()` ou `Folder::tree()`.

`Cake\Filesystem\Folder::findRecursive`(*string $pattern = '.*', boolean $sort = false*)

Retourne un tableau de tous les fichiers correspondants dans et en-dessous du répertoire courant :

```
// Trouve de façon récursive les fichiers commençant par test ou index
$dir = new Folder(WWW_ROOT);
$files = $dir->findRecursive('(test|index).*');
/*
Array
(
    [0] => /var/www/cake/webroot/index.php
    [1] => /var/www/cake/webroot/test.php
    [2] => /var/www/cake/webroot/img/test-skip-icon.png
    [3] => /var/www/cake/webroot/img/test-fail-icon.png
    [4] => /var/www/cake/webroot/img/test-error-icon.png
    [5] => /var/www/cake/webroot/img/test-pass-icon.png
)
*/
```

`Cake\Filesystem\Folder::inCakePath`(*string $path = ''*)

Retourne `true` si le fichier est dans un CakePath donné.

`Cake\Filesystem\Folder::inPath`(*string $path = '', boolean $reverse = false*)

Retourne `true` si le fichier est dans le chemin donné :

```
$Folder = new Folder(WWW_ROOT);
$result = $Folder->inPath(APP);
// $result = true, /var/www/example/ est dans /var/www/example/webroot/

$result = $Folder->inPath(WWW_ROOT . 'img' . DS, true);
// $result = true, /var/www/example/webroot/ est dans /var/www/example/webroot/img/
```

`static Cake\Filesystem\Folder::isAbsolute`(*string $path*)

Retourne `true` si le $path donné est un chemin absolu.

static Cake\Filesystem\Folder::**isSlashTerm**(*string $path*)

Retourne `true` si le $path donné finit par un slash (par exemple. se termine-par-un-slash) :

```
$result = Folder::isSlashTerm('/my/test/path');
// $result = false
$result = Folder::isSlashTerm('/my/test/path/');
// $result = true
```

static Cake\Filesystem\Folder::**isWindowsPath**(*string $path*)

Retourne `true` si le $path donné est un chemin Windows.

Cake\Filesystem\Folder::**messages**()

Récupère les messages de la dernière méthode.

Cake\Filesystem\Folder::**move**(*array $options*)

Déplace le répertoire de façon récursive.

static Cake\Filesystem\Folder::**normalizeFullPath**(*string $path*)

Retourne un chemin avec des slashes normalisés pour le système d'exploitation.

Cake\Filesystem\Folder::**pwd**()

Retourne le chemin courant.

Cake\Filesystem\Folder::**read**(*boolean $sort = true*, *array|boolean $exceptions = false*, *boolean $fullPath = false*)

Retourne un tableau du contenu du répertoire courant. Le tableau retourné contient deux sous-tableaux : Un des repertoires et un des fichiers :

```
$dir = new Folder(WWW_ROOT);
$files = $dir->read(true, ['files', 'index.php']);
/*
Array
(
    [0] => Array // dossiers
        (
            [0] => css
            [1] => img
            [2] => js
        )
    [1] => Array // fichiers
        (
            [0] => .htaccess
            [1] => favicon.ico
            [2] => test.php
        )
)
*/
```

Cake\Filesystem\Folder::**realpath**(*string $path*)

Récupère le vrai chemin (en prenant en compte « .. » etc…).

static Cake\Filesystem\Folder::**slashTerm**(*string $path*)

Retourne $path avec le slash ajouté à la fin (corrigé pour Windows ou d'autres OS).

`Cake\Filesystem\Folder::tree`(*null|string $path = null*, *array|boolean $exceptions = true*, *null|string $type = null*)

Retourne un tableau de répertoires imbriqués et de fichiers dans chaque répertoire.

L'API de File

class `Cake\Filesystem\File`(*string $path*, *boolean $create = false*, *integer $mode = 755*)

```
// Crée un nouveau fichier avec les permissions à 0644
$file = new File('/path/to/file.php', true, 0644);
```

property `Cake\Filesystem\File::$Folder`

L'objet Folder du fichier.

property `Cake\Filesystem\File::$name`

Le nom du fichier avec l'extension. Diffère de `File::name()` qui retourne le nom sans l'extension.

property `Cake\Filesystem\File::$info`

Un tableau du fichier info. Utilisez `File::info()` à la place.

property `Cake\Filesystem\File::$handle`

Contient le fichier de gestion des ressources si le fichier est ouvert.

property `Cake\Filesystem\File::$lock`

Active le blocage du fichier en lecture et en écriture.

property `Cake\Filesystem\File::$path`

Le chemin absolu du fichier courant.

`Cake\Filesystem\File::append`(*string $data*, *boolean $force = false*)

Ajoute la chaîne de caractères donnée au fichier courant.

`Cake\Filesystem\File::close`()

Ferme le fichier courant s'il est ouvert.

`Cake\Filesystem\File::copy`(*string $dest*, *boolean $overwrite = true*)

Copie le fichier vers `$dest`.

`Cake\Filesystem\File::create`()

Crée le fichier.

`Cake\Filesystem\File::delete`()

Supprime le fichier.

`Cake\Filesystem\File::executable`()

Retourne `true` si le fichier est exécutable.

`Cake\Filesystem\File::exists`()

Retourne `true` si le fichier existe.

`Cake\Filesystem\File::ext`()

Retourne l'extension du fichier.

`Cake\Filesystem\File::Folder`()

Retourne le dossier courant.

`Cake\Filesystem\File::group()`
Retourne le groupe du fichier ou `false` en cas d'erreur.

`Cake\Filesystem\File::info()`
Retourne l'info du fichier.

`Cake\Filesystem\File::lastAccess()`
Retourne l'heure du dernier d'accès.

`Cake\Filesystem\File::lastChange()`
Retourne l'heure de la dernière modification ou `false` en cas d'erreur.

`Cake\Filesystem\File::md5(integer|boolean $maxsize = 5)`
Récupère la checksum MD5 du fichier avec vérification préalable de la taille du fichier ou `false` en cas d'erreur.

`Cake\Filesystem\File::name()`
Retourne le nom du fichier sans l'extension.

`Cake\Filesystem\File::offset(integer|boolean $offset = false, integer $seek = 0)`
Définit ou récupère l'offset pour le fichier ouvert.

`Cake\Filesystem\File::open(string $mode = 'r', boolean $force = false)`
Ouvre le fichier courant avec un $mode donné.

`Cake\Filesystem\File::owner()`
Retourne le propriétaire du fichier.

`Cake\Filesystem\File::perms()`
Retourne le « chmod » (permissions) du Fichier.

`static Cake\Filesystem\File::prepare(string $data, boolean $forceWindows = false)`
Prépare une chaîne de caractères ascii pour l'écriture. Convertit les fins de lignes en caractère correct pour la plateforme courante. Si c'est Windows « \r\n » sera utilisé, toutes les autres plateformes utiliseront « \n ».

`Cake\Filesystem\File::pwd()`
Retourne un chemin complet du fichier.

`Cake\Filesystem\File::read(string $bytes = false, string $mode = 'rb', boolean $force = false)`
Retourne les contenus du fichier en chaîne de caractères ou retourne `false` en cas d'échec.

`Cake\Filesystem\File::readable()`
Retourne `true` si le fichier est lisible.

`Cake\Filesystem\File::safe(string $name = null, string $ext = null)`
Sécurise le nom de fichier pour la sauvegarde.

`Cake\Filesystem\File::size()`
Retourne la taille du fichier en octets.

`Cake\Filesystem\File::writable()`
Retourne `true` si le fichier est ouvert en écriture.

`Cake\Filesystem\File::write(string $data, string $mode = 'w', boolean$force = false)`
Ecrit le $data donné dans le fichier.

`Cake\Filesystem\File::mime()`
Récupère le mimetype du fichier, retourne `false` en cas d'échec.

`Cake\Filesystem\File::replaceText($search, $replace)`
Remplace le texte dans un fichier. Retourne `false` en cas d'échec et `true` en cas de succès.

CHAPITRE 33

Hash

class Cake\Utility**Hash**

La gestion de tableau, si elle est bien faite, peut être un outil très puissant et utile pour construire du code plus intelligent et plus optimisé. CakePHP offre un ensemble d'utilitaires statiques très utile dans la classe Hash qui vous permet de faire justement cela.

La classe Hash de CakePHP peut être appelée à partir de n'importe quel model ou controller de la même façon que pour un appel à Inflector Exemple : *Hash::combine()*.

Syntaxe de chemin Hash

La syntaxe de chemin décrite ci-dessous est utilisée par toutes les méthodes dans `hash()`. Les parties de la syntaxe du chemin ne sont pas toutes disponibles dans toutes les méthodes. Une expression en chemin est faite depuis n'importe quel nombre de tokens. Les Tokens sont composés de deux groupes. Les Expressions sont utilisées pour parcourir le tableau de données, alors que les matchers sont utilisés pour qualifier les éléments. Vous appliquez les matchers aux éléments de l'expression.

Types d'expression

Expression	Définition
`{n}`	Représente une clé numérique. Va matcher toute chaîne ou clé numérique.
`{s}`	Représente une chaîne. Va matcher toute valeur de chaîne y compris les valeurs de chaîne numérique.
`{*}`	Toutes les valeurs seront *matchées*
`Foo`	Matche les clés avec exactement la même valeur.

Tous les éléments d'expression supportent toutes les méthodes. En plus des éléments d'expression, vous pouvez utiliser les attributs qui matchent avec certaines méthodes. Il y a `extract()`, `combine()`, `format()`, `check()`, `map()`, `reduce()`, `apply()`, `sort()`, `insert()`, `remove()` et `nest()`.

Les Types d'Attribut Correspondants

Matcher	Définition
`[id]`	Match les éléments avec une clé de tableau donnée.
`[id=2]`	Match les éléments avec un id égal à 2.
`[id!=2]`	Match les éléments avec un id non égal à 2.
`[id>2]`	Match les éléments avec un id supérieur à 2.
`[id>=2]`	Match les éléments avec un id supérieur ou égal à 2.
`[id<2]`	Match les éléments avec un id inférieur à 2.
`[id<=2]`	Match les éléments avec un id inférieur ou égal à 2.
`[text=/.../]`	Match les éléments qui ont des valeurs matchant avec l'expression régulière à l'intérieur de `...`.

static `Cake\Utility\Hash::`**`get`**(*array|\ArrayAccess $data*, *$path*, *$default = null*)

`get()` est une version simplifiée de `extract()`, elle ne supporte que les expressions de chemin direct. Les chemins avec `{n}`, `{s}`, `{*}` ou les matchers ne sont pas supportés. Utilisez `get()` quand vous voulez exactement une valeur sortie d'un tableau. Si un chemin correspondant n'est pas trouvé, la valeur par défaut sera retournée.

static `Cake\Utility\Hash::`**`extract`**(*array|\ArrayAccess $data*, *$path*)

`Hash::extract()` supporte toutes les expressions, les components matcher de la *Syntaxe de chemin Hash*. Vous pouvez utiliser l'extract pour récupérer les données à partir des tableaux, ou bien un objet implémentant l'interface `ArrayAccess` avec des chemins arbitraires rapidement sans avoir à parcourir les structures de données. A la place, vous utilisez les expressions de chemin pour qualifier les éléments que vous souhaitez retourner :

```
// Utilisation habituelle:
$users = [
    ['id' => 1, 'name' => 'mark'],
    ['id' => 2, 'name' => 'jane'],
    ['id' => 3, 'name' => 'sally'],
    ['id' => 4, 'name' => 'jose'],
];
$results = Hash::extract($users, '{n}.id');
// $results égal à:
// [1,2,3,4];
```

static `Cake\Utility\Hash::`**`insert`**(*array $data*, *$path*, *$values = null*)

Insère `$values` dans un tableau tel que défini dans `$path` :

```
$a = [
    'pages' => ['name' => 'page']
];
$result = Hash::insert($a, 'files', ['name' => 'files']);
// $result ressemble maintenant à:
[
    [pages] => [
            [name] => page
    ]
    [files] => [

            [name] => files
    ]
]
```

Vous pouvez utiliser les chemins en utilisant `{n}`, `{s}` et `{*}` pour insérer des données dans des points multiples :

```
$users = Hash::insert($users, '{n}.new', 'value');
```

Les matchers d'attribut fonctionnent aussi avec `insert()` :

```
$data = [
    0 => ['up' => true, 'Item' => ['id' => 1, 'title' => 'first']],
    1 => ['Item' => ['id' => 2, 'title' => 'second']],
    2 => ['Item' => ['id' => 3, 'title' => 'third']],
    3 => ['up' => true, 'Item' => ['id' => 4, 'title' => 'fourth']],
    4 => ['Item' => ['id' => 5, 'title' => 'fifth']],
];
$result = Hash::insert($data, '{n}[up].Item[id=4].new', 9);
/* $result ressemble maintenant à:
    [
        ['up' => true, 'Item' => ['id' => 1, 'title' => 'first']],
        ['Item' => ['id' => 2, 'title' => 'second']],
        ['Item' => ['id' => 3, 'title' => 'third']],
        ['up' => true, 'Item' => ['id' => 4, 'title' => 'fourth', 'new' => 9]],
        ['Item' => ['id' => 5, 'title' => 'fifth']],
    ]
*/
```

static `Cake\Utility\Hash::`**`remove`**(*array $data*, *$path*)

Retire tous les éléments d'un tableau qui matche avec `$path` :

```
$a = [
    'pages' => ['name' => 'page'],
    'files' => ['name' => 'files']
];
$result = Hash::remove($a, 'files');
/* $result ressemble maintenant à:
    [
        [pages] => [
                [name] => page
    ]

    ]
*/
```

L'utilisation de `{n}`, `{s}` et `{*}` vous autorisera à retirer les valeurs multiples en une fois. Vous pouvez aussi utiliser les matchers d'attribut avec `remove()` :

```
$data = [
    0 => ['clear' => true, 'Item' => ['id' => 1, 'title' => 'first']],
    1 => ['Item' => ['id' => 2, 'title' => 'second']],
    2 => ['Item' => ['id' => 3, 'title' => 'third']],
    3 => ['clear' => true, 'Item' => ['id' => 4, 'title' => 'fourth']],
    4 => ['Item' => ['id' => 5, 'title' => 'fifth']],
];
$result = Hash::remove($data, '{n}[clear].Item[id=4]');
/* $result ressemble maintenant à:
    [
```

(suite sur la page suivante)

(suite de la page précédente)

```
        ['clear' => true, 'Item' => ['id' => 1, 'title' => 'first']],
        ['Item' => ['id' => 2, 'title' => 'second']],
        ['Item' => ['id' => 3, 'title' => 'third']],
        ['clear' => true],
        ['Item' => ['id' => 5, 'title' => 'fifth']],
    ]
*/
```

static Cake\Utility\Hash::**combine**(*array $data*, *$keyPath*, *$valuePath = null*, *$groupPath = null*)

Crée un tableau associatif en utilisant `$keyPath` en clé pour le chemin à construire, et optionnellement `$valuePath` comme chemin pour récupérer les valeurs. Si `$valuePath` n'est pas spécifiée, ou ne matche rien, les valeurs seront initialisées à null. Vous pouvez grouper en option les valeurs par ce qui est obtenu en suivant le chemin spécifié dans `$groupPath` :

```
$a = [
    [
        'User' => [
            'id' => 2,
            'group_id' => 1,
            'Data' => [
                'user' => 'mariano.iglesias',
                'name' => 'Mariano Iglesias'
            ]
        ]
    ],
    [
        'User' => [
            'id' => 14,
            'group_id' => 2,
            'Data' => [
                'user' => 'phpnut',
                'name' => 'Larry E. Masters'
            ]
        ]
    ],
];

$result = Hash::combine($a, '{n}.User.id');
/* $result ressemble maintenant à:
    [
        [2] =>
        [14] =>
    ]
*/

$result = Hash::combine($a, '{n}.User.id', '{n}.User.Data.user');
/* $result ressemble maintenant à:
    [
        [2] => 'mariano.iglesias'
        [14] => 'phpnut'
    ]
*/
```

(suite sur la page suivante)

(suite de la page précédente)

```
$result = Hash::combine($a, '{n}.User.id', '{n}.User.Data');
/* $result ressemble maintenant à:
    [
        [2] => [
                [user] => mariano.iglesias
                [name] => Mariano Iglesias
        ]
        [14] => [
                [user] => phpnut
                [name] => Larry E. Masters
        ]
    ]
*/

$result = Hash::combine($a, '{n}.User.id', '{n}.User.Data.name');
/* $result ressemble maintenant à:
    [
        [2] => Mariano Iglesias
        [14] => Larry E. Masters
    ]
*/

$result = Hash::combine($a, '{n}.User.id', '{n}.User.Data', '{n}.User.group_id');
/* $result ressemble maintenant à:
    [
        [1] => [
                [2] => [
                        [user] => mariano.iglesias
                        [name] => Mariano Iglesias
                ]
        ]
        [2] => [
                [14] => [
                        [user] => phpnut
                        [name] => Larry E. Masters
                ]
        ]
    ]
*/

$result = Hash::combine($a, '{n}.User.id', '{n}.User.Data.name', '{n}.User.group_id
↪');
/* $result ressemble maintenant à:
    [
        [1] => [
                [2] => Mariano Iglesias
        ]
        [2] => [
                [14] => Larry E. Masters
        ]
    ]
```

(suite sur la page suivante)

(suite de la page précédente)

```
*/
```

Vous pouvez fournir des tableaux pour les deux `$keyPath` et `$valuePath`. Si vous le faîtes, la première valeur sera utilisée comme un format de chaîne de caractères, pour les valeurs extraites par les autres chemins :

```
$result = Hash::combine(
    $a,
    '{n}.User.id',
    ['%s: %s', '{n}.User.Data.user', '{n}.User.Data.name'],
    '{n}.User.group_id'
);
/* $result ressemble maintenant à:
    [
        [1] => [
                [2] => mariano.iglesias: Mariano Iglesias
        ]
        [2] => [
                [14] => phpnut: Larry E. Masters
        ]
    ]
*/

$result = Hash::combine(
    $a,
    ['%s: %s', '{n}.User.Data.user', '{n}.User.Data.name'],
    '{n}.User.id'
);
/* $result ressemble maintenant à:
    [
        [mariano.iglesias: Mariano Iglesias] => 2
        [phpnut: Larry E. Masters] => 14
    ]
*/
```

static `Cake\Utility\Hash::`**format**(*array $data*, *array $paths*, *$format*)

Retourne une série de valeurs extraites d'un tableau, formaté avec un format de chaîne de caractères :

```
$data = [
    [
        'Person' => [
            'first_name' => 'Nate',
            'last_name' => 'Abele',
            'city' => 'Boston',
            'state' => 'MA',
            'something' => '42'
        ]
    ],
    [
        'Person' => [
            'first_name' => 'Larry',
            'last_name' => 'Masters',
            'city' => 'Boondock',
```

(suite sur la page suivante)

(suite de la page précédente)

```
            'state' => 'TN',
            'something' => '{0}'
        ]
    ],
    [
        'Person' => [
            'first_name' => 'Garrett',
            'last_name' => 'Woodworth',
            'city' => 'Venice Beach',
            'state' => 'CA',
            'something' => '{1}'
        ]
    ]
];

$res = Hash::format($data, ['{n}.Person.first_name', '{n}.Person.something'], '%2$d,
↪ %1$s');
/*
[
    [0] => 42, Nate
    [1] => 0, Larry
    [2] => 0, Garrett
]
*/

$res = Hash::format($data, ['{n}.Person.first_name', '{n}.Person.something'], '%1$s,
↪ %2$d');
/*
[
    [0] => Nate, 42
    [1] => Larry, 0
    [2] => Garrett, 0
]
*/
```

static Cake\Utility\Hash::**contains**(*array $data*, *array $needle*)

Détermine si un Hash ou un tableau contient les clés et valeurs exactes d'un autre :

```
$a = [
    0 => ['name' => 'main'],
    1 => ['name' => 'about']
];
$b = [
    0 => ['name' => 'main'],
    1 => ['name' => 'about'],
    2 => ['name' => 'contact'],
    'a' => 'b'
];

$result = Hash::contains($a, $a);
// true
$result = Hash::contains($a, $b);
```

(suite sur la page suivante)

(suite de la page précédente)

```
// false
$result = Hash::contains($b, $a);
// true
```

static Cake\Utility\Hash::**check**(*array $data*, *string $path = null*)

Vérifie si un chemin particulier est défini dans un tableau :

```
$set = [
    'My Index 1' => ['First' => 'The first item']
];
$result = Hash::check($set, 'My Index 1.First');
// $result == True

$result = Hash::check($set, 'My Index 1');
// $result == True

$set = [
    'My Index 1' => [
        'First' => [
            'Second' => [
                'Third' => [
                    'Fourth' => 'Heavy. Nesting.'
                ]
            ]
        ]
    ]
];
$result = Hash::check($set, 'My Index 1.First.Second');
// $result == True

$result = Hash::check($set, 'My Index 1.First.Second.Third');
// $result == True

$result = Hash::check($set, 'My Index 1.First.Second.Third.Fourth');
// $result == True

$result = Hash::check($set, 'My Index 1.First.Seconds.Third.Fourth');
// $result == False
```

static Cake\Utility\Hash::**filter**(*array $data, $callback = ['Hash', 'filter']*)

Filtre les éléments vides en dehors du tableau, en excluant "0". Vous pouvez aussi fournir un `$callback` personnalisé pour filtrer les éléments de tableau. Votre callback devrait retourner `false` pour retirer les éléments du tableau résultant :

```
$data = [
    '0',
    false,
    true,
    0,
    ['one thing', 'I can tell you', 'is you got to be', false]
];
$res = Hash::filter($data);
```

(suite sur la page suivante)

(suite de la page précédente)

```
/* $res ressemble maintenant à:
    [
        [0] => 0
        [2] => true
        [3] => 0
        [4] => [
                [0] => one thing
                [1] => I can tell you
                [2] => is you got to be
        ]
    ]
*/
```

static Cake\Utility\Hash::**flatten**(*array $data*, *string $separator = '.'*)

Réduit un tableau multi-dimensionnel en un tableau à une seule dimension :

```
$arr = [
    [
        'Post' => ['id' => '1', 'title' => 'First Post'],
        'Author' => ['id' => '1', 'user' => 'Kyle'],
    ],
    [
        'Post' => ['id' => '2', 'title' => 'Second Post'],
        'Author' => ['id' => '3', 'user' => 'Crystal'],
    ],
];
$res = Hash::flatten($arr);
/* $res ressemble maintenant à:
    [
        [0.Post.id] => 1
        [0.Post.title] => First Post
        [0.Author.id] => 1
        [0.Author.user] => Kyle
        [1.Post.id] => 2
        [1.Post.title] => Second Post
        [1.Author.id] => 3
        [1.Author.user] => Crystal
    ]
*/
```

static Cake\Utility\Hash::**expand**(*array $data*, *string $separator = '.'*)

Développe un tableau qui a déjà été aplatie avec `Hash::flatten()` :

```
$data = [
    '0.Post.id' => 1,
    '0.Post.title' => First Post,
    '0.Author.id' => 1,
    '0.Author.user' => Kyle,
    '1.Post.id' => 2,
    '1.Post.title' => Second Post,
    '1.Author.id' => 3,
```

(suite sur la page suivante)

(suite de la page précédente)

```
    '1.Author.user' => Crystal,
];
$res = Hash::expand($data);
/* $res ressemble maintenant à:
[
    [
        'Post' => ['id' => '1', 'title' => 'First Post'],
        'Author' => ['id' => '1', 'user' => 'Kyle'],
    ],
    [
        'Post' => ['id' => '2', 'title' => 'Second Post'],
        'Author' => ['id' => '3', 'user' => 'Crystal'],
    ],
];
*/
```

static Cake\Utility\Hash::**merge**(*array $data*, *array $merge*)

Cette fonction peut être vue comme un hybride entre le `array_merge` et le `array_merge_recursive` de PHP. La différence entre les deux est que si une clé du tableau contient un autre tableau, alors la fonction se comporte de façon récursive (pas comme `array_merge`) mais ne le fait pas pour les clés contenant les chaînes de caractères (pas comme `array_merge_recursive`).

> **Note**
>
> Cette fonction va fonctionner avec un montant illimité d'arguments et convertit les paramètres de non-tableau en tableaux.

```
$array = [
    [
        'id' => '48c2570e-dfa8-4c32-a35e-0d71cbdd56cb',
        'name' => 'mysql raleigh-workshop-08 < 2008-09-05.sql ',
        'description' => 'Importing an sql dump'
    ],
    [
        'id' => '48c257a8-cf7c-4af2-ac2f-114ecbdd56cb',
        'name' => 'pbpaste | grep -i Unpaid | pbcopy',
        'description' => 'Remove all lines that say "Unpaid".',
    ]
];
$arrayB = 4;
$arrayC = [0 => "test array", "cats" => "dogs", "people" => 1267];
$arrayD = ["cats" => "felines", "dog" => "angry"];
$res = Hash::merge($array, $arrayB, $arrayC, $arrayD);

/* $res ressemble maintenant à:
[
    [0] => [
            [id] => 48c2570e-dfa8-4c32-a35e-0d71cbdd56cb
            [name] => mysql raleigh-workshop-08 < 2008-09-05.sql
            [description] => Importing an sql dump
    ]
```

(suite sur la page suivante)

(suite de la page précédente)

```
    [1] => [
            [id] => 48c257a8-cf7c-4af2-ac2f-114ecbdd56cb
            [name] => pbpaste | grep -i Unpaid | pbcopy
            [description] => Remove all lines that say "Unpaid".
    ]
    [2] => 4
    [3] => test array
    [cats] => felines
    [people] => 1267
    [dog] => angry
]
*/
```

static Cake\Utility\Hash::**numeric**(*array $data*)

Vérifie pour voir si toutes les valeurs dans le tableau sont numériques :

```
$data = ['one'];
$res = Hash::numeric(array_keys($data));
// $res est à true

$data = [1 => 'one'];
$res = Hash::numeric($data);
// $res est à false
```

static Cake\Utility\Hash::**dimensions**(*array $data*)

Compte les dimensions d'un tableau. Cette méthode va seulement considérer la dimension du premier élément dans le tableau :

```
$data = ['one', '2', 'three'];
$result = Hash::dimensions($data);
// $result == 1

$data = ['1' => '1.1', '2', '3'];
$result = Hash::dimensions($data);
// $result == 1

$data = ['1' => ['1.1' => '1.1.1'], '2', '3' => ['3.1' => '3.1.1']];
$result = Hash::dimensions($data);
// $result == 2

$data = ['1' => '1.1', '2', '3' => ['3.1' => '3.1.1']];
$result = Hash::dimensions($data);
// $result == 1

$data = ['1' => ['1.1' => '1.1.1'], '2', '3' => ['3.1' => ['3.1.1' => '3.1.1.1']]];
$result = Hash::dimensions($data);
// $result == 2
```

static Cake\Utility\Hash::**maxDimensions**(*array $data*)

Similaire à `dimensions()`, cependant cette méthode retourne le nombre le plus profond de dimensions de tout élément dans le tableau :

```
$data = ['1' => '1.1', '2', '3' => ['3.1' => '3.1.1']];
$result = Hash::maxDimensions($data);
// $result == 2

$data = ['1' => ['1.1' => '1.1.1'], '2', '3' => ['3.1' => ['3.1.1' => '3.1.1.1']]];
$result = Hash::maxDimensions($data);
// $result == 3
```

static Cake\Utility\Hash::**map**(*array $data*, *$path*, *$function*)

Crée un nouveau tableau, en extrayant `$path`, et mappe `$function` à travers les résultats. Vous pouvez utiliser les deux, expression et le matching d'éléments avec cette méthode :

```
// Appel de la fonction noop $this->noop() sur chaque element de $data
$result = Hash::map($data, "{n}", [$this, 'noop']);

public function noop(array $array)
{
    // Fait des choses au tableau et retourne les résultats
    return $array;
}
```

static Cake\Utility\Hash::**reduce**(*array $data*, *$path*, *$function*)

Crée une valeur unique, en extrayant `$path`, et en réduisant les résultats extraits avec `$function`. Vous pouvez utiliser les deux, expression et le matching d'éléments avec cette méthode.

static Cake\Utility\Hash::**apply**(*array $data*, *$path*, *$function*)

Appliquer un callback à un ensemble de valeurs extraites en utilisant `$function`. La fonction va récupérer les valeurs extraites en premier argument :

```
$data = [
    ['date' => '01-01-2016', 'booked' => true],
    ['date' => '01-01-2016', 'booked' => false],
    ['date' => '02-01-2016', 'booked' => true]
];
$result = Hash::apply($data, '{n}[booked=true].date', 'array_count_values');
/* $result ressemble maintenant à:
    [
        '01-01-2016' => 1,
        '02-01-2016' => 1,
    ]
*/
```

static Cake\Utility\Hash::**sort**(*array $data*, *$path*, *$dir*, *$type = 'regular'*)

Type renvoyé
array

Trie un tableau selon n'importe quelle valeur, déterminé par une *Syntaxe de chemin Hash*. Seuls les éléments de type expression sont supportés par cette méthode :

```
$a = [
    0 => ['Person' => ['name' => 'Jeff']],
    1 => ['Shirt' => ['color' => 'black']]
];
```

(suite sur la page suivante)

(suite de la page précédente)

```
$result = Hash::sort($a, '{n}.Person.name', 'asc');
/* $result ressemble maintenant à:
    [
        [0] => [
                [Shirt] => [
                        [color] => black
                ]
        ]
        [1] => [
                [Person] => [
                        [name] => Jeff
                ]
        ]
    ]
*/
```

`$dir` peut être soit `asc`, soit `desc`. Le `$type` peut être une des valeurs suivantes :
— `regular` pour le trier régulier.
— `numeric` pour le tri des valeurs avec leurs valeurs numériques équivalentes.
— `string` pour le tri des valeurs avec leur valeur de chaîne.
— `natural` pour trier les valeurs d'une façon humaine. Va trier `foo10` en-dessous de `foo2` par exemple.

static `Cake\Utility\Hash::`**`diff`**(*array $data*, *array $compare*)

Calcule la différence entre deux tableaux :

```
$a = [
    0 => ['name' => 'main'],
    1 => ['name' => 'about']
];
$b = [
    0 => ['name' => 'main'],
    1 => ['name' => 'about'],
    2 => ['name' => 'contact']
];

$result = Hash::diff($a, $b);
/* $result ressemble maintenant à:
    [
        [2] => [
                [name] => contact
        ]
    ]
*/
```

static `Cake\Utility\Hash::`**`mergeDiff`**(*array $data*, *array $compare*)

Cette fonction fusionne les deux tableaux et pousse les différences dans les données à la fin du tableau résultant.

Exemple 1

```
$array1 = ['ModelOne' => ['id' => 1001, 'field_one' => 'a1.m1.f1', 'field_two' =>
↪'a1.m1.f2']];
$array2 = ['ModelOne' => ['id' => 1003, 'field_one' => 'a3.m1.f1', 'field_two' =>
↪'a3.m1.f2', 'field_three' => 'a3.m1.f3']];
```

(suite sur la page suivante)

(suite de la page précédente)

```
$res = Hash::mergeDiff($array1, $array2);

/* $res ressemble maintenant à:
    [
        [ModelOne] => [
                [id] => 1001
                [field_one] => a1.m1.f1
                [field_two] => a1.m1.f2
                [field_three] => a3.m1.f3
            ]
    ]
*/
```

Exemple 2

```
$array1 = ["a" => "b", 1 => 20938, "c" => "string"];
$array2 = ["b" => "b", 3 => 238, "c" => "string", ["extra_field"]];
$res = Hash::mergeDiff($array1, $array2);
/* $res ressemble maintenant à:
    [
        [a] => b
        [1] => 20938
        [c] => string
        [b] => b
        [3] => 238
        [4] => [
                [0] => extra_field
        ]
    ]
*/
```

static Cake\Utility\Hash::**normalize**(*array $data*, *$assoc = true*)

Normalise un tableau. Si `$assoc` est à `true`, le tableau résultant sera normalisé en un tableau associatif. Les clés numériques avec les valeurs, seront convertis en clés de type chaîne avec des valeurs null. Normaliser un tableau, facilite l'utilisation des résultats avec *Hash::merge()* :

```
$a = ['Tree', 'CounterCache',
    'Upload' => [
        'folder' => 'products',
        'fields' => ['image_1_id', 'image_2_id']
    ]
];
$result = Hash::normalize($a);
/* $result ressemble maintenant à:
    [
        [Tree] => null
        [CounterCache] => null
        [Upload] => [
                [folder] => products
                [fields] => [
                        [0] => image_1_id
                        [1] => image_2_id
```

(suite sur la page suivante)

(suite de la page précédente)

```
                ]
        ]
    ]
*/

$b = [
    'Cacheable' => ['enabled' => false],
    'Limit',
    'Bindable',
    'Validator',
    'Transactional'
];
$result = Hash::normalize($b);
/* $result ressemble maintenant à:
    [
        [Cacheable] => [
                [enabled] => false
        ]

        [Limit] => null
        [Bindable] => null
        [Validator] => null
        [Transactional] => null
    ]
*/
```

static Cake\Utility\Hash::**nest**(*array $data*, *array $options = []*)

Prend un ensemble de tableau aplati, et crée une structure de données imbriquée ou chaînée.

Options :

- `children` Le nom de la clé à utiliser dans l'ensemble de résultat pour les enfants. Par défaut à "children".
- `idPath` Le chemin vers une clé qui identifie chaque entrée. Doit être compatible avec *Hash::extract()*. Par défaut à `{n}.$alias.id`
- `parentPath` Le chemin vers une clé qui identifie le parent de chaque entrée. Doit être compatible avec *Hash::extract()*. Par défaut à `{n}.$alias.parent_id`.
- `root` L'id du résultat le plus désiré.

Exemple :

```
$data = [
    ['ThreadPost' => ['id' => 1, 'parent_id' => null]],
    ['ThreadPost' => ['id' => 2, 'parent_id' => 1]],
    ['ThreadPost' => ['id' => 3, 'parent_id' => 1]],
    ['ThreadPost' => ['id' => 4, 'parent_id' => 1]],
    ['ThreadPost' => ['id' => 5, 'parent_id' => 1]],
    ['ThreadPost' => ['id' => 6, 'parent_id' => null]],
    ['ThreadPost' => ['id' => 7, 'parent_id' => 6]],
    ['ThreadPost' => ['id' => 8, 'parent_id' => 6]],
    ['ThreadPost' => ['id' => 9, 'parent_id' => 6]],
    ['ThreadPost' => ['id' => 10, 'parent_id' => 6]]
];

$result = Hash::nest($data, ['root' => 6]);
```

(suite sur la page suivante)

(suite de la page précédente)

```
/* $result ressemble maintenant à:
    [
        (int) 0 => [
            'ThreadPost' => [
                'id' => (int) 6,
                'parent_id' => null
            ],
            'children' => [
                (int) 0 => [
                    'ThreadPost' => [
                        'id' => (int) 7,
                        'parent_id' => (int) 6
                    ],
                    'children' => []
                ],
                (int) 1 => [
                    'ThreadPost' => [
                        'id' => (int) 8,
                        'parent_id' => (int) 6
                    ],
                    'children' => []
                ],
                (int) 2 => [
                    'ThreadPost' => [
                        'id' => (int) 9,
                        'parent_id' => (int) 6
                    ],
                    'children' => []
                ],
                (int) 3 => [
                    'ThreadPost' => [
                        'id' => (int) 10,
                        'parent_id' => (int) 6
                    ],
                    'children' => []
                ]
            ]
        ]
    ]
    */
```

CHAPITRE 34

Client Http

class Cake\Http**Client**(*mixed $config = []*)

CakePHP intègre un client HTTP basique respectant le standard PSR-18, que vous pouvez utiliser pour faire des requêtes. C'est un bon moyen de communiquer avec des services webs et des APIs distantes.

Faire des Requêtes

Faire des requêtes est simple et direct. Faire une requête GET ressemble à ceci :

```
use Cake\Http\Client;

$http = new Client();

// Simple GET
$response = $http->get('http://example.com/test.html');

// Simple GET avec querystring
$response = $http->get('http://example.com/search', ['q' => 'widget']);

// Simple GET avec querystring & headers supplémentaires
$response = $http->get('http://example.com/search', ['q' => 'widget'], [
  'headers' => ['X-Requested-With' => 'XMLHttpRequest']
]);
```

Faire des requêtes POST et PUT est tout aussi simple :

```
// Envoi d'une requête POST avec des données encodées en application/x-www-form-
↪urlencoded
$http = new Client();
```

(suite sur la page suivante)

(suite de la page précédente)

```
$response = $http->post('http://example.com/posts/add', [
  'title' => 'testing',
  'body' => 'content in the post'
]);

// Envoi d'une requête PUT avec des données encodées en application/x-www-form-urlencoded
$response = $http->put('http://example.com/posts/add', [
  'title' => 'testing',
  'body' => 'content in the post'
]);

// Autres méthodes.
$http->delete(/* ... */);
$http->head(/* ... */);
$http->patch(/* ... */);
```

Si vous avez créé un objet de requêtes PSR-7, vous pouvez l'envoyer avec `sendRequest()` :

```
use Cake\Http\Client;
use Cake\Http\Client\Request as ClientRequest;

$request = new ClientRequest(
    'http://example.com/search',
    ClientRequest::METHOD_GET
);
$client = new Client();
$response = $client->sendRequest($request);
```

Créer des Requêtes Multipart avec des Fichiers

Vous pouvez inclure des fichiers dans des corps de requête en incluant un gestionnaire de fichier dans le tableau de données :

```
$http = new Client();
$response = $http->post('http://example.com/api', [
  'image' => fopen('/path/to/a/file', 'r'),
]);
```

Le gestionnaire de fichiers sera lu jusqu'à sa fin ; il ne sera pas rembobiné avant d'être lu.

Construire des Corps de Requête Multipart

Il peut arriver que vous souhaitiez construire un corps de requête d'une façon très spécifique. Dans ces situations, vous pouvez utiliser `Cake\Http\Client\FormData` pour fabriquer la requête HTTP multipart spécifique que vous souhaitez :

```
use Cake\Http\Client\FormData;

$data = new FormData();

// Création d'une partie XML
$xml = $data->newPart('xml', $xmlString);
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Définit le type de contenu.
$xml->type('application/xml');
$data->add($xml);

// Création d'un upload de fichier avec addFile()
// Ceci va aussi ajouter le fichier aux données du formulaire.
$file = $data->addFile('upload', fopen('/some/file.txt', 'r'));
$file->contentId('abc123');
$file->disposition('attachment');

// Envoi de la requête.
$response = $http->post(
    'http://example.com/api',
    (string)$data,
    ['headers' => ['Content-Type' => $data->contentType()]]
);
```

Envoyer des Corps de Requête

Lorsque vous utilisez des API REST, vous avez souvent besoin d'envoyer des corps de requête qui ne sont pas encodés. Http\Client le permet grâce à l'option type :

```
// Envoi d'un body JSON.
$http = new Client();
$response = $http->post(
  'http://example.com/tasks',
  json_encode($data),
  ['type' => 'json']
);
```

La clé `type` peut être soit "json", soit "xml" ou bien un mime type complet. Quand vous utilisez l'option `type`, vous devrez fournir les données en chaîne de caractères. Si vous faites une requête GET qui a besoin à la fois de paramètres querystring et d'un corps de requête, vous pouvez faire ceci :

```
// Envoi d'un body JSON dans une requête GET avec des paramètres query string.
$http = new Client();
$response = $http->get(
  'http://example.com/tasks',
  ['q' => 'test', '_content' => json_encode($data)],
  ['type' => 'json']
);
```

Options de la Méthode Request

Chaque méthode HTTP prend un paramètre `$options` qui est utilisé pour fournir des informations de requête supplémentaires. les clés suivantes peuvent être utilisées dans `$options` :

- `headers` - Tableau de headers supplémentaires
- `cookie` - Tableau de cookies à utiliser.
- `proxy` - Tableau d'informations proxy.
- `auth` - Tableau de données d'authentification, la clé `type` est utilisée pour déléguer à une stratégie d'authentification. Par défaut c'est l'authentification Basic qui est utilisée.

- `ssl_verify_peer` - par défaut à `true`. Définie à `false` pour désactiver la certification SSL (non recommandé)
- `ssl_verify_peer_name` - par défaut à `true`. Définie à `false` pour désactiver la vérification du nom d'hôte lors des vérifications des certificats SSL (non recommandé).
- `ssl_verify_depth` - par défaut à 5. Profondeur de recherche dans la chaîne des autorités de certification (CA).
- `ssl_verify_host` - par défaut à `true`. Valide le certificat SSL au regard du nom d'hôte.
- `ssl_cafile` - par défaut le fichier d'autorités de certification intégré. Définissez cette option manuellement pour utiliser des autorités de certification personnalisées.
- `timeout` - Durée d'attente maximale en secondes.
- `type` - Envoie un corps de requête dans un type de contenu personnalisé. Nécessite que `$data` soit une chaîne ou que l'option `_content` soit définie quand vous faites des requêtes GET.
- `redirect` - Nombre de redirections à suivre. `false` par défaut.
- `curl` - Un tableau d'option supplémentaires pour curl (si l'adaptateur curl est utilisé). Par exemple `[CURLOPT_SSLKEY => 'key.pem']`.

Le paramètre options est toujours le 3ème paramètre dans chaque méthode HTTP. Elles peuvent aussi être utilisées en construisant `Client` pour créer des *clients scoped*.

Authentification

`Cake\Http\Client` supporte quelques systèmes d'authentification différents. Des stratégies d'authentification différentes peuvent être ajoutées par les développeurs. Les stratégies d'authentification sont appelées avant que la requête ne soit envoyée, et d'ajouter les headers au contexte de la requête.

Utiliser l'Authentication Basic

Un exemple simple d'authentification :

```
$http = new Client();
$response = $http->get('http://example.com/profile/1', [], [
  'auth' => ['username' => 'marc', 'password' => 'secret']
]);
```

Par défaut `Cake\Http\Client` va utiliser l'authentification basic s'il n'y a pas de clé `'type'` dans l'option auth.

Utiliser l'Authentification Digest

Un exemple simple d'authentification :

```
$http = new Client();
$response = $http->get('http://example.com/profile/1', [], [
  'auth' => [
    'type' => 'digest',
    'username' => 'marc',
    'password' => 'secret',
    'realm' => 'myrealm',
    'nonce' => 'valeurunique',
    'qop' => 1,
    'opaque' => 'unevaleur'
  ]
]);
```

En configurant la clé "type" à "digest", vous dites au sous-système d'authentification d'utiliser l'authentification digest.

Authentification OAuth 1

Plusieurs services web modernes nécessitent une authentication OAuth pour accéder à leur API. L'authentification OAuth incluse suppose que vous ayez déjà votre clé de consommateur et un secret de consommateur :

```
$http = new Client();
$response = $http->get('http://example.com/profile/1', [], [
  'auth' => [
    'type' => 'oauth',
    'consumerKey' => 'grandeclé',
    'consumerSecret' => 'secret',
    'token' => '...',
    'tokenSecret' => '...',
    'realm' => 'tickets',
  ]
]);
```

Authentification OAuth 2

Il n'y a pas d'adapteur d'authentification spécialisé car OAuth2 est souvent un simple en-tête. À la place, vous pouvez créer un client avec le token d'accès :

```
$http = new Client([
    'headers' => ['Authorization' => 'Bearer ' . $accessToken]
]);
$response = $http->get('https://example.com/api/profile/1');
```

Authentification Proxy

Certains proxies ont besoin d'une authentification pour les utiliser. Généralement cette authentification est Basic, mais elle peut être implémentée par n'importe quel adaptateur d'authentification. Par défaut, Http\Client va supposer une authentification Basic, à moins que la clé type ne soit définie :

```
$http = new Client();
$response = $http->get('http://example.com/test.php', [], [
  'proxy' => [
    'username' => 'marc',
    'password' => 'testing',
    'proxy' => '127.0.0.1:8080',
  ]
]);
```

Le deuxième paramètre du proxy doit être une chaîne avec une IP ou un domaine sans protocole. Le nom d'utilisateur et le mot de passe seront passés dans les en-têtes de la requête, alors que la chaîne du proxy sera passée dans stream_context_create() [183].

Créer des Clients Délimités (Scoped Clients)

Devoir retaper le nom de domaine, les paramètres d'authentification et de proxy peut devenir fastidieux et source d'erreurs. Pour réduire ce risque d'erreur et rendre l'exercice moins pénible, vous pouvez créer des clients délimités :

183. https://php.net/manual/en/function.stream-context-create.php

```
// Création d'un client délimité.
$http = new Client([
  'host' => 'api.example.com',
  'scheme' => 'https',
  'auth' => ['username' => 'marc', 'password' => 'testing']
]);

// Faire une requête vers api.example.com
$response = $http->get('/test.php');
```

Si votre client délimité a seulement besoin d'informations sur l'URL, vous pouvez utiliser `createFromUrl()` :

```
$http = Client::createFromUrl('https://api.example.com/v1/test');
```

Le code ci-dessus crée une instance client avec les options `protocol`, `host`, et `basePath` déjà définies.

Les informations suivantes peuvent être utilisées lors de la création d'un client délimité :

- host
- basepath
- scheme
- proxy
- auth
- port
- cookies
- timeout
- ssl_verify_peer
- ssl_verify_depth
- ssl_verify_host

Chacune de ces options peut être remplacée en les spécifiant quand vous faites des requêtes. host, scheme, proxy, port sont remplacées dans l'URL de la requête :

```
// Utilisation du client délimité que nous avons créé précédemment.
$response = $http->get('http://foo.com/test.php');
```

Le code ci-dessus va remplacer le domaine, le scheme, et le port. Cependant, cette requête va continuer à utiliser toutes les autres options définies quand le client délimité a été créé. Consultez *Options de la Méthode Request* pour plus d'informations sur les options intégrées.

Configurer et Gérer les Cookies

Http\Client peut aussi accepter les cookies quand on fait des requêtes. En plus d'accepter les cookies, il va aussi automatiquement stocker les cookies valides définis dans les réponses. À chaque réponse avec des cookies, ceux-ci seront stockés dans l'instance d'origine de Http\Client. Les cookies stockés dans une instance Client sont automatiquement inclus dans les futures requêtes qui correspondent au domaine + chemin :

```
$http = new Client([
    'host' => 'cakephp.org'
]);

// Création d'une requête qui définit des cookies
$response = $http->get('/');
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Les cookies de la première requête seront inclus par défaut.
$response2 = $http->get('/changelogs');
```

Vous pouvez toujours remplacer les cookies auto-inclus en les définissant dans les paramètres `$options` de la requête :

```
// Remplacement d'un cookie existant par une valeur personnalisée.
$response = $http->get('/changelogs', [], [
    'cookies' => ['sessionid' => '123abc']
]);
```

Vous pouvez ajouter des cookies au client après l'avoir créé en utilisant la méthode `addCookie()` :

```
use Cake\Http\Cookie\Cookie;

$http = new Client([
    'host' => 'cakephp.org'
]);
$http->addCookie(new Cookie('session', 'abc123'));
```

Objets Response

class Cake\Http\Client**Response**

Les objets Response ont un certain nombre de méthodes pour parcourir les données de réponse.

Lire les Corps des Réponses

Vous pouvez lire le corps entier de la réponse en chaîne de caractères :

```
// Lit le corps entier de la réponse en chaîne de caractères.
$response->getStringBody();
```

Vous pouvez aussi accéder à l'objet stream de la réponse et utiliser ses méthodes :

```
// Récupère une Psr\Http\Message\StreamInterface contenant le corps de la réponse
$stream = $response->getBody();

// Lit un stream par blocs de 100 bytes.
while (!$stream->eof()) {
    echo $stream->read(100);
}
```

Lire des Corps de Réponse JSON et XML

Puisque les réponses JSON et XML sont souvent utilisées, les objets response fournissent des accesseurs pour lire les données décodées. Les données JSON décodées sont fournies sous forme de tableau, tandis que les données XML sont décodées en un arbre `SimpleXMLElement` :

```
// Récupération du XML.
$http = new Client();
$response = $http->get('http://example.com/test.xml');
$xml = $response->getXml();
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Récupération du JSON.
$http = new Client();
$response = $http->get('http://example.com/test.json');
$json = $response->getJson();
```

Les données de réponse décodées sont stockées dans l'objet response, donc y accéder plusieurs fois n'augmente pas la charge.

Accéder aux En-têtes de la Réponse

Vous pouvez accéder aux en-têtes de différentes manières. Les noms de l'en-tête sont toujours traités comme des valeurs sensibles à la casse quand vous y accédez par une méthode :

```
// Récupère les en-têtes sous la forme d'un tableau associatif.
$response->getHeaders();

// Récupère un en-tête unique sous la forme d'un tableau.
$response->getHeader('content-type');

// Récupère un en-tête sous la forme d'une chaîne de caractères
$response->getHeaderLine('content-type');

// Récupère l'encodage de la réponse
$response->getEncoding();
```

Accéder aux Données des Cookies

Vous pouvez lire les cookies avec différentes méthodes selon la quantité de données que vous souhaitez sur les cookies :

```
// Récupère tous les cookies (toutes les données)
$response->getCookies();

// Récupère une valeur d'un cookie unique.
$response->getCookie('session_id');

// Récupère les données complètes pour un unique cookie
// includes value, expires, path, httponly, secure keys.
$response->getCookieData('session_id');
```

Vérifier le Code de Statut

Les objets Response fournissent quelques méthodes pour vérifier les codes de statuts :

```
// La réponse était-elle 20x
$response->isOk();

// La réponse était-elle 30x
$response->isRedirect();

// Récupère le code de statut
$response->getStatusCode();
```

Changer les Adaptateurs de Transport

Par défaut, `Http\Client` préférera utiliser un adaptateur de transport basé sur `curl`. Si l'extension curl n'est pas disponible, il utilisera à la place un adaptateur basé sur le stream. Vous pouvez forcer la sélection d'un adaptateur de transport en utilisant une option du constructeur :

```
use Cake\Http\Client\Adapter\Stream;

$client = new Client(['adapter' => Stream::class]);
```

Tests

trait Cake\Http\TestSuite**HttpClientTrait**

Dans les tests, vous voudrez souvent créer des réponses de mocks vers des API externes. Vous pouvez utiliser `HttpClientTrait` pour définir des réponses aux requêtes faites par votre application :

```
use Cake\Http\TestSuite\HttpClientTrait;
use Cake\TestSuite\TestCase;

class CartControllerTests extends TestCase
{
    use HttpClientTrait;

    public function testCheckout()
    {
        // Mocker une requête POST qui sera faite.
        $this->mockClientPost(
            'https://example.com/process-payment',
            $this->newClientResponse(200, [], json_encode(['ok' => true]))
        );
        $this->post("/cart/checkout");
        // Faire des assertions.
    }
}
```

Il existe des méthodes pour mocker les méthodes HTTP les plus courantes :

```
$this->mockClientGet(/* ... */);
$this->mockClientPatch(/* ... */);
$this->mockClientPost(/* ... */);
$this->mockClientPut(/* ... */);
$this->mockClientDelete(/* ... */);
```

Cake\Http\TestSuite\HttpClientTrait::**newClientResponse**(*int $code = 200*, *array $headers = []*, *string $body = ''*)

Comme vu précédemment, vous pouvez utiliser la méthode `newClientResponse()` pour créer des réponses pour les requêtes que fera votre application. Les en-têtes doivent être une liste de chaînes de caractères :

```
$headers = [
    'Content-Type: application/json',
    'Connection: close',
```

(suite sur la page suivante)

(suite de la page précédente)

```
];
$response = $this->newClientResponse(200, $headers, $body)
```

CHAPITRE 35

Inflector

class Cake\Utility**Inflector**

La classe Inflector prend une chaîne de caractères et peut la manipuler pour gérer les variations de mot comme les mises au pluriel ou les mises en Camel et est normalement accessible statiquement. Exemple : `Inflector::pluralize('example')` retourne « examples ».

Vous pouvez essayer les inflexions en ligne sur inflector.cakephp.org [184].

Résumé des Méthodes d'Inflector et de leurs Sorties

Petit résumé des méthodes intégrées à l'Inflector et des résultats produits lorsque vous passez plusieurs mots en argument :

184. https://inflector.cakephp.org/

Method	Argument	Output
pluralize()	BigApple	BigApples
	big_apple	big_apples
singularize()	BigApples	BigApple
	big_apples	big_apple
camelize()	big_apples	BigApples
	big apple	BigApple
underscore()	BigApples	big_apples
	Big Apples	big apples
humanize()	big_apples	Big Apples
	bigApple	BigApple
classify()	big_apples	BigApple
	big apple	BigApple
dasherize()	BigApples	big-apples
	big apple	big apple
tableize()	BigApple	big_apples
	Big Apple	big apples
variable()	big_apple	bigApple
	big apples	bigApples
slug()	Big Apple	big-apple
	BigApples	BigApples

Créer des Formes Pluriel et Singulier

static Cake\Utility\Inflector::**singularize**(*$singular*)

static Cake\Utility\Inflector::**pluralize**(*$singular*)

pluralize et singularize() fonctionnent pour la plupart des noms Anglais. Si vous devez supporter d'autres langues, vous pouvez utiliser la *Configuration d'Inflexion* pour personnaliser les règles utilisées :

```
// Apples
echo Inflector::pluralize('Apple');
```

> **Note**
>
> pluralize() peut ne pas toujours convertir correctement un nom qui est déjà sous sa forme plurielle.

```
// Person
echo Inflector::singularize('People');
```

> **Note**
>
> singularize() peut ne pas toujours convertir correctement un nom qui est déjà sous sa forme singulière.

Créer des Formes en CamelCase et en Underscore

static Cake\Utility\Inflector::**camelize**(*$underscored*)

static Cake\Utility\Inflector::**underscore**(*$camelCase*)

Ces méthodes sont utiles lors de la création de noms de classes ou de propriétés :

```
// ApplePie
Inflector::camelize('Apple_pie')

// apple_pie
Inflector::underscore('ApplePie');
```

Il doit être noté que les underscores vont seulement convertir les mots formatés en camelCase. Les mots qui contiennent des espaces seront en minuscules, mais ne contiendront pas d'underscore.

Créer des Formes Lisibles par l'Homme

Cette méthode est utile pour convertir des formes avec underscore en forme « Title Case » pour être lisible par l'homme :

```
// Apple Pie
Inflector::humanize('apple_pie');
```

Créer des Formes pour les Tables et les Noms de Classe

static Cake\Utility\Inflector::**classify**(*$underscored*)

static Cake\Utility\Inflector::**dasherize**(*$dashed*)

static Cake\Utility\Inflector::**tableize**(*$camelCase*)

Quand vous générez du code ou quand vous utilisez les conventions de CakePHP, vous pouvez infléchir les noms de table ou les noms de classe :

```
// UserProfileSettings
Inflector::classify('user_profile_settings');

// user-profile-setting
Inflector::dasherize('UserProfileSetting');

// user_profile_settings
Inflector::tableize('UserProfileSetting');
```

Créer des Noms de Variable

static Cake\Utility\Inflector::**variable**(*$underscored*)

Les noms de variable sont souvent utiles quand vous faîtes des tâches meta-programming qui impliquent la génération de code ou des opérations basées sur les conventions :

```
// applePie
Inflector::variable('apple_pie');
```

Configuration d'Inflexion

Les conventions de nommage de CakePHP peuvent être très sympas - vous pouvez nommer votre table de base de données `big\_boxes`, votre model `BigBoxes`, votre controller `BigBoxesController`, et tout fonctionnera automatiquement. CakePHP connaît la façon dont les choses sont liées grâce à l'*inflexion* des mots entre leurs formes singulière et plurielle.

Il existe des cas (spécialement pour nos amis non-anglais) où l'inflector de CakePHP (la classe qui pluralise, singularise, met en camelCase et en underscore) ne fonctionnera pas comme vous le souhaitez. Si CakePHP ne reconnaîtra pas votre Foci ou Fish, vous pouvez dire à CakePHP vos cas spécifiques.

Charger les Inflexions Personnalisées

static `Cake\Utility\Inflector::`**rules**(*$type*, *$rules*, *$reset = false*)

Définit une nouvelle inflexion et des règles de transliteration que Inflector va utiliser. Souvent, cette méthode est utilisée dans votre **config/bootstrap.php** :

```
Inflector::rules('singular', ['/^(bil)er$/i' => '\1', '/^(inflec|contribu)tors$/i' => '\
↪1ta']);
Inflector::rules('uninflected', ['singulars']);
Inflector::rules('irregular', ['phylum' => 'phyla']); // The key is singular form, value␣
↪is plural form
```

Les règles fournies vont être fusionnées dans l'ensemble d'inflexion défini dans `Cake/Utility/Inflector`, avec les règles ajoutées qui supplantent les règles du coeur. Vous pouvez utiliser `Inflector::reset()` pour nettoyer les règles et restaurer l'état d'Inflector originel.

CHAPITRE 36

Number

class Cake\I18n**Number**

Si vous avez besoin des fonctionnalités de `NumberHelper` en-dehors d'une `View`, utilisez la classe `Number` :

```
namespace App\Controller;

use Cake\I18n\Number;

class UsersController extends AppController
{

    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('Auth');
    }

    public function afterLogin()
    {
        $storageUsed = $this->Auth->user('storage_used');
        if ($storageUsed > 5000000) {
            // Notify users of quota
            $this->Flash->success(__('You are using {0} storage', Number::toReadableSize(
↪$storageUsed)));
        }
    }
}
```

Toutes ces fonctions retournent le nombre formaté ; Elles n'affichent pas automatiquement la sortie dans la vue.

Formatage des Devises

`Cake\I18n\Number::currency(mixed $value, string $currency = null, array $options = [])`

Cette méthode est utilisée pour afficher un nombre dans des formats de monnaie courante (EUR,GBP,USD). L'utilisation dans une vue ressemble à ceci :

```
// Appelé avec NumberHelper
echo $this->Number->currency($value, $currency);

// Appelé avec Number
echo CakeNumber::currency($value, $currency);
```

Le premier paramètre `$value`, doit être un nombre décimal qui représente le montant d'argent que vous désirez. Le second paramètre est utilisé pour choisir un schéma de formatage de monnaie courante :

$currency	1234.56, formaté par le type courant
EUR	€1.234,56
GBP	£1,234.56
USD	$1,234.56

Le troisième paramètre est un tableau d'options pour définir la sortie. Les options suivantes sont disponibles :

Option	Description
before	Chaine de caractères à placer avant le nombre.
after	Chaine de caractères à placer après le nombre.
zero	Le texte à utiliser pour des valeurs à zéro, peut être une chaîne de caractères ou un nombre. ex : 0, "Free !"
places	Nombre de décimales à utiliser. ex : 2
precision	Nombre maximal de décimale à utiliser. ex :2
locale	Le nom de la locale utilisée pour formatter le nombre, ie. « fr_FR ».
fractionSymbol	Chaîne de caractères à utiliser pour les nombres en fraction. ex : " cents"
fractionPosition	Soit "before" soit "after" pour placer le symbole de fraction
pattern	Un modèle de formatage ICU à utiliser pour formatter le nombre. ex : #,###.00
useIntlCode	Mettre à `true` pour remplacer le symbole monétaire par le code monétaire international

Si la valeur de $currency est `null`, la devise par défaut est récupérée par *`Cake\I18n\Number::defaultCurrency()`*.

Paramétrage de la Devise par Défaut

`Cake\I18n\Number::defaultCurrency(string $currency)`

Setter/getter pour la monnaie par défaut. Ceci retire la nécessité de toujours passer la monnaie à *`Cake\I18n\Number::currency()`* et change toutes les sorties de monnaie en définissant les autres par défaut. Si `$currency` est `false`, cela effacera la valeur actuellement enregistrée. Par défaut, cette fonction retourne la valeur `intl.default_locale` si définie et "en_US" sinon.

Formatage Des Nombres A Virgules Flottantes

`Cake\I18n\Number::precision`(*float $value*, *int $precision = 3*, *array $options = []*)

Cette méthode affiche un nombre avec la précision spécifiée (place de la décimale). Elle arrondira afin de maintenir le niveau de précision défini :

```
// Appelé avec NumberHelper
echo $this->Number->precision(456.91873645, 2 );

// Sortie
456.92

// Appelé avec Number
echo Number::precision(456.91873645, 2 );
```

Formatage Des Pourcentages

`Cake\I18n\Number::toPercentage`(*mixed $value*, *int $precision = 2*, *array $options = []*)

Option	Description
multiply	Booléen pour indiquer si la valeur doit être multipliée par 100. Utile pour les pourcentages avec décimale.

Comme *Cake\I18n\Number::precision()*, cette méthode formate un nombre selon la précision fournie (où les nombres sont arrondis pour parvenir à ce degré de précision). Cette méthode exprime aussi le nombre en tant que pourcentage et ajoute un signe de pourcent à la sortie :

```
// appelé avec NumberHelper. Sortie: 45.69%
echo $this->Number->toPercentage(45.691873645);

// appelé avec Number. Sortie: 45.69%
echo Number::toPercentage(45.691873645);

// Appelé avec multiply. Sortie: 45.69%
echo Number::toPercentage(0.45691, 2, [
    'multiply' => true
]);
```

Interagir Avec Des Valeurs Lisibles Par L'Homme

`Cake\I18n\Number::toReadableSize`(*string $dataSize*)

Cette méthode formate les tailles de données dans des formes lisibles pour l'homme. Elle fournit une manière raccourcie de convertir les en KB, MB, GB, et TB. La taille est affichée avec un niveau de précision à deux chiffres, selon la taille de données fournie (ex : les tailles supérieurs sont exprimées dans des termes plus larges) :

```
// Appelé avec NumberHelper
echo $this->Number->toReadableSize(0); // 0 Byte
echo $this->Number->toReadableSize(1024); // 1 KB
```

(suite sur la page suivante)

(suite de la page précédente)

```
echo $this->Number->toReadableSize(1321205.76); // 1.26 MB
echo $this->Number->toReadableSize(5368709120); // 5 GB

// Appelé avec Number
echo Number::toReadableSize(0); // 0 Byte
echo Number::toReadableSize(1024); // 1 KB
echo Number::toReadableSize(1321205.76); // 1.26 MB
echo Number::toReadableSize(5368709120); // 5 GB
```

Formatage Des Nombres

`Cake\I18n\Number::format(mixed $value, array $options=[])`

Cette méthode vous donne beaucoup plus de contrôle sur le formatage des nombres pour l'utilisation dans vos vues (et est utilisée en tant que méthode principale par la plupart des autres méthodes de NumberHelper). L'utilisation de cette méthode pourrait ressembler à cela :

```
// Appelé avec NumberHelper
$this->Number->format($value, $options);

// Appelé avec Number
Number::format($value, $options);
```

Le paramètre `$value` est le nombre que vous souhaitez formater pour la sortie. Avec aucun `$options` fourni, le nombre 1236.334 sortirait comme ceci : 1,236. Notez que la précision par défaut est d'aucun chiffre après la virgule.

Le paramètre `$options` est là où réside la réelle magie de cette méthode.

- Si vous passez un entier alors celui-ci devient le montant de précision pour la fonction.
- Si vous passez un tableau associatif, vous pouvez utiliser les clés suivantes :

Option	Description
places	Nombre de décimales à utiliser. ex : 2
precision	Nombre maximal de décimale à utiliser. ex :2
pattern	Un modèle de formatage ICU à utiliser pour formatter le nombre. ex : #,###.00
locale	Le nom de la locale utilisée pour formatter le nombre, ie. « fr_FR ».
before	Chaine de caractères à placer avant le nombre.
after	Chaine de caractères à placer après le nombre.

Exemple :

```
// Appelé avec NumberHelper
echo $this->Number->format('123456.7890', [
    'places' => 2,
    'before' => '¥ ',
    'after' => ' !'
]);
// Sortie '¥ 123,456.79 !'

echo $this->Number->format('123456.7890', [
    'locale' => 'fr_FR'
]);
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Sortie '123 456,79 !'

// Appelé avec Number
echo Number::format('123456.7890', [
    'places' => 2,
    'before' => '¥ ',
    'after' => ' !'
]);
// Sortie '¥ 123,456.79 !'

echo Number::format('123456.7890', [
    'locale' => 'fr_FR'
]);
// Sortie '123 456,79 !'
```

`Cake\I18n\Number::ordinal`(*mixed $value*, *array $options = []*)

Cette méthode va afficher un nombre ordinal.

Exemples :

```
echo Number::ordinal(1);
// Affiche '1st'

echo Number::ordinal(2);
// Affiche '2nd'

echo Number::ordinal(2, [
    'locale' => 'fr_FR'
]);
// Affiche '2e'

echo Number::ordinal(410);
// Affiche '410th'
```

Formatage Des Différences

`Cake\I18n\Number::formatDelta`(*mixed $value*, *mixed $options=[]*)

Cette méthode affiche les différences en valeur comme un nombre signé :

```
// Appelé avec NumberHelper
$this->Number->formatDelta($value, $options);

// Appelé avec Number
Number::formatDelta($value, $options);
```

Le paramètre `$value` est le nombre que vous planifiez sur le formatage de sortie. Avec aucun `$options` fourni, le nombre 1236.334 sortirait 1,236. Notez que la valeur de precision par défaut est aucune décimale.

Le paramètre `$options` prend les mêmes clés que *`Number::format()`* lui-même :

Option	Description
places	Nombre de décimales à utiliser. ex : 2
precision	Nombre maximal de décimale à utiliser. ex :2
pattern	Un modèle de formatage ICU à utiliser pour formatter le nombre. ex : #,###.00
locale	Le nom de la locale utilisée pour formatter le nombre, ie. « fr_FR ».
before	Chaine de caractères à placer avant le nombre.
after	Chaine de caractères à placer après le nombre.

Example :

```
// Appelé avec NumberHelper
echo $this->Number->formatDelta('123456.7890', [
    'places' => 2,
    'before' => '[',
    'after' => ']'
]);
// Sortie '[+123,456.79]'

// Appelé avec Number
echo Number::formatDelta('123456.7890', [
    'places' => 2,
    'before' => '[',
    'after' => ']'
]);
// Sortie '[+123,456.79]'
```

Configurer le Formatage

`Cake\I18n\Number::config`(*string $locale*, *int $type = NumberFormatter : :DECIMAL*, *array $options = []*)

Cette méthode vous permet de configurer le formatage par défaut qui sera utilisé de façon persistante à travers toutes les méthodes.

Par exemple :

```
Number::config('en_IN', \NumberFormatter::CURRENCY, [
    'pattern' => '#,##,##0'
]);
```

CHAPITRE 37

Objets Registry

Les classes registry sont une façon simple de créer et récupérer les instances chargées d'un type d'objet donné. Il y a des classes registry pour les Components, les Helpers, les Tasks et les Behaviors.

Dans les exemples ci-dessous, on va utiliser les Components, mais le même comportement peut être attendu pour les Helpers, les Behaviors et les Tasks en plus des Components.

Charger les Objets

Les objets peuvent être chargés à la volée en utilisant add<registry-object>() Exemple :

```
$this->loadComponent('Acl.Acl');
$this->addHelper('Flash')
```

Va permettre de charger la propriété `Toolbar` et le helper `Flash`. La configuration peut aussi être définie à la volée. Exemple :

```
$this->loadComponent('Cookie', ['name' => 'sweet']);
```

Toutes clés & valeurs fournies vont être passées au constructeur du Component La seule exception à cette règle est `className`. Classname est une clé spéciale qui est utilisée pour faire des alias des objets dans un registry. Cela vous permet d'avoir des noms de component qui ne correspondent pas aux noms de classes, ce qui peut être utile quand vous étendez les components du cœur :

```
$this->Auth = $this->loadComponent('Auth', ['className' => 'MyCustomAuth']);
$this->Auth->user(); // Utilise en fait MyCustomAuth::user();
```

Attraper les Callbacks

Les Callbacks ne sont pas fournis par les objets registry. Vous devez utiliser les *events system* pour dispatcher tout events/callbacks dans votre application.

Désactiver les Callbacks

Dans les versions précédentes, les objets collection fournissaient une méthode `disable()` pour désactiver les objets à partir des callbacks reçus. Pour le faire maintenant, vous devez utiliser les fonctionnalités dans le système d'événements. Par exemple, vous pouvez désactiver les callbacks du component de la façon suivante :

```
// Retire Auth des callbacks.
$this->getEventManager()->off($this->Auth);

// Re-active Auth pour les callbacks.
$this->getEventManager()->on($this->Auth);
```

CHAPITRE 38

Text

class Cake\Utility**Text**

La classe Text inclut des méthodes pratiques pour créer et manipuler des chaînes de caractères et est normalement accessible statiquement. Exemple : `Text::uuid()`.

Si vous avez besoin des fonctionnalités de `TextHelper` en-dehors d'une `View`, utilisez la classe `Text` :

```
namespace App\Controller;

use Cake\Utility\Text;

class UsersController extends AppController
{

    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('Auth');
    }

    public function afterLogin()
    {
        $message = $this->Users->find('new_message');
        if (!empty($message)) {
            // notifie un nouveau message à l'utilisateur
            $this->FLash->success(__(
                'Vous avez un message: {0}',
                Text::truncate($message['Message']['body'], 255, ['html' => true])
            ));
        }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }
}
```

Convertion des Chaînes de Caractères en ASCII

static Cake\Utility\Text::**transliterate**(*$string*, *$transliteratorId = null*)

La méthode `translisterate` convertit par défaut tous les caractères dans la chaîne soumise dans son équivalent en caractères ASCII. La méthode s'attend à avoir une chaîne encodée en UTF-8 en entrée. La conversion des caractères peut être controllée en utilisant des identifiants de translitération que vous passez via l'argument `$transliteratorId` de la méthode ou en changeant l'identifiant par défaut à l'aide de `Text::setTransliteratorId()`. Les identifiants de translitération d'ICU sont généralement sous la forme `<source script>:<target script>` et vous pouvez spécifier plusieurs couples de conversion en les séparant par des `;`. Vous trouverez plus d'informations sur les identifiants de transliterateurs ici [185] :

```
// apple purée
Text::transliterate('apple purée');

// Ubermensch (seuls les caractères latin seront translitérés)
Text::transliterate('Übérmensch', 'Latin-ASCII;');
```

Créer des chaînes saines pour les URL

static Cake\Utility\Text::**slug**(*$string*, *$options = []*)

La méthode `slug` va translitérer tous les caractères en leurs équivalents ASCII et convertir tous les caractères non reconnus ainsi que les espaces en trait d'union (-). Elle s'attend à recevoir une chaîne encodée en UTF-8 en entrée.

Vous pouvez lui passer un tableau d'options pour avoir plus de contrôle sur le slug retourné. Le paramètre `$options` peut aussi être une chaîne de caractères, auquel cas il sera utilisé comme caractère de remplacement. Les options supporté sont :

- `replacement` La chaîne de remplacement, le défaut étant "-".
- **`transliteratorId` Un identifiant translitérateur valide.**
  S'il vaut `null` (valeur par défaut), `Text::$_defaultTransliteratorId` sera utilisé. S'il vaut `false`, aucune translitération ne sera faite. Seul les sous-chaînes qui ne sont pas des lettres/chiffres seront supprimées.
- **`preserve` Les caractères qui ne sont pas des lettres ou des chiffres à**
  préserver. Vaut `null` par défaut. Par exemple, cette option peut être définie à "." pour générer des noms de fichiers propres :

  ```
  // apple-puree
  Text::slug('apple purée');

  // apple_puree
  Text::slug('apple purée', '_');

  // foo-bar.tar.gz
  Text::slug('foo bar.tar.gz', ['preserve' => '.']);
  ```

185. https://unicode-org.github.io/icu/userguide/transforms/general/#transliterator-identifiers

Générer des UUIDs

static Cake\Utility\Text::**uuid**

La méthode UUID est utilisée pour générer des identificateurs uniques comme per **RFC 4122** [186]. UUID est une chaîne de caractères de 128-bit au format `485fc381-e790-47a3-9794-1337c0a8fe68` :

```
Text::uuid(); // 485fc381-e790-47a3-9794-1337c0a8fe68
```

Parseur de chaînes simples

static Cake\Utility\Text::**tokenize**(*$data*, *$separator = ', '*, *$leftBound = '('*, *$rightBound = ')'*)

Tokenizes une chaîne en utilisant `$separator`, en ignorant toute instance de `$separator` qui apparait entre `$leftBound` et `$rightBound`.

Cette méthode peut être utile quand on sépare les données en formatage régulier comme les listes de tag :

```
$data = "cakephp 'great framework' php";
$result = Text::tokenize($data, ' ', "'", "'");
// le résultat contient
['cakephp', "'great framework'", 'php'];
```

Cake\Utility\Text::**parseFileSize**(*string $size*, *$default*)

Cette méthode enlève le format d'un nombre à partir d'une taille de byte lisible par un humain en un nombre entier de bytes :

```
$int = Text::parseFileSize('2GB');
```

Formater une chaîne

static Cake\Utility\Text::**insert**(*$string*, *$data*, *$options = []*)

La méthode insérée est utilisée pour créer des chaînes templates et pour permettre les remplacements de clé/valeur :

```
Text::insert("Mon nom est :name et j'ai :age ans.", ['name' => 'Bob', 'age' => '65']);
// génère: "Mon nom est Bob et j'ai 65 ans."
```

static Cake\Utility\Text::**cleanInsert**(*$string*, *$options = []*)

Nettoie une chaîne formatée `Text::insert` avec `$options` donnée qui dépend de la clé "clean" dans `$options`. La méthode par défaut utilisée est le texte mais html est aussi disponible. Le but de cette fonction est de remplacer tous les espaces blancs et les balises non nécessaires autour des placeholders qui ne sont pas remplacés par Set : :insert.

Vous pouvez utiliser les options suivantes dans le tableau options :

```
$options = [
    'clean' => [
        'method' => 'text', // ou html
    ],

    'before' => '',
```

(suite sur la page suivante)

186. https://datatracker.ietf.org/doc/html/rfc4122.html

(suite de la page précédente)

```
    'after' => ''
];
```

Fixer la largeur d'un texte

static Cake\Utility\Text::**wrap**(*$text*, *$options = []*)

Entoure un block de texte pour un ensemble de largeur, et indente aussi les blocks. Peut entourer intelligemment le texte ainsi les mots ne sont pas coupés d'une ligne à l'autre :

```
$text = 'Ceci est la chanson qui ne stoppe jamais.';
$result = Text::wrap($text, 22);

// retourne
Ceci est la chanson
qui ne stoppe jamais.
```

Vous pouvez fournir un tableau d'options qui contrôlent la façon dont on entoure. Les options possibles sont :

- `width` La largeur de l'enroulement. Par défaut à 72.
- `wordWrap` Entoure ou non les mots entiers. Par défaut à `true`.
- `indent` Le caractère avec lequel on indente les lignes. Par défaut à "".
- `indentAt` Le nombre de ligne pour commencer l'indentation du texte. Par défaut à 0.

static Cake\Utility\Text::**wrapBlock**(*$text*, *$options = []*)

Si vous devez vous assurer que la largeur totale du bloc généré ne dépassera pas une certaine largeur y compris si elle contient des indentations, vous devez utiliser `wrapBlock()` au lieu de `wrap()`. C'est particulièrement utile pour générer du texte dans la console par exemple. Elle accepte les mêmes options que `wrap()` :

```
$text = 'Ceci est la chanson qui ne stoppe jamais. Ceci est la chanson qui ne stoppe␣
↪jamais.';
$result = Text::wrapBlock($text, [
    'width' => 22,
    'indent' => ' → ',
    'indentAt' => 1
]);

// Génère
Ceci est la chanson
 → qui ne stoppe
 → jamais. Ceci est
 → la chanson qui ne
 → stoppe jamais.
```

Subrillance de Sous-Chaîne

Cake\Utility\Text::**highlight**(*string $haystack*, *string $needle*, *array $options = []*)

Mettre en avant `$needle` dans `$haystack` en utilisant la chaîne spécifique `$options['format']` ou une chaîne par défaut.

Options :

- `format` - chaîne la partie de html avec laquelle la phrase sera mise en exergue.

— `html` - booléen Si `true`, va ignorer tous les tags HTML, s'assurant que seul le bon texte est mise en avant.

Exemple :

```
// appelé avec TextHelper
echo $this->Text->highlight(
    $lastSentence,
    'using',
    ['format' => '<span class="highlight">\1</span>']
);

// appelé avec Text
use Cake\Utility\Text;

echo Text::highlight(
    $lastSentence,
    'using',
    ['format' => '<span class="highlight">\1</span>']
);
```

Sortie :

```
Highlights $needle in $haystack <span class="highlight">using</span> the
$options['format'] string specified or a default string.
```

Retirer les Liens

`Cake\Utility\Text::stripLinks(`*$text*`)`

Enlève le `$text` fourni de tout lien HTML.

Tronquer le Texte

`Cake\Utility\Text::truncate(`*string $text*, *int $length = 100*, *array $options*`)`

Si `$text` est plus long que `$length`, cette méthode le tronque à la longueur `$length` et ajoute un suffixe `'ellipsis'`, si défini. Si `'exact'` est passé à `false`, le truchement va se faire au premier espace après le point où `$length` a dépassé. Si `'html'` est passé à `true`, les balises html seront respectés et ne seront pas coupés.

`$options` est utilisé pour passer tous les paramètres supplémentaires, et a les clés suivantes possibles par défaut, celles-ci étant toutes optionnelles :

```
[
    'ellipsis' => '...',
    'exact' => true,
    'html' => false
]
```

Exemple :

```
// appelé avec TextHelper
echo $this->Text->truncate(
    'The killer crept forward and tripped on the rug.',
    22,
```

(suite sur la page suivante)

(suite de la page précédente)

```
    [
        'ellipsis' => '...',
        'exact' => false
    ]
);

// appelé avec Text
App::uses('Text', 'Utility');
echo Text::truncate(
    'The killer crept forward and tripped on the rug.',
    22,
    [
        'ellipsis' => '...',
        'exact' => false
    ]
);
```

Sortie :

```
The killer crept...
```

Tronquer une chaîne par la fin

`Cake\Utility\Text::tail`(*string $text*, *int $length = 100*, *array $options*)

Si `$text` est plus long que `$length`, cette méthode retire une sous-chaîne initiale avec la longueur de la différence et ajoute un préfixe `'ellipsis'`, s'il est défini. Si `'exact'` est passé à `false`, le truchement va se faire au premier espace avant le moment où le truchement aurait été fait.

`$options` est utilisé pour passer tous les paramètres supplémentaires, et a les clés possibles suivantes par défaut, toutes sont optionnelles :

```
[
    'ellipsis' => '...',
    'exact' => true
]
```

Exemple :

```
$sampleText = 'I packed my bag and in it I put a PSP, a PS3, a TV, ' .
    'a C# program that can divide by zero, death metal t-shirts'

// appelé avec TextHelper
echo $this->Text->tail(
    $sampleText,
    70,
    [
        'ellipsis' => '...',
        'exact' => false
    ]
);
```

(suite sur la page suivante)

(suite de la page précédente)

```
// appelé avec Text
App::uses('Text', 'Utility');
echo Text::tail(
    $sampleText,
    70,
    [
        'ellipsis' => '...',
        'exact' => false
    ]
);
```

Sortie :

```
...a TV, a C# program that can divide by zero, death metal t-shirts
```

Générer un Extrait

`Cake\Utility\Text::excerpt`(*string $haystack*, *string $needle*, *integer $radius=100*, *string $ellipsis="..."*)

Génère un extrait de `$haystack` entourant le `$needle` avec un nombre de caractères de chaque côté déterminé par `$radius`, et préfixé/suffixé avec `$ellipsis`. Cette méthode est spécialement pratique pour les résultats de recherches. La chaîne requêtée ou les mots clés peuvent être montrés dans le document résultant :

```
// appelé avec TextHelper
echo $this->Text->excerpt($lastParagraph, 'method', 50, '...');

// appelé avec Text
use Cake\Utility\Text;

echo Text::excerpt($lastParagraph, 'méthode', 50, '...');
```

Génère : :

…$radius,et préfixé/suffixé avec $ellipsis. Cette méthode est spécialement pratique pour les résultats de r…

Convertir un tableau sous la forme d'une phrase

`Cake\Utility\Text::toList`(*array $list*, *$and='and'*, *$separator=', '*)

Crée une liste séparée avec des virgules, où les deux derniers items sont joins avec "and" :

```
$colors = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet'];

// appelé avec TextHelper
echo $this->Text->toList($colors);

// appelé avec Text
use Cake\Utility\Text;

echo Text::toList($colors);
```

Sortie :

```
red, orange, yellow, green, blue, indigo et violet
```

CHAPITRE 39

Date & Time

class Cake\I18n**FrozenTime**

Si vous avez besoin de fonctionnalités `TimeHelper` en-dehors d'une `View`, utilisez la classe `FrozenTime` :

```
use Cake\I18n\FrozenTime;

class UsersController extends AppController
{
    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('Auth');
    }

    public function afterLogin()
    {
        $time = new FrozenTime($this->Auth->user('date_de_naissance'));
        if ($time->isToday()) {
            // accueillir l'utilisateur avec un message de bon anniversaire
            $this->Flash->success(__('Bon anniversaire à toi...'));
        }
    }
}
```

En interne, CakePHP utilise Chronos [187] pour faire fonctionner l'utilitaire `FrozenTime`. Tout ce que vous pouvez faire avec `Chronos` et `DateTime`, vous pouvez le faire avec `FrozenTime` et `FrozenDate`.

Pour plus d'informations sur Chronos, rendez-vous sur la documentation de l'API [188].

187. https://github.com/cakephp/chronos
188. https://api.cakephp.org/chronos/1.0/

Créer des Instances FrozenTime

Les objets `FrozenTime` ne sont pas modifiables. Ils sont utiles quand vous voulez éviter des changements accidentels de données, ou lorsque le bon fonctionnement dépend de l'ordre des traitements. Reportez-vous aux instances `Time` pour les objets modifiables.

Il y a plusieurs façons de créer des instances `Time` :

```
use Cake\I18n\FrozenTime;

// Crée à partir d'une chaîne datetime.
$time = FrozenTime::createFromFormat(
    'Y-m-d H:i:s',
    '2021-01-31 22:11:30',
    'America/New_York'
);

// Crée à partir d'un timestamp
$time = FrozenTime::createFromTimestamp(1612149090, 'America/New_York');

// Récupère le temps actuel.
$time = FrozenTime::now();

// Ou utilise juste 'new'
$time = new FrozenTime('2014-01-10 11:11', 'America/New_York');

$time = new FrozenTime('2 hours ago');
```

Le constructeur de la classe `FrozenTime` peut prendre les mêmes paramètres que la classe PHP interne `DateTimeImmutable`. Quand vous passez un nombre ou une valeur numérique, elle sera interprétée comme un timestamp UNIX.

Dans les cas de test, vous pouvez mock out `now()` en utilisant `setTestNow()` :

```
// Fixe le temps.
$time = new FrozenTime('2021-01-31 22:11:30');
FrozenTime::setTestNow($time);

// Affiche '2021-01-31 22:11:30'
$now = FrozenTime::now();
echo $now->i18nFormat('yyyy-MM-dd HH:mm:ss');

// Affiche '2021-01-31 22:11:30'
$now = FrozenTime::parse('now');
echo $now->i18nFormat('yyyy-MM-dd HH:mm:ss');
```

Manipulation

Souvenez-vous que quand ses setters sont appelés, une instance `FrozenTime` renvoie toujours une autre instance plutôt que de se modifier elle-même :

```
$time = FrozenTime::now();

// Créer et réassigner une nouvelle instance
```

(suite sur la page suivante)

(suite de la page précédente)

```
$newTime = $time->year(2013)
    ->month(10)
    ->day(31);

// Affiche '2013-10-31 22:11:30'
echo $newTime->i18nFormat('yyyy-MM-dd HH:mm:ss');
```

Vous pouvez aussi utiliser les méthodes fournies nativement par la classe PHP `DateTime` :

```
$time = $time->setDate(2013, 10, 31);
```

Si vous ne réassignez pas la nouvelle instance `FrozenTime`, la variable contiendra toujours l'instance originale non modifiée :

```
$time->year(2013)
    ->month(10)
    ->day(31);
// Affiche '2021-01-31 22:11:30'
echo $time->i18nFormat('yyyy-MM-dd HH:mm:ss');
```

Vous pouvez créer une autre instance avec des dates modifiées par soustraction et addition de ses composantes :

```
$time = FrozenTime::create(2021, 1, 31, 22, 11, 30);
$newTime = $time->subDays(5)
    ->addHours(-2)
    ->addMonth(1);
// Affiche '2/26/21, 8:11 PM'
echo $newTime;

// Utilisation des chaînes strtotime.
$newTime = $time->modify('+1 month -5 days -2 hours');
// Affiche '2/26/21, 8:11 PM'
echo $newTime;
```

Vous pouvez obtenir des composantes internes d'une date en accédant à ses propriétés :

```
$time = FrozenTime::create(2021, 1, 31, 22, 11, 30);
echo $time->year; // 2021
echo $time->month; // 1
echo $time->day; // 31
echo $time->timezoneName; // America/New_York
```

Formatage

static `Cake\I18n\FrozenTime::`**`setJsonEncodeFormat`**(*$format*)

Cette méthode définit le format par défaut utilisé lors de la conversion d'un objet en json :

```
Time::setJsonEncodeFormat('yyyy-MM-dd HH:mm:ss');  // Pour tout DateTime modifiable
FrozenTime::setJsonEncodeFormat('yyyy-MM-dd HH:mm:ss');  // Pour tout DateTime non␣
↪modifiable
Date::setJsonEncodeFormat('yyyy-MM-dd HH:mm:ss');  // Pour toute Date modifiable
```

(suite sur la page suivante)

(suite de la page précédente)

```
FrozenDate::setJsonEncodeFormat('yyyy-MM-dd HH:mm:ss');  // Pour toute Date non␣
↪modifiable

$time = FrozenTime::parse('2021-01-31 22:11:30');
echo json_encode($time);   // Affiche '2021-01-31 22:11:30'

// Ajouté dans 4.1.0
FrozenDate::setJsonEncodeFormat(static function($time) {
    return $time->format(DATE_ATOM);
});
```

Note

Cette méthode doit être appelée statiquement.

`Cake\I18n\FrozenTime::i18nFormat`(*$format = null*, *$timezone = null*, *$locale = null*)

Une chose habituelle à faire avec les instances `Time` est d'afficher les dates formatées. CakePHP facilite cela :

```
$time = FrozenTime::parse('2021-01-31 22:11:30');

// Affiche un stamp datetime localisé. Affiche '1/31/21, 10:11 PM'
echo $time;

// Affiche '1/31/21, 10:11 PM' pour la locale en-US
echo $time->i18nFormat();

// Utilise la date complète et le format time. Affiche 'Sunday, January 31, 2021 at␣
↪10:11:30 PM Eastern Standard Time'
echo $time->i18nFormat(\IntlDateFormatter::FULL);

// Utilise la date complète mais un format court de temps. Affiche 'Sunday, January 31,␣
↪2021 at 10:11 PM'
echo $time->i18nFormat([\IntlDateFormatter::FULL, \IntlDateFormatter::SHORT]);

// Affiche '2021-Jan-31 22:11:30'
echo $time->i18nFormat('yyyy-MMM-dd HH:mm:ss');
```

Il est possible de spécifier le format d'affichage désiré. Vous pouvez soit passer une constante IntlDateFormatter [189] en premier argument de cette fonction, soit une chaîne complète de formatage tel que spécifié dans cette ressource : https://unicode-org.github.io/icu/userguide/format_parse/datetime/#datetime-format-syntax.

Vous pouvez aussi formater les dates avec des calendriers non-grégoriens :

```
// Sur ICU version 66.1
$time = FrozenTime::create(2021, 1, 31, 22, 11, 30);

// Affiche 'Friday, Aban 9, 1393 AP at 12:00:00 AM GMT'
$result = $now->i18nFormat(\IntlDateFormatter::FULL, null, 'en-IR@calendar=persian');
```

(suite sur la page suivante)

189. https://www.php.net/manual/en/class.intldateformatter.php

(suite de la page précédente)

```
// Affiche 'Sunday, January 31, 3 Reiwa at 10:11:30 PM Eastern Standard Time'
echo $time->i18nFormat(\IntlDateFormatter::FULL, null, 'en-JP@calendar=japanese');

// Affiche 'Sunday, Twelfth Month 19, 2020(geng-zi) at 10:11:30 PM Eastern Standard Time'
echo $time->i18nFormat(\IntlDateFormatter::FULL, null, 'en-CN@calendar=chinese');

// Affiche 'Sunday, Jumada II 18, 1442 AH at 10:11:30 PM Eastern Standard Time'
echo $time->i18nFormat(\IntlDateFormatter::FULL, null, 'en-SA@calendar=islamic');
```

Les types de calendrier suivants sont supportés :

- japanese
- buddhist
- chinese
- persian
- indian
- islamic
- hebrew
- coptic
- ethiopic

Note

Pour les chaînes constantes, par exemple pour IntlDateFormatter : :FULL, Intl utilise la librairie ICU qui alimente ses données à partir de CLDR (https ://cldr.unicode.org/) dont la version peut varier selon l'installation PHP et donner des résultats différents.

`Cake\I18n\FrozenTime::nice()`

Affiche un format prédéfini "nice" :

```
$time = FrozenTime::parse('2021-01-31 22:11:30', new \DateTimeZone('America/New_York'));

// Affiche 'Jan 31, 2021, 10:11 PM' in en-US
echo $time->nice();
```

Vous pouvez modifier le timezone avec lequel la date est affichée sans modifier l'objet `FrozenTime` ou `Time` lui-même. C'est utile quand vous stockez des dates dans un timezone, mais que vous voulez les afficher dans un timezone propre à un utilisateur :

```
// Affiche 'Monday, February 1, 2021 at 4:11:30 AM Central European Standard Time'
echo $time->i18nFormat(\IntlDateFormatter::FULL, 'Europe/Paris');

// Affiche 'Monday, February 1, 2021 at 12:11:30 PM Japan Standard Time'
echo $time->i18nFormat(\IntlDateFormatter::FULL, 'Asia/Tokyo');

// Le timezone est inchangé. Affiche 'America/New_York'
echo $time->timezoneName;
$now = Time::parse('2014-10-31');
```

Laisser le premier paramètre à `null` va utiliser la chaine de formatage par défaut :

```
// Affiche '2/1/21, 4:11 AM'
echo $time->i18nFormat(null, 'Europe/Paris');
```

Enfin, il est possible d'utiliser une locale différente pour l'affichage d'une date :

```
// Outputs 'lundi 1 février 2021 à 04:11:30 heure normale d'Europe centrale'
echo $time->i18nFormat(\IntlDateFormatter::FULL, 'Europe/Paris', 'fr-FR');

// Outputs '1 févr. 2021 à 04:11'
echo $time->nice('Europe/Paris', 'fr-FR');
```

Définir la Locale par défaut et la Chaîne Format

La locale par défaut avec laquelle les dates sont affichées quand vous utilisez `nice` `i18nFormat` est prise à partir de la directive intl.default_locale [190]. Vous pouvez cependant modifier ceci par défaut à la volée :

```
Time::setDefaultLocale('es-ES'); // Pour tout DateTime modifiable
FrozenTime::setDefaultLocale('es-ES'); // Pour tout DateTime non modifiable
Date::setDefaultLocale('es-ES'); // Pour toute Date modifiable
FrozenDate::setDefaultLocale('es-ES'); // Pour toute Date non modifiable

// Affiche '31 ene. 2021 22:11'
echo $time->nice();
```

À partir de maintenant, les datetimes vont s'afficher avec un format préféré en Espagnol, à moins qu'une locale différente ne soit spécifiée directement dans la méthode de formatage.

De même, il est possible de modifier la chaîne de formatage par défaut à utiliser pour le format `i18nFormat` :

```
Time::setToStringFormat(\IntlDateFormatter::SHORT); // Pour tout DateTime modifiable
FrozenTime::setToStringFormat(\IntlDateFormatter::SHORT); // Pour tout DateTime non␣
↪modifiable
Date::setToStringFormat(\IntlDateFormatter::SHORT); // Pour toute Date modifiable
FrozenDate::setToStringFormat(\IntlDateFormatter::SHORT); // Pour toute Date non␣
↪modifiable

// La même méthode existe pour les Date, FrozenDate, et Time
FrozenTime::setToStringFormat([
    \IntlDateFormatter::FULL,
    \IntlDateFormatter::SHORT
]);
// Affiche 'Sunday, January 31, 2021 at 10:11 PM'
echo $time;

// La même méthode existe pour les Date, FrozenDate, et Time
FrozenTime::setToStringFormat("EEEE, MMMM dd, yyyy 'at' KK:mm:ss a");
// Affiche 'Sunday, January 31, 2021 at 10:11:30 PM'
echo $time;
```

Il est recommandé de toujours utiliser les constantes plutôt que de directement passer une date en format chaîne de caractère.

190. https://www.php.net/manual/en/intl.configuration.php#ini.intl.default-locale

Formater les Temps Relatifs

`Cake\I18n\FrozenTime::timeAgoInWords(array $options = [])`

Souvent, il est utile d'afficher les temps par rapport au présent :

```
$time = new FrozenTime('Jan 31, 2021');
// Le 12 juin 2021, ceci afficherait '4 months, 1 week, 6 days ago'
echo $time->timeAgoInWords(
    ['format' => 'MMM d, YYY', 'end' => '+1 year']
);
// Le 10 novembre 2011, cela afficherait: 2 months, 2 weeks, 6 days ago
```

L'option `end` définit à partir de quel point les temps relatifs doivent être formatés en utilisant l'option `format`. L'option `accuracy` nous permet de contrôler le niveau de détail qui devra être utilisé pour chaque intervalle :

```
// Affiche '4 months ago'
echo $time->timeAgoInWords([
    'accuracy' => ['month' => 'month'],
    'end' => '1 year'
]);
```

En définissant `accuracy` par une chaîne de texte, vous pouvez spécifier le niveau maximum de détail que vous souhaitez afficher :

```
$time = new FrozenTime('+23 hours');
// Affiche 'in about a day'
echo $time->timeAgoInWords([
    'accuracy' => 'day'
]);
```

Conversion

`Cake\I18n\FrozenTime::toQuarter()`

Une fois créées, les instances `Time` peuvent être converties en timestamps ou en trimestre (*quarter* est un quart d'année, c'est-à-dire un trimestre) :

```
$time = new FrozenTime('2021-01-31');
echo $time->toQuarter();  // Affiche '1'
echo $time->toUnixString();  // Affiche '1612069200'
```

Comparer Avec le Present

`Cake\I18n\FrozenTime::isYesterday()`

`Cake\I18n\FrozenTime::isThisWeek()`

`Cake\I18n\FrozenTime::isThisMonth()`

`Cake\I18n\FrozenTime::isThisYear()`

Vous pouvez comparer une instance `Time` avec le temps présent de plusieurs façons :

```
$time = new FrozenTime('+3 days');

debug($time->isYesterday());
debug($time->isThisWeek());
debug($time->isThisMonth());
debug($time->isThisYear());
```

Chacune des méthodes ci-dessus va retourner `true`/`false` selon que l'instance `Time` est ou non dans la même unité de temps que le temps présent.

Comparer Avec Des Intervalles

`Cake\I18n\FrozenTime::isWithinNext($interval)`

Vous pouvez regarder si une instance `Time` tombe dans un intervalle de temps en utilisant `wasWithinLast()` et `isWithinNext()` :

```
$time = new FrozenTime('+3 days');

// Dans moins de 2 jours. Affiche 'false'
debug($time->isWithinNext('2 days'));

// Dans les 2 prochaines semaines. Affiche 'true'
debug($time->isWithinNext('2 weeks'));
```

`Cake\I18n\FrozenTime::wasWithinLast($interval)`

Vous pouvez aussi comparer une instance `FrozenTime` avec un intervalle dans le passé :

```
$time = new FrozenTime('-72 hours');

// Dans les 2 derniers jours. Affiche 'false'
debug($time->wasWithinLast('2 days'));

// Dans les 3 derniers jours. Affiche 'true'
debug($time->wasWithinLast('3 days'));

// Dans les 2 dernières semaines. Affiche 'true'
debug($time->wasWithinLast('2 weeks'));
```

FrozenDate

La classe immuable `FrozenDate` dans CakePHP implémente la même API et les mêmes méthodes que *`Cake\I18n\FrozenTime`*. La différence principale entre `FrozenTime` et `FrozenDate` est que `FrozenDate` ne tient pas compte des composantes d'heure. Voici un exemple :

```
use Cake\I18n\FrozenDate;
$date = new FrozenDate('2021-01-31');

$newDate = $date->modify('+2 hours');
// Affiche '2021-01-31 00:00:00'
echo $newDate->format('Y-m-d H:i:s');
```

(suite sur la page suivante)

(suite de la page précédente)

```
$newDate = $date->addHours(36);
// Affiche '2021-01-31 00:00:00'
echo $newDate->format('Y-m-d H:i:s');

$newDate = $date->addDays(10);
// Affiche '2021-02-10 00:00:00'
echo $newDate->format('Y-m-d H:i:s');
```

Les tentatives de modification de timezone sur une instance `FrozenDate` seront toujours ignorées :

```
use Cake\I18n\FrozenDate;
$date = new FrozenDate('2021-01-31', new \DateTimeZone('America/New_York'));
$newDate = $date->setTimezone(new \DateTimeZone('Europe/Berlin'));

// Affiche 'America/New_York'
echo $newDate->format('e');
```

Dates et Heures Modifiables

class Cake\I18n**Time**

class Cake\I18n**Date**

CakePHP offre des classes de date et d'heure modifiables qui implémentent la même interface que leurs équivalents immuables. Les objets immuables sont utiles quand vous voulez éviter des changements accidentels de données, ou lorsque le bon fonctionnement dépend de l'ordre des traitements. Prenez le code suivant :

```
use Cake\I18n\Time;
$time = new Time('2015-06-15 08:23:45');
$time->modify('+2 hours');

// Cette méthode modifie également l'instance $time
$this->someOtherFunction($time);

// Ici, la sortie est inconnue.
echo $time->format('Y-m-d H:i:s');
```

Si on change l'ordre d'appel aux méthodes, ou si `someOtherFunction` est modifiée, le résultat peut être inattendu. La mutabilité de vos objets crée un couplage temporel. En utilisant des objets immuables, nous pourrions éviter ce type de problème :

```
use Cake\I18n\FrozenTime;
$time = new FrozenTime('2015-06-15 08:23:45');
$time = $time->modify('+2 hours');

// La modification de cette méthode ne change pas $time
$this->someOtherFunction($time);

// La sortie est connue.
echo $time->format('Y-m-d H:i:s');
```

Les date et heures immuables sont utiles dans les entities car elles évitent les modifications accidentelles, et forcent les modifications à être exprimées de façon explicite. Utiliser des objets immuables aide l'ORM à mieux suivre les modifications et assurer que les colones date/datetime sont persistées correctement :

```
// Cette modification sera perdue lrsque l'article sera enregistré.
$article->updated->modify('+1 hour');

// En remplaçant l'objet time, la propriété sera auvegardée
$article->updated = $article->updated->modify('+1 hour');
```

Accepter des Données de Requête Localisées

Quand vous créez des inputs de texte qui manipulent des dates, vous voudrez probablement accepter et parser des chaînes datetime localisées. Consultez *Parser les Données Datetime Localisées*.

Timezones Supportés

CakePHP supporte tous les timezones valides de PHP. Pour la liste des timezones, consultez cette page [191].

191. https://php.net/manual/fr/timezones.php

CHAPITRE 40

Xml

class Cake\Utility**Xml**

La classe Xml vous permet de transformer des tableaux en SimpleXMLElement ou en objets DOMDocument, et de nouveau les transformer en tableaux.

Importer les données vers la classe Xml

static Cake\Utility\Xml::**build**(*$input*, *array $options = []*)

Vous pouvez utiliser `Xml::build()` pour construire des objets XML. Selon votre paramètre `$options`, cette méthode va retourner un objet SimpleXMLElement (default) ou un objet DOMDocument. Vous pouvez utiliser `Xml::build()` pour construire les objets XML à partir d'une variété de sources. Par exemple, vous pouvez charger le XML à partir de chaînes :

```
$text = '<?xml version="1.0" encoding="utf-8"?>
<post>
    <id>1</id>
    <title>Meilleur post</title>
    <body> ... </body>
</post>';
$xml = Xml::build($text);
```

Vous pouvez aussi construire des objets Xml à partir de fichiers locaux :

```
// fichier local
$xml = Xml::build('/home/awesome/unicorns.xml');
```

Vous pouvez aussi construire des objets Xml en utilisant un tableau :

```
$data = [
    'post' => [
        'id' => 1,
        'title' => 'Best post',
        'body' => ' ... '
    ]
];
$xml = Xml::build($data);
```

Si votre entrée est invalide, la classe Xml enverra une Exception :

```
$xmlString = 'What is XML?';
try {
    $xmlObject = Xml::build($xmlString); // Ici une Exception va être lancée
} catch (\Cake\Utility\Exception\XmlException $e) {
    throw new InternalErrorException();
}
```

Note

DOMDocument [192] et SimpleXML [193] implémentent différentes APIs. Assurez-vous d'utiliser les bonnes méthodes sur l'objet que vous requêtez à partir d'un Xml.

Transformer une Chaîne de Caractères XML en Tableau

static Cake\Utility\Xml::**toArray**(*$obj*)

Convertir des chaînes XML en tableaux est aussi facile avec la classe Xml. Par défaut, vous obtiendrez un objet SimpleXml en retour :

```
$xmlString = '<?xml version="1.0"?><root><child>value</child></root>';
$xmlArray = Xml::toArray(Xml::build($xmlString));
```

Si votre XML est invalide, cela enverra une `Cake\Utility\Exception\XmlException`.

Transformer un tableau en une chaîne de caractères XML

```
$xmlArray = ['root' => ['child' => 'value']];
// Vous pouvez aussi utiliser Xml::build().
$xmlObject = Xml::fromArray($xmlArray, ['format' => 'tags']);
$xmlString = $xmlObject->asXML();
```

Votre tableau ne doit avoir qu'un élément de « niveau supérieur » et il ne doit pas être numérique. Si le tableau n'est pas dans le bon format, Xml va lancer une Exception. Des Exemples de tableaux invalides :

```
// Niveau supérieur avec une clé numérique
[
    ['key' => 'value']
```

(suite sur la page suivante)

192. https://php.net/domdocument
193. https://php.net/simplexml

(suite de la page précédente)

```
];

// Plusieurs clés au niveau supérieur
[
    'key1' => 'première valeur',
    'key2' => 'autre valeur'
];
```

Par défaut les valeurs de tableau vont être sorties en tags XML, si vous souhaitez définir les attributs ou les valeurs de texte, vous pouvez préfixer les clés qui sont supposées être des attributs avec `@`. Pour value text, utilisez `@` en clé :

```
$xmlArray = [
    'project' => [
        '@id' => 1,
        'name' => 'Name of project, as tag',
        '@' => 'Value of project'
    ]
];
$xmlObject = Xml::fromArray($xmlArray);
$xmlString = $xmlObject->asXML();
```

Le contenu de `$xmlString` va être :

```
<?xml version="1.0"?>
<project id="1">Value of project<name>Nom du projet, en tag</name></project>
```

Utiliser des Namespaces

Pour utiliser les Namespaces XML, dans votre tableau vous devez créer une clé avec le nom `xmlns:` vers un namespace générique ou avec le préfixe `xmlns:` dans un namespace personnalisé. Regardez les exemples :

```
$xmlArray = [
    'root' => [
        'xmlns:' => 'https://cakephp.org',
        'child' => 'value'
    ]
];
$xml1 = Xml::fromArray($xmlArray);

$xmlArray(
    'root' => [
        'tag' => [
            'xmlns:pref' => 'https://cakephp.org',
            'pref:item' => [
                'item 1',
                'item 2'
            ]
        ]
    ]
);
$xml2 = Xml::fromArray($xmlArray);
```

La valeur de `$xml1` et `$xml2` sera, respectivement :

```
<?xml version="1.0"?>
<root xmlns="https://cakephp.org"><child>value</child>

<?xml version="1.0"?>
<root><tag xmlns:pref="https://cakephp.org"><pref:item>item 1</pref:item><pref:item>item␣
↪2</pref:item></tag></root>
```

Créer un enfant

Après avoir créé votre document XML, vous utilisez seulement les interfaces natives pour votre type de document à ajouter, à retirer, ou manipuler les noeuds enfant :

```
// Utilisation de SimpleXML
$myXmlOriginal = '<?xml version="1.0"?><root><child>value</child></root>';
$xml = Xml::build($myXmlOriginal);
$xml->root->addChild('young', 'new value');

// Utilisation de DOMDocument
$myXmlOriginal = '<?xml version="1.0"?><root><child>value</child></root>';
$xml = Xml::build($myXmlOriginal, ['return' => 'domdocument']);
$child = $xml->createElement('young', 'new value');
$xml->firstChild->appendChild($child);
```

Astuce

Après avoir manipulé votre XML en utilisant SimpleXMLElement ou DomDocument vous pouvez utiliser `Xml::toArray()` sans problèmes.

CHAPITRE 41

Globales & Fonctions

Alors que la plupart de vos activités quotidiennes avec CakePHP sera d'initialiser des classes du noyau, CakePHP dispose d'un certain nombre de fonctions globales de confort qui peuvent arriver à point nommé. La plupart de ses fonctions sont à utiliser avec les classes cakePHP (classes de chargement ou de component), mais beaucoup d'autres rendent le travail avec les tableaux ou les chaînes de caractères un peu plus simple.

Nous allons aussi couvrir une partie des constantes disponibles dans les applications CakePHP. L'utilisation des constantes disponibles vous aidera à faire des mises à jour plus lisses, mais sont aussi des moyens pratiques pour pointer certains fichiers ou répertoires dans vos applications CakePHP.

Fonctions Globales

Voici les fonctions disponibles dans le monde CakePHP. La plupart sont juste des emballages pratiques pour d'autres fonctionnalités CakePHP, comme le débogage et la traduction de contenu.

__(*string $string_id*[, *$formatArgs*])

Cette fonction gère la localisation dans les applications CakePHP. `$string_id` identifie l'ID de la traduction. Vous pouvez fournir des arguments supplémentaires pour remplacer les espaces réservés dans votre chaîne :

```
__('Vous avez {0} messages non-lus', $number);
```

Vous pouvez également fournir un tableau associatif de remplacements :

```
__('Vous avez {unread} messages non-lus', ['unread' => $number]);
```

> **Note**
>
> Regardez la section *Internationalisation & Localisation* pour plus d'information.

__d(*string $domain*, *string $msg*, *mixed $args = null*)

Vous permet de remplacer le domaine courant lors de la recherche d'un message.

Utile pour internationaliser un plugin :

```
echo __d('PluginName', 'Ceci est mon plugin');
```

__dn(*string $domain*, *string $singular*, *string $plural*, *integer $count*, *mixed $args = null*)

Vous permet de redéfinir le domaine courant pour une recherche simple au pluriel d'un message. Retourne la forme pluriel correcte d'un message identifié par `$singular` et `$plural` pour le compteur `$count` depuis le domaine `$domain`.

__dx(*string $domain*, *string $context*, *string $msg*, *mixed $args = null*)

Vous permet de remplacer le domaine courant pour la recherche d'un message. Permet également de spécifier une contexte.

Le contexte est un identifiant unique pour la chaîne de traductions qui le rend unique dans le même domaine.

__dxn(*string $domain*, *string $context*, *string $singular*, *string $plural*, *integer $count*, *mixed $args = null*)

Vous permet de remplacer le domaine courant pour la recherche simple au pluriel d'un message. Cela permet également de spécifier une contexte. Retourne la forme correcte d'un message identifié par `$singular` et `$plural` pour le compteur `$count` depuis le domaine `$domain`. Certaines langues ont plus d'une forme de pluriel dépendant du compteur.

Le contexte est un identifiant unique pour la chaîne de traductions qui le rend unique dans le même domaine.

__n(*string $singular*, *string $plural*, *integer $count*, *mixed $args = null*)

Retourne la forme correcte d'un message identifié par `$singular` et `$plural` pour le compteur `$count`. Certaines langues ont plus d'une forme de pluriel dépendant du compteur.

__x(*string $context*, *string $msg*, *mixed $args = null*)

Le contexte est un identifiant unique pour la chaîne de traductions qui le rend unique dans le même domaine.

__xn(*string $context*, *string $singular*, *string $plural*, *integer $count*, *mixed $args = null*)

Retourne la forme correcte d'un message identifié par `$singular` et `$plural` pour le compteur `$count`. Cela permet également de spécifier une contexte. Certaines langues ont plus d'une forme de pluriel dépendant du compteur.

Le contexte est un identifiant unique pour la chaîne de traductions qui le rend unique dans le même domaine.

collection(*mixed $items*)

Vous permet d'instancier un objet `Cake\Collection\Collection` et wrap l'argument passé. le paramètre `$items` accepte soit un objet `Traversable` soit un tableau.

debug(*mixed $var*, *boolean $showHtml = null*, *$showFrom = true*)

Si la variable `$debug` du cœur est à `true`, $var est affiché. Si `$showHTML` est `true` ou laissé null, la donnée est formatée pour être visualisée facilement dans un navigateur.

Si `$showFrom` n'est pas défini à `false`, debug retournera en sortie la ligne depuis laquelle il a été appelé. Voir aussi *Debugger*

env(*string $key*, *string $default = null*)

Récupère une variable d'environnement depuis les sources disponibles. Utilisé en secours si `$_SERVER` ou `$_ENV` sont désactivés.

Cette fonction émule également `PHP_SELF` et `DOCUMENT_ROOT` sur les serveurs ne les supportant pas. En fait, c'est une bonne idée de toujours utiliser `env()` plutôt que `$_SERVER` ou `getenv()` (notamment si vous prévoyez de distribuer le code), puisque c'est un wrapper d'émulation totale.

h(*string $text*, *boolean $double = true*, *string $charset = null*)

Raccourci pratique pour `htmlspecialchars()`.

pluginSplit(*string $name*, *boolean $dotAppend = false*, *string $plugin = null*)

Divise le nom d'un plugin en notation par point en plugin et classname (nom de classe). Si `$name` de contient pas de point, alors l'index 0 sera `null`.

Communément utilisé comme ceci `list($plugin, $name) = pluginSplit('Users.User');`

namespaceSplit(*string $class*)

Divise le namespace du nom de la classe.

Communément utilisé comme ceci `list($namespace, $className) = namespaceSplit('Cake\Core\App');`

dd(*mixed $var*, *boolean $showHtml = null*)

Cette méthode fonctionne comme `debug()` sauf qu'elle arrêtera l'exécution du script. Si la variable « core » `$debug` vaut `true`, `$var` sera affichée. Si `$showHTML` vaut `true` ou est laissée à `null`, les données seront rendues dans un affichage *user-friendly*. Plus de détails : *Debugger*

pr(*mixed $var*)

Raccourci pratique pour `print_r()`, avec un ajout de balises `<pre>` autour de la sortie.

pj(*mixed $var*)

JSON pretty print convenience function, with the addition of wrapping `<pre>` tags around the output.

Il a pour objectif de debugger la représentation JSON des objets et tableaux.

Définitions des constantes du noyau

La plupart des constantes suivantes font référence aux chemins dans votre application.

constant APP

Chemin absolu du répertoire de l'application avec un slash.

constant APP_DIR

La même chose que `app` ou le nom du répertoire de votre application.

constant CACHE

Chemin vers le répertoire de cache. il peut être partagé entre les hôtes dans une configuration multi-serveurs.

constant CAKE

Chemin vers le répertoire de CAKE.

constant CAKE_CORE_INCLUDE_PATH

Chemin vers la racine du répertoire lib.

constant CONFIG

Chemin vers le répertoire config.

constant CORE_PATH

Chemin vers le répertoire racine avec un slash à la fin.

constant DS

Raccourci pour la constante PHP `DIRECTORY_SEPARATOR`, qui est égale à `/` pour Linux et \ pour Windows.

constant LOGS

Chemin du répertoire des logs.

constant ROOT

Chemin vers le répertoire racine.

constant TESTS

Chemin vers le répertoire de test.

constant TMP

Chemin vers le répertoire des fichiers temporaires.

constant WWW_ROOT

Chemin d'accès complet vers la racine web (webroot).

Définition de Constantes de Temps

constant TIME_START

timestamp Unix en microseconde au format float du démarrage de l'application.

constant SECOND

Égale à 1

constant MINUTE

Égale à 60

constant HOUR

Égale à 3600

constant DAY

Égale à 86400

constant WEEK

Égale à 604800

constant MONTH

Égale à 2592000

constant YEAR

Égale à 31536000

CHAPITRE 42

Chronos

Cette page a été déplacée [194].

194. https://book.cakephp.org/chronos/1.x/fr/

CHAPITRE 43

Debug Kit

Cette page a été déplacée [195].

195. https://book.cakephp.org/debugkit/3.x/fr/

CHAPITRE 44

Migrations

Cette page a été déplacée [196].

196. https://book.cakephp.org/migrations/3/fr/

CHAPITRE 45

ElasticSearch

Cette page a été déplacée [197].

197. https://book.cakephp.org/elasticsearch/3/fr/

CHAPITRE 46

Annexes

Les annexes contiennent des informations sur les nouvelles fonctionnalités introduites dans chaque version et le chemin de migration entre les versions.

4.x Guide de Migration

Mise à Niveau des Fixtures

À partir de la version 4.3.0, le schéma de fixture et les responsabilités de gestion des données sont scindés. Maintenir le schéma dans les classes de fixtures et les migrations rajoutait de la complexité et un coût de maintenance pour les applications. Dans 4.3.0, de nouvelles APIs ont été ajoutées pour permettre de réutiliser plus facilement les migrations existantes ou des outils de gestion de schéma avec des tests.

Pour mettre à niveau vers le nouveau système de fixtures, vous devrez faire quelques mises à jour :

1. Tout d'abord, enlevez le bloc `<listeners>` de votre fichier `phpunit.xml`.
2. Ajoutez ceci à votre fichier `phpunit.xml` :

   ```
   <extensions>
       <extension class="\Cake\TestSuite\Fixture\PHPUnitExtension" />
   </extensions>
   ```

   Cela enlève la gestion de schéma du gestionnaire de fixture de test. À la place, votre application a besoin de créer/mettre à jour le schéma au début de chaque test.
3. Ensuite, mettez à jour `tests/boostrap.php` pour créer le schéma. Il y a plusieurs façons de créer le schéma. Consultez *Créer un Schéma de Base de Données de Test* pour les méthodes fournies par CakePHP.
4. Puis supprimez toutes les propriétés `$fields` et `$import` de vos fixtures. Ces propriétés ne sont plus utilisées dans le nouveau système de fixture.

Vos tests devraient continuer à passer, et vous pouvez essayer avec *Gestionnaires d'Etat des Fixtures*. `TransactionStrategy` qui apporte des gains de performance significatifs. La contrepartie est qu'avec `TransactionStrategy`, vos valeurs auto-incrémentées ne commenceront plus à 1 à chaque test.

Documentation des Anciennes Fixture

La documentation suivante s'applique uniquement aux fixtures basées sur des listeners, ce qui était par défaut avant 4.3.0.

Schéma de Fixture

Nous utilisons `$fields` pour spécifier quels champs feront partie de cette table, et comment ils sont définis. Le format utilisé pour définir les champs est le même que celui de `Cake\Database\Schema\Table`. Les clés disponibles pour la définition de la table sont :

type
Type de données interne à CakePHP. Actuellement supportés :
- `string` : mappe `VARCHAR`
- `char` : mappe `CHAR`
- `uuid` : mappe `UUID`
- `text` : mappe `TEXT`
- `integer` : mappe `INT`
- `biginteger` : mappe `BIGINTEGER`
- `decimal` : mappe `DECIMAL`
- `float` : mappe `FLOAT`
- `datetime` : mappe `DATETIME`
- `datetimefractional` : mappe `DATETIME(6)` or `TIMESTAMP`
- `timestamp` : mappe `TIMESTAMP`
- `timestampfractional` : mappe `TIMESTAMP(6)` or `TIMESTAMP`
- `time` : mappe `TIME`
- `date` : mappe `DATE`
- `binary` : mappe `BLOB`

length
La longueur spécifique que le champ devrait avoir.

precision
Le nombre de décimales utilisées pour les champs float & decimal.

null
Soit `true` (pour autoriser les valeurs NULLs), soit `false` (pour les interdire NULLs).

default
La valeur par défaut pour le champ.

Importer les Informations de la Table

Il peut devenir vraiment délicat de définir le schéma dans les fichiers de fixture quand vous créez des plugins, ou des bibliothèques, ou si vous créez des applications qui doivent pouvoir être portées entre plusieurs logiciels de bases de données.Dans de grandes applications, il peut être difficile de maintenir la redéfinition du schéma dans les fixtures. Pour ces raisons, CakePHP propose d'importer le schéma depuis une connexion existante et d'utiliser les fonctionnalités de réflexion pour obtenir la définition de la table, et créer la définition de table qui sera utilisée dans les tests.

Prenons un exemple. Supposons que nous avons une table nommée articles, et changeons de la façon suivante la fixture d'exemple donnée dans la section précédente (**tests/Fixture/ArticlesFixture.php**) :

```
class ArticlesFixture extends TestFixture
{
    public $import = ['table' => 'articles'];
}
```

Si vous voulez utiliser une autre connexion, utilisez :

```
class ArticlesFixture extends TestFixture
{
    public $import = ['table' => 'articles', 'connection' => 'other'];
}
```

Généralement, vous avez une classe Table parallèlement à votre fixture. Vous pouvez donc vous en servir pour récupérer le nom de la table :

```
class ArticlesFixture extends TestFixture
{
    public $import = ['model' => 'Articles'];
}
```

Cela supporte aussi la syntaxe de plugin.

Vous pouvez bien entendu importer la définition de votre table depuis un modèle ou une table existants, mais avoir vos enregistrements définis directement sur la fixture, comme montré dans la précédente section. Par exemple :

```
class ArticlesFixture extends TestFixture
{
    public $import = ['table' => 'articles'];
    public $records = [
        [
          'title' => 'First Article',
          'body' => 'First Article Body',
          'published' => '1',
          'created' => '2007-03-18 10:39:23',
          'modified' => '2007-03-18 10:41:31'
        ],
        [
          'title' => 'Second Article',
          'body' => 'Second Article Body',
          'published' => '1',
          'created' => '2007-03-18 10:41:23',
          'modified' => '2007-03-18 10:43:31'
        ],
        [
          'title' => 'Third Article',
          'body' => 'Third Article Body',
          'published' => '1',
          'created' => '2007-03-18 10:43:23',
          'modified' => '2007-03-18 10:45:31'
        ]
    ];
}
```

Pour finir, il est aussi possible de ne pas charger ni créer de schéma dans la fixture. C'est utile si vous avez déjà une configuration de base de données de test avec toutes les tables vides déjà créées. Si ni `$fields` ni `$import` ne sont définis, une fixture se contentera d'insérer ses enregistrements et de tronquer les enregistrements à chaque méthode de test.

Mimer la Compatibilité Descendante

Si vous devez/voulez mimer le comportement de 3.x, ou migrer partiellement par étapes, consultez le plugin Shim [198] qui peut vous aider à atténuer certains changements entraînant une rupture de compatibilité descendante.

Mimer la Compatibilité Ascendante

Mimer la compatibilité ascendante peut préparer votre application 3.x pour la version majeure suivante (4.x).

Si vous voulez mimer d'ores et déjà le comportement de 4.x dans votre 3.x, consultez le plugin Shim [199]. Ce plugin est conçu pour atténuer certaines ruptures de compatibilité descendante et aide à porter des fonctionnalités de 4.x dans 3.x. Plus votre application 3.x se rapproche de 4.x, moins vous aurez de changements entre les deux, et la mise à niveau finale en sera d'autant plus sereine.

Informations générales

Le Processus de Développement CakePHP

Les projets CakePHP suivent grosso modo *SemVer <https ://semver.org/>*. Cela signifie que :

- Les versions sont numérotées sous la forme **A.B.C**
- Les versions **A** sont les *versions majeures*. Elles contiennent des ruptures de comportement et nécessiteront une certaine quantité de travail pour une mise à niveau depuis une version **A** inférieure.
- Les versions **A.B** sont des *versions de fonctionnalités*. Chaque version est rétrocompatible mais peut introduire de nouvelles dépréciations. Si un changement de comportement est absolument nécessaire, il sera indiqué dans le guide de migration pour cette version.
- Les versions **A.B.C** sont des versions de *patch*. Elles sont en principe rétrocompatibles avec la précédente version patch. La seule exception à cette règle peut concerner la découverte d'une faille de sécurité, si la seule solution est de modifier l'API existante.

Consultez *Guide de Compatibilité Rétroactive* pour en savoir plus sur ce que nous considérons comme une rétrocompatibilité ou une rupture de comportement.

Versions majeures

Les versions majeures introduisent de nouvelles fonctionnalités et peuvent supprimer des fonctionnalités dépréciées dans une précédente version. Ces versions sont maintenues dans les branches `next` correspondant à leur numéro de version, telles que `5.next`. Une fois publiées, elles sont promues en branche `master` et la branche `5.next` est utilisée pour de futures versions de fonctionnalités.

Versions de fonctionnalités

Les versions de fonctionnalités sont l'endroit où sont introduites de nouvelles fonctionnalités, ou des extensions de fonctionnalités existantes. Chaque série de versions recevant des mises à jour a une branche `next`, par exemple `4.next`. Si vous souhaitez contribuer à une nouvelle fonctionnalité, veuillez cibler ces branches.

Versions de patch

Les versions de patch résolvent des bugs dans le code ou la documentation, et sont censées être toujours compatibles avec les versions de patch antérieures de la même version de fonctionnalités. Ces versions sont créées à partir des branches stables. Les branches stables sont souvent nommées après d'après la série de versions, telle que `3.x`.

198. https://github.com/dereuromark/cakephp-shim
199. https://github.com/dereuromark/cakephp-shim

Cadence de livraison

- Les *Versions Majeures* sont livrées approximativement tous les deux ou trois ans. Cette durée nous oblige à rester mesurés et attentifs sur les ruptures de compatibilité, et donne du temps à la communauté pour se maintenir à niveau sans avoir l'impression d'être à la traîne.
- Les *Versions de Fonctionnalités* sont livrées tous les cinq à huit mois.
- Les *Versions de Patch* sont livrées au départ toutes les deux semaines. Lorsqu'une version de fonctionnalité mûrit, cette cadence ralentit et devient mensuelle.

Politique de Dépréciations

Avant qu'une fonctionnalité ne soit supprimée dans une version majeure, elle doit être dépréciée. Quand un comportement est déprécié dans une version **A.x**, il continue à fonctionner dans toutes les versions restantes **A.x**. Les dépréciations sont généralement indiquées par des avertissements PHP. Vous pouvez activer les avertissements de dépréciation en ajoutant `E_USER_DEPRECATED` à la valeur de `Error.level` dans votre application.

Une fois déprécié, le comportement ne sera pas supprimé avant la prochaine version majeurs. Par exemple, un comportement déprécié dans `4.1` sera supprimé dans `5.0`.

Glossaire

attributs HTML
Un tableau de clé => valeurs qui sont composées dans les attributs HTML. Par exemple :

```
// Par exemple
['class' => 'ma-classe', '_target' => 'blank']

// générerait
class="ma-classe" _target="blank"
```

Si une option peut être minimisée ou a le même nom que sa valeur, alors `true` peut être utilisée :

```
// Par exemple
['checked' => true]

// Générerait
checked="checked"
```

CDN
Content Delivery Network. Une librairie tierce que vous pouvez payer pour vous aider à distribuer votre contenu vers des centres de données dans le monde entier. Cela aide à rapprocher géographiquement vos assets static pour les utilisateurs.

champ(s)
Terme générique utilisé à la fois pour décrire des propriétés d'entity ou des colonnes de base de données ; souvent utilisé avec tout ce qui est lié au FormHelper.

colonnes
Utilisé dans l'ORM lorsqu'il est question de colonne de tables dans une base de données.

CSRF
Les Requêtes de site croisées de Contrefaçon. Empêche les attaques de replay, les soumissions doubles et les requêtes contrefaites provenant d'autres domaines.

DSN
Nom de Source de Données (Data Source Name). Un format de chaîne de connexion qui est formé comme un URI. CakePHP supporte les DSN pour les connections Cache, base de données, Log et Email.

DRY
Ne vous répétez pas vous-même. C'est un principe de développement de logiciel qui a pour objectif de réduire

les répétitions d'information de tout type. Dans CakePHP, DRY est utilisé pour vous permettre de coder des choses et de les réutiliser à travers votre application.

notation avec points

La notation avec points (ou *dot notation*) définit un chemin de tableau, en séparant les niveaux imbriqués avec le caractère `.`. Par exemple :

```
Cache.default.engine
```

Pointerait vers la valeur suivante :

```
[
    'Cache' => [
        'default' => [
            'engine' => 'File'
        ]
    ]
]
```

PaaS

Plate-forme en tant que service (Platform as a Service). Les fournisseurs de plate-forme en tant que service fournissent des hébergements, des bases de données et des ressources de caching basés sur le Cloud. Quelques fournisseurs populaires sont Heroku, EngineYard et PagodaBox

propriétés

Utilisé pour parler de colonnes mappées à des objets `Entity` de l'ORM

routes.php

Un fichier dans APP/Config qui contient la configuration de routing. Ce fichier est inclus avant que chaque requête soit traitée. Il doit connecter toutes les routes dont votre application a besoin afin que les requêtes puissent être routées aux controllers + actions correctes.

syntaxe de plugin

La syntaxe de Plugin fait référence au nom de la classe avec un point en séparation indiquant que les classes sont une partie d'un plugin. Par ex : `DebugKit.Toolbar`, le plugin est DebugKit, et le nom de classe est Toolbar.

tableau de routing

Un tableau des attributs qui sont passés au `Router::url()`. Typiquement, il ressemble à cela :

```
['controller' => 'Posts', 'action' => 'view', 5]
```

PHP Namespace Index

C

Cake\Cache, 571
Cake\Collection, 781
Cake\Console, 615
Cake\Console\Exception, 655
Cake\Controller, 237
Cake\Controller\Component, 270
Cake\Controller\Exception, 655
Cake\Core, 180
Cake\Core\Exception, 656
Cake\Database, 402
Cake\Database\Exception, 655
Cake\Database\Schema, 565
Cake\Datasource, 401
Cake\Datasource\Exception, 656
Cake\Error, 624
Cake\Filesystem, 803
Cake\Form, 691
Cake\Http, 827
Cake\Http\Client, 833
Cake\Http\Cookie, 233
Cake\Http\Exception, 653
Cake\Http\TestSuite, 835
Cake\I18n, 857
Cake\Log, 687
Cake\Mailer, 631
Cake\ORM, 452
Cake\ORM\Behavior, 555
Cake\ORM\Exception, 655
Cake\Routing, 185
Cake\Routing\Exception, 656
Cake\Utility, 867
Cake\Validation, 767
Cake\View, 289
Cake\View\Exception, 655
Cake\View\Helper, 386

Index

Symboles

() (*méthode*), **287**
:action, 186
:controller, 186
:plugin, 186
$this->request, 215
$this->response, 225
__d() (*global function*), **871**
__dn() (*global function*), **872**
__dx() (*global function*), **872**
__dxn() (*global function*), **872**
__n() (*global function*), **872**
__x() (*global function*), **872**
__xn() (*global function*), **872**
__() (*global function*), **871**

A

acceptLanguage() (*Cake\Http\ServerRequest method*), **224**
accepts() (*Cake\Http\ServerRequest method*), **224**
accepts() (*méthode RequestHandlerComponent*), **276**
addArgument() (*Cake\Console\ConsoleOptionParser method*), **601**
addArguments() (*Cake\Console\ConsoleOptionParser method*), **602**
addBehavior() (*Cake\ORM\Table method*), **459**
addCrumb() (*Cake\View\Helper\HtmlHelper method*), **367**
addDetector() (*Cake\Http\ServerRequest method*), **221**
addOption() (*Cake\Console\ConsoleOptionParser method*), **602**
addOptions() (*Cake\Console\ConsoleOptionParser method*), **603**
addPathElement() (*Cake\Filesystem\Folder method*), **804**
admin routing, 195
afterDelete() (*Cake\ORM\Table method*), **457**
afterDeleteCommit() (*Cake\ORM\Table method*), **457**
afterFilter() (*Cake\Controller\Controller method*), **245**
afterLayout() (*méthode Helper*), **394**
afterMarshal() (*Cake\ORM\Table method*), **455**
afterRender() (*méthode Helper*), **394**
afterRenderFile() (*méthode Helper*), **394**
afterRules() (*Cake\ORM\Table method*), **457**
afterSave() (*Cake\ORM\Table method*), **457**
afterSaveCommit() (*Cake\ORM\Table method*), **457**
alert() (*Cake\Log\Log method*), **688**
allControls() (*Cake\View\Helper\FormHelper method*), **350**
allow() (*méthode AuthComponent*), **261**
allowMethod() (*Cake\Http\ServerRequest method*), **223**
App (*classe dans Cake\Core*), **777**
APP (*global constant*), **873**
app.php, 175
APP_DIR (*global constant*), **873**
app_local.example.php, 175
append() (*Cake\Collection\Collection method*), **796**
append() (*Cake\Filesystem\File method*), **808**
application exceptions, 652
apply() (*Cake\Utility\Hash method*), **822**
ask() (*Cake\Console\ConsoleIo method*), **598**
attributs HTML, **887**
AuthComponent (*class*), **246**
autoLink() (*Cake\View\Helper\TextHelper method*), **382**
autoLinkEmails() (*Cake\View\Helper\TextHelper method*), **381**
autoLinkUrls() (*Cake\View\Helper\TextHelper method*), **381**
autoParagraph() (*Cake\View\Helper\TextHelper method*), **382**
avg() (*Cake\Collection\Collection method*), **789**

B

BadRequestException, 653
beforeDelete() (*Cake\ORM\Table method*), 457
beforeFilter() (*Cake\Controller\Controller method*), 244
beforeFind() (*Cake\ORM\Table method*), 456
beforeLayout() (*méthode Helper*), 394
beforeMarshal() (*Cake\ORM\Table method*), 455
beforeRender() (*Cake\Controller\Controller method*), 244
beforeRender() (*méthode Helper*), 394
beforeRenderFile() (*méthode Helper*), 394
beforeRules() (*Cake\ORM\Table method*), 456
beforeSave() (*Cake\ORM\Table method*), 457
blackHole() (*méthode SecurityComponent*), 266
BreadcrumbsHelper (*classe dans Cake\View\Helper*), 311
breakpoint() (*global function*), 623
buffered() (*Cake\Collection\Collection method*), 800
build() (*Cake\Utility\Xml method*), 867
build() (*Cake\View\Helper\UrlHelper method*), 386
buildFromArray() (*Cake\Console\ConsoleOptionParser method*), 604
buildRules() (*Cake\ORM\Table method*), 456
buildValidator() (*Cake\ORM\Table method*), 456
button() (*Cake\View\Helper\FormHelper method*), 344

C

Cache (*classe dans Cake\Cache*), 571
CACHE (*global constant*), 873
cache() (*Cake\View\View method*), 300
CacheEngine (*classe dans Cake\Cache*), 579
CAKE (*global constant*), 873
CAKE_CORE_INCLUDE_PATH (*global constant*), 873
Cake\Cache (*namespace*), 571
Cake\Collection (*namespace*), 781
Cake\Console (*namespace*), 583, 585, 595, 601, 615
Cake\Console\Exception (*namespace*), 655
Cake\Controller (*namespace*), 237
Cake\Controller\Component (*namespace*), 264, 270
Cake\Controller\Exception (*namespace*), 655
Cake\Core (*namespace*), 180, 777
Cake\Core\Exception (*namespace*), 656
Cake\Database (*namespace*), 402
Cake\Database\Exception (*namespace*), 655
Cake\Database\Schema (*namespace*), 565
Cake\Datasource (*namespace*), 401
Cake\Datasource\Exception (*namespace*), 656
Cake\Error (*namespace*), 624
Cake\Filesystem (*namespace*), 803
Cake\Form (*namespace*), 691
Cake\Http (*namespace*), 215, 827
Cake\Http\Client (*namespace*), 833
Cake\Http\Cookie (*namespace*), 233
Cake\Http\Exception (*namespace*), 653
Cake\Http\TestSuite (*namespace*), 835
Cake\I18n (*namespace*), 841, 857
Cake\Log (*namespace*), 687
Cake\Mailer (*namespace*), 631
Cake\ORM (*namespace*), 415, 452, 461, 471, 506, 529
Cake\ORM\Behavior (*namespace*), 543, 545, 547, 555
Cake\ORM\Exception (*namespace*), 655
Cake\Routing (*namespace*), 185
Cake\Routing\Exception (*namespace*), 656
Cake\Utility (*namespace*), 715, 811, 837, 849, 867
Cake\Validation (*namespace*), 767
Cake\View (*namespace*), 289
Cake\View\Exception (*namespace*), 655
Cake\View\Helper (*namespace*), 311, 314, 315, 355, 368, 373, 381, 385, 386
camelize() (*Cake\Utility\Inflector method*), 839
cd() (*Cake\Filesystem\Folder method*), 804
CDN, 887
champ(s), 887
charset() (*Cake\View\Helper\HtmlHelper method*), 355
check() (*Cake\Core\Configure method*), 181
check() (*Cake\Utility\Hash method*), 818
check() (*méthode Session*), 729
checkbox() (*Cake\View\Helper\FormHelper method*), 331
CheckHttpCacheComponent (*class*), 282
chmod() (*Cake\Filesystem\Folder method*), 804
chunk() (*Cake\Collection\Collection method*), 785
chunkWithKeys() (*Cake\Collection\Collection method*), 786
classify() (*Cake\Utility\Inflector method*), 839
classname() (*Cake\Core\App method*), 777
cleanInsert() (*Cake\Utility\Text method*), 851
clear() (*Cake\Cache\Cache method*), 576
clear() (*Cake\Cache\CacheEngine method*), 580
clear() (*Cake\ORM\TableLocator method*), 461
clearGroup() (*Cake\Cache\Cache method*), 578
clearGroup() (*Cake\Cache\CacheEngine method*), 580
Client (*classe dans Cake\Http*), 827
clientIp() (*Cake\Http\ServerRequest method*), 223
close() (*Cake\Filesystem\File method*), 808
Collection (*classe dans Cake\Collection*), 781
Collection (*classe dans Cake\Database\Schema*), 569
collection() (*global function*), 872
colonnes, 887
combine() (*Cake\Collection\Collection method*), 784
combine() (*Cake\Utility\Hash method*), 814
Command (*classe dans Cake\Console*), 585
compile() (*Cake\Collection\Collection method*), 801
CONFIG (*global constant*), 873
config() (*Cake\Cache\Cache method*), 572
config() (*Cake\I18n\Number method*), 846

configuration, 175
Configure (*classe dans Cake\Core*), 180
configured() (*Cake\Log\Log method*), 687
ConflictException, 653
Connection (*classe dans Cake\Database*), 409
ConnectionManager (*classe dans Cake\Datasource*), 401
ConsoleException, 655
ConsoleIo (*classe dans Cake\Console*), 595
ConsoleOptionParser (*classe dans Cake\Console*), 601
consume() (*Cake\Core\Configure method*), 181
consume() (*méthode Session*), 729
consumeOrFail() (*Cake\Core\Configure method*), 181
contains() (*Cake\Collection\Collection method*), 795
contains() (*Cake\Utility\Hash method*), 817
control() (*Cake\View\Helper\FormHelper method*), 320
Controller (*classe dans Cake\Controller*), 237
controls() (*Cake\View\Helper\FormHelper method*), 350
Cookie (*classe dans Cake\Http\Cookie*), 233
CookieCollection (*classe dans Cake\Http\Cookie*), 233
copy() (*Cake\Filesystem\File method*), 808
copy() (*Cake\Filesystem\Folder method*), 804
core() (*Cake\Core\App method*), 778
CORE_PATH (*global constant*), 873
correctSlashFor() (*Cake\Filesystem\Folder method*), 805
countBy() (*Cake\Collection\Collection method*), 790
counter() (*Cake\View\Helper\PaginatorHelper method*), 378
CounterCacheBehavior (*classe dans Cake\ORM\Behavior*), 543
create() (*Cake\Filesystem\File method*), 808
create() (*Cake\Filesystem\Folder method*), 805
create() (*Cake\View\Helper\FormHelper method*), 315
createFile() (*Cake\Console\ConsoleIo method*), 598
critical() (*Cake\Log\Log method*), 688
CSRF, 887
css() (*Cake\View\Helper\HtmlHelper method*), 356
currency() (*Cake\I18n\Number method*), 842
currency() (*Cake\View\Helper\NumberHelper method*), 368
current() (*Cake\View\Helper\PaginatorHelper method*), 378

D

dasherize() (*Cake\Utility\Inflector method*), 839
Date (*classe dans Cake\I18n*), 865
date() (*Cake\View\Helper\FormHelper method*), 339
dateTime() (*Cake\View\Helper\FormHelper method*), 339
DateTimeFractionalType (*classe dans Cake\Database*), 403
DateTimeTimezoneType (*classe dans Cake\Database*), 404
DateTimeType (*classe dans Cake\Database*), 403
DAY (*global constant*), 874
dd() (*global function*), 873
debug() (*Cake\Log\Log method*), 688
debug() (*global function*), 872
Debugger (*classe dans Cake\Error*), 624
decrement() (*Cake\Cache\Cache method*), 577
decrement() (*Cake\Cache\CacheEngine method*), 580
decrypt() (*Cake\Utility\Security method*), 715
defaultCurrency() (*Cake\I18n\Number method*), 842
defaultCurrency() (*Cake\View\Helper\NumberHelper method*), 369
delete() (*Cake\Cache\Cache method*), 576
delete() (*Cake\Cache\CacheEngine method*), 579
delete() (*Cake\Core\Configure method*), 181
delete() (*Cake\Filesystem\File method*), 808
delete() (*Cake\Filesystem\Folder method*), 805
delete() (*Cake\ORM\Table method*), 529
delete() (*méthode Session*), 729
deleteAll() (*Cake\ORM\Table method*), 530
deleteMany() (*Cake\Cache\Cache method*), 576
deleteOrFail() (*Cake\ORM\Table method*), 530
deny() (*méthode AuthComponent*), 261
destroy() (*méthode Session*), 729
diff() (*Cake\Utility\Hash method*), 823
dimensions() (*Cake\Utility\Hash method*), 821
dirsize() (*Cake\Filesystem\Folder method*), 805
dirty() (*Cake\ORM\Entity method*), 466
disable() (*Cake\Cache\Cache method*), 579
dispatchShell() (*Cake\Console\Shell method*), 618
doc (*role*), 142
docType() (*Cake\View\Helper\HtmlHelper method*), 358
domain() (*Cake\Http\ServerRequest method*), 222
drop() (*Cake\Cache\Cache method*), 574
drop() (*Cake\Core\Configure method*), 182
drop() (*Cake\Log\Log method*), 687
drop() (*Cake\Mailer\Mailer method*), 640
DRY, 887
DS (*global constant*), 873
DSN, 887
dump() (*Cake\Core\Configure method*), 183
dump() (*Cake\Error\Debugger method*), 624

E

each() (*Cake\Collection\Collection method*), 782
element() (*Cake\View\View method*), 297
emergency() (*Cake\Log\Log method*), 688
enable() (*Cake\Cache\Cache method*), 579
enabled() (*Cake\Cache\Cache method*), 579

encrypt() (*Cake\Utility\Security method*), 715
end() (*Cake\View\Helper\FormHelper method*), 345
Entity (*classe dans Cake\ORM*), 461
env() (*Cake\Http\ServerRequest method*), 219
env() (*global function*), 872
error() (*Cake\Log\Log method*), 688
error() (*Cake\View\Helper\FormHelper method*), 342
errors() (*Cake\Filesystem\Folder method*), 805
every() (*Cake\Collection\Collection method*), 787
Exception, 656
ExceptionRenderer (*class*), 646
excerpt() (*Cake\Error\Debugger method*), 625
excerpt() (*Cake\Utility\Text method*), 855
excerpt() (*Cake\View\Helper\TextHelper method*), 384
executable() (*Cake\Filesystem\File method*), 808
execute() (*Cake\Database\Connection method*), 409
exists() (*Cake\Filesystem\File method*), 808
expand() (*Cake\Utility\Hash method*), 819
ext() (*Cake\Filesystem\File method*), 808
extensions() (*Cake\Routing\RouterBuilder method*), 199
extract() (*Cake\Collection\Collection method*), 783
extract() (*Cake\Utility\Hash method*), 812

F

fallbacks() (*Cake\Routing\RouterBuilder method*), 212
fetchTable() (*Cake\Controller\Controller method*), 243
File (*classe dans Cake\Filesystem*), 808
file extensions, 199
file() (*Cake\View\Helper\FormHelper method*), 338
filter() (*Cake\Collection\Collection method*), 786
filter() (*Cake\Utility\Hash method*), 818
find() (*Cake\Filesystem\Folder method*), 805
find() (*Cake\ORM\Table method*), 473
findOrCreate() (*Cake\ORM\Table method*), 527
findRecursive() (*Cake\Filesystem\Folder method*), 806
first() (*Cake\Collection\Collection method*), 796
first() (*Cake\View\Helper\PaginatorHelper method*), 377
firstMatch() (*Cake\Collection\Collection method*), 787
FlashComponent (*classe dans Cake\Controller\Component*), 264
FlashHelper (*classe dans Cake\View\Helper*), 314
flatten() (*Cake\Utility\Hash method*), 819
Folder (*Cake\Filesystem\File property*), 808
Folder (*classe dans Cake\Filesystem*), 804
Folder() (*Cake\Filesystem\File method*), 808
ForbiddenException, 653
Form (*classe dans Cake\Form*), 691
format() (*Cake\I18n\Number method*), 844
format() (*Cake\Utility\Hash method*), 816
format() (*Cake\View\Helper\NumberHelper method*), 370
formatDelta() (*Cake\I18n\Number method*), 845
formatDelta() (*Cake\View\Helper\NumberHelper method*), 372
FormHelper (*classe dans Cake\View\Helper*), 315
FormProtection (*class*), 279
FrozenTime (*classe dans Cake\I18n*), 857

G

gc() (*Cake\Cache\Cache method*), 577
gc() (*Cake\Cache\CacheEngine method*), 580
generateUrl() (*Cake\View\Helper\PaginatorHelper method*), 379
get() (*Cake\Datasource\ConnectionManager method*), 401
get() (*Cake\ORM\Entity method*), 463
get() (*Cake\ORM\Table method*), 472
get() (*Cake\ORM\TableLocator method*), 460
get() (*Cake\Utility\Hash method*), 812
getCrumbList() (*Cake\View\Helper\HtmlHelper method*), 367
getCrumbs() (*Cake\View\Helper\HtmlHelper method*), 367
getData() (*Cake\Http\ServerRequest method*), 216
getMethod() (*Cake\Http\ServerRequest method*), 223
getQuery() (*Cake\Http\ServerRequest method*), 216
getType() (*Cake\Error\Debugger method*), 625
getUploadedFile() (*Cake\Http\ServerRequest method*), 218
getUploadedFiles() (*Cake\Http\ServerRequest method*), 218
GoneException, 653
greedy star, 186
group() (*Cake\Filesystem\File method*), 808
groupBy() (*Cake\Collection\Collection method*), 789
groupConfigs() (*Cake\Cache\Cache method*), 578

H

h() (*global function*), 872
handle (*Cake\Filesystem\File property*), 808
Hash (*classe dans Cake\Utility*), 811
hash() (*Cake\Utility\Security method*), 716
hasNext() (*Cake\View\Helper\PaginatorHelper method*), 378
hasPage() (*Cake\View\Helper\PaginatorHelper method*), 378
hasPrev() (*Cake\View\Helper\PaginatorHelper method*), 378
Helper (*class*), 393
hidden() (*Cake\View\Helper\FormHelper method*), 327
highlight() (*Cake\Utility\Text method*), 852

highlight() (*Cake\View\Helper\TextHelper method*), 382
host() (*Cake\Http\ServerRequest method*), 222
HOUR (*global constant*), 874
HtmlHelper (*classe dans Cake\View\Helper*), 355
HttpClientTrait (*trait in Cake\Http\TestSuite*), 835

I

i18nFormat() (*Cake\I18n\FrozenTime method*), 860
identify() (*méthode AuthComponent*), 249
image() (*Cake\View\Helper\HtmlHelper method*), 359
inCakePath() (*Cake\Filesystem\Folder method*), 806
increment() (*Cake\Cache\Cache method*), 577
increment() (*Cake\Cache\CacheEngine method*), 580
indexBy() (*Cake\Collection\Collection method*), 790
Inflector (*classe dans Cake\Utility*), 837
info (*Cake\Filesystem\File property*), 808
info() (*Cake\Filesystem\File method*), 809
info() (*Cake\Log\Log method*), 688
initialize() (*Cake\Console\Shell method*), 621
initialize() (*Cake\ORM\Table method*), 455
inPath() (*Cake\Filesystem\Folder method*), 806
input() (*Cake\Http\ServerRequest method*), 219
insert() (*Cake\Collection\Collection method*), 797
insert() (*Cake\Utility\Hash method*), 812
insert() (*Cake\Utility\Text method*), 851
InternalErrorException, 654
InvalidCsrfTokenException, 653
is() (*Cake\Http\ServerRequest method*), 220
isAbsolute() (*Cake\Filesystem\Folder method*), 806
isAtom() (*méthode RequestHandlerComponent*), 276
isEmpty() (*Cake\Collection\Collection method*), 794
isFieldError() (*Cake\View\Helper\FormHelper method*), 343
isMobile() (*méthode RequestHandlerComponent*), 276
isNotModified() (*Cake\Http\Response method*), 232
isRss() (*méthode RequestHandlerComponent*), 276
isSlashTerm() (*Cake\Filesystem\Folder method*), 806
isThisMonth() (*Cake\I18n\FrozenTime method*), 863
isThisWeek() (*Cake\I18n\FrozenTime method*), 863
isThisYear() (*Cake\I18n\FrozenTime method*), 863
isWap() (*méthode RequestHandlerComponent*), 277
isWindowsPath() (*Cake\Filesystem\Folder method*), 807
isWithinNext() (*Cake\I18n\FrozenTime method*), 864
isXml() (*méthode RequestHandlerComponent*), 276
isYesterday() (*Cake\I18n\FrozenTime method*), 863

J

JsonView (*class*), 309

L

label() (*Cake\View\Helper\FormHelper method*), 341
last() (*Cake\Collection\Collection method*), 796
last() (*Cake\View\Helper\PaginatorHelper method*), 377
lastAccess() (*Cake\Filesystem\File method*), 809
lastChange() (*Cake\Filesystem\File method*), 809
levels() (*Cake\Log\Log method*), 688
limitControl() (*Cake\View\Helper\PaginatorHelper method*), 379
link() (*Cake\View\Helper\HtmlHelper method*), 359
listNested() (*Cake\Collection\Collection method*), 793
load() (*Cake\Core\Configure method*), 182
loadComponent() (*Cake\Controller\Controller method*), 244
lock (*Cake\Filesystem\File property*), 808
Log (*classe dans Cake\Log*), 687
log() (*Cake\Error\Debugger method*), 624
log() (*Cake\Log\LogTrait method*), 688
logout() (*méthode AuthComponent*), 258
LOGS (*global constant*), 873
LogTrait (*trait in Cake\Log*), 688

M

Mailer (*classe dans Cake\Mailer*), 631
map() (*Cake\Collection\Collection method*), 782
map() (*Cake\Database\TypeFactory method*), 404
map() (*Cake\Utility\Hash method*), 822
match() (*Cake\Collection\Collection method*), 787
max() (*Cake\Collection\Collection method*), 788
maxDimensions() (*Cake\Utility\Hash method*), 821
md5() (*Cake\Filesystem\File method*), 809
media() (*Cake\View\Helper\HtmlHelper method*), 361
median() (*Cake\Collection\Collection method*), 789
merge() (*Cake\Console\ConsoleOptionParser method*), 604
merge() (*Cake\Utility\Hash method*), 820
mergeDiff() (*Cake\Utility\Hash method*), 823
messages() (*Cake\Filesystem\Folder method*), 807
meta() (*Cake\View\Helper\HtmlHelper method*), 357
MethodNotAllowedException, 653
middleware() (*Cake\Controller\Controller method*), 245
mime() (*Cake\Filesystem\File method*), 809
min() (*Cake\Collection\Collection method*), 788
MINUTE (*global constant*), 874
MissingActionException, 655
MissingBehaviorException, 656
MissingCellException, 655
MissingCellViewException, 655
MissingComponentException, 655
MissingConnectionException, 655
MissingControllerException, 656
MissingDispatcherFilterException, 656
MissingDriverException, 655
MissingElementException, 655

MissingEntityException, 655
MissingExtensionException, 655
MissingHelperException, 655
MissingLayoutException, 655
MissingRouteException, 656
MissingShellException, 655
MissingShellMethodException, 655
MissingTableException, 655
MissingTaskException, 655
MissingTemplateException, 655
MissingViewException, 655
mode (*Cake\Filesystem\Folder property*), 804
MONTH (*global constant*), 874
month() (*Cake\View\Helper\FormHelper method*), 340
move() (*Cake\Filesystem\Folder method*), 807

N

name (*Cake\Filesystem\File property*), 808
name() (*Cake\Filesystem\File method*), 809
namespaceSplit() (*global function*), 873
nest() (*Cake\Collection\Collection method*), 792
nest() (*Cake\Utility\Hash method*), 825
nested commands, 584
nestedList() (*Cake\View\Helper\HtmlHelper method*), 363
newClientResponse() (*Cake\Http\TestSuite\HttpClientTrait method*), 835
newQuery() (*Cake\Database\Connection method*), 410
next() (*Cake\View\Helper\PaginatorHelper method*), 377
nice() (*Cake\I18n\FrozenTime method*), 861
normalize() (*Cake\Utility\Hash method*), 824
normalizeFullPath() (*Cake\Filesystem\Folder method*), 807
NotAcceptableException, 653
notation avec points, 888
NotFoundException, 653
notice() (*Cake\Log\Log method*), 688
NotImplementedException, 654
Number (*classe dans Cake\I18n*), 841
NumberHelper (*classe dans Cake\View\Helper*), 368
numbers() (*Cake\View\Helper\PaginatorHelper method*), 376
numeric() (*Cake\Utility\Hash method*), 821

O

offset() (*Cake\Filesystem\File method*), 809
open() (*Cake\Filesystem\File method*), 809
options() (*Cake\View\Helper\PaginatorHelper method*), 379
ordinal() (*Cake\I18n\Number method*), 845
ordinal() (*Cake\View\Helper\NumberHelper method*), 371
owner() (*Cake\Filesystem\File method*), 809

P

PaaS, 888
paginate() (*Cake\Controller\Controller method*), 243
PaginatorComponent (*classe dans Cake\Controller\Component*), 270
PaginatorHelper (*classe dans Cake\View\Helper*), 373
parseFileSize() (*Cake\Utility\Text method*), 851
passed arguments, 205
password() (*Cake\View\Helper\FormHelper method*), 327
path (*Cake\Filesystem\File property*), 808
path (*Cake\Filesystem\Folder property*), 804
path() (*Cake\Core\App method*), 778
perms() (*Cake\Filesystem\File method*), 809
PersistenceFailedException, 656
php:attr (*directive*), 144
php:attr (*role*), 145
php:class (*directive*), 143
php:class (*role*), 145
php:const (*directive*), 143
php:const (*role*), 145
php:exc (*role*), 145
php:exception (*directive*), 143
php:func (*role*), 145
php:function (*directive*), 143
php:global (*directive*), 143
php:global (*role*), 145
php:meth (*role*), 145
php:method (*directive*), 144
php:staticmethod (*directive*), 144
pj() (*global function*), 873
plugin routing, 196
plugin() (*Cake\Routing\RouterBuilder method*), 197
pluginSplit() (*global function*), 873
pluralize() (*Cake\Utility\Inflector method*), 838
postButton() (*Cake\View\Helper\FormHelper method*), 346
postLink() (*Cake\View\Helper\FormHelper method*), 347
pr() (*global function*), 873
precision() (*Cake\I18n\Number method*), 843
precision() (*Cake\View\Helper\NumberHelper method*), 369
prefers() (*méthode RequestHandlerComponent*), 278
prefix routing, 195
prefix() (*Cake\Routing\RouterBuilder method*), 195
prepare() (*Cake\Filesystem\File method*), 809
prev() (*Cake\View\Helper\PaginatorHelper method*), 376
PrivateActionException, 655
propriétés, 888
pwd() (*Cake\Filesystem\File method*), 809

pwd() (*Cake\Filesystem\Folder method*), 807

Q

Query (*classe dans Cake\ORM*), 415
query() (*Cake\Database\Connection method*), 409

R

radio() (*Cake\View\Helper\FormHelper method*), 332
randomBytes() (*Cake\Utility\Security method*), 717
read() (*Cake\Cache\Cache method*), 575
read() (*Cake\Cache\CacheEngine method*), 579
read() (*Cake\Core\Configure method*), 180
read() (*Cake\Filesystem\File method*), 809
read() (*Cake\Filesystem\Folder method*), 807
read() (*méthode Session*), 729
readable() (*Cake\Filesystem\File method*), 809
readMany() (*Cake\Cache\Cache method*), 576
readOrFail() (*Cake\Core\Configure method*), 181
realpath() (*Cake\Filesystem\Folder method*), 807
RecordNotFoundException, 656
redirect() (*Cake\Controller\Controller method*), 242
redirectUrl() (*méthode AuthComponent*), 250
reduce() (*Cake\Collection\Collection method*), 788
reduce() (*Cake\Utility\Hash method*), 822
ref (*role*), 143
referer() (*Cake\Http\ServerRequest method*), 223
reject() (*Cake\Collection\Collection method*), 786
remember() (*Cake\Cache\Cache method*), 575
remove() (*Cake\Utility\Hash method*), 813
render() (*Cake\Controller\Controller method*), 240
renderAs() (*méthode RequestHandlerComponent*), 278
renew() (*méthode Session*), 730
replaceText() (*Cake\Filesystem\File method*), 809
RequestHandlerComponent (*class*), 275
requireAuth() (*méthode SecurityComponent*), 267
requireSecure() (*méthode SecurityComponent*), 267
respondAs() (*méthode RequestHandlerComponent*), 278
Response (*classe dans Cake\Http*), 225
Response (*classe dans Cake\Http\Client*), 833
responseHeader() (*Cake\Core\Exception\Exception method*), 656
responseType() (*méthode RequestHandlerComponent*), 278
restore() (*Cake\Core\Configure method*), 183
reverse() (*Cake\Routing\RouterBuilder method*), 206
RFC
    RFC 2606, 158
    RFC 2616 Section 10.4, 654
    RFC 2616 Section 10.5, 654
    RFC 4122, 851
ROOT (*global constant*), 873
RouterBuilder (*classe dans Cake\Routing*), 185
routes.php, 185, 888
rules() (*Cake\Utility\Inflector method*), 840

S

safe() (*Cake\Filesystem\File method*), 809
sample() (*Cake\Collection\Collection method*), 795
save() (*Cake\ORM\Table method*), 519
saveMany() (*Cake\ORM\Table method*), 528
saveOrFail() (*Cake\ORM\Table method*), 526
script() (*Cake\View\Helper\HtmlHelper method*), 362
scriptBlock() (*Cake\View\Helper\HtmlHelper method*), 363
scriptEnd() (*Cake\View\Helper\HtmlHelper method*), 363
scriptStart() (*Cake\View\Helper\HtmlHelper method*), 363
SECOND (*global constant*), 874
secure() (*Cake\View\Helper\FormHelper method*), 354
Security (*classe dans Cake\Utility*), 715
SecurityComponent (*class*), 265
select() (*Cake\View\Helper\FormHelper method*), 333
ServerRequest (*classe dans Cake\Http*), 215
ServiceUnavailableException, 654
Session (*class*), 728
set() (*Cake\Controller\Controller method*), 239
set() (*Cake\ORM\Entity method*), 463
set() (*Cake\View\View method*), 291
setAction() (*Cake\Controller\Controller method*), 243
setAttachments() (*Cake\Mailer\Mailer method*), 635
setConfig() (*Cake\Core\Configure method*), 182
setConfig() (*Cake\Log\Log method*), 687
setDescription() (*Cake\Console\ConsoleOptionParser method*), 606
setEmailPattern() (*Cake\Mailer\Mailer method*), 636
setEpilog() (*Cake\Console\ConsoleOptionParser method*), 606
setJsonEncodeFormat() (*Cake\I18n\FrozenTime method*), 859
setRouteClass() (*Cake\Routing\RouterBuilder method*), 212
setTemplates() (*Cake\View\Helper\HtmlHelper method*), 367
setTemplates() (*Cake\View\Helper\PaginatorHelper method*), 374
setTimezone() (*Cake\Database\DateTimeType method*), 403
setUser() (*méthode AuthComponent*), 258
Shell (*classe dans Cake\Console*), 615
shuffle() (*Cake\Collection\Collection method*), 795
singularize() (*Cake\Utility\Inflector method*), 838
size() (*Cake\Filesystem\File method*), 809
skip() (*Cake\Collection\Collection method*), 796
slashTerm() (*Cake\Filesystem\Folder method*), 807
slug() (*Cake\Utility\Text method*), 850
some() (*Cake\Collection\Collection method*), 787

sort (*Cake\Filesystem\Folder property*), 804
sort() (*Cake\Utility\Hash method*), 822
sort() (*Cake\View\Helper\PaginatorHelper method*), 374
sortBy() (*Cake\Collection\Collection method*), 791
sortDir() (*Cake\View\Helper\PaginatorHelper method*), 375
sortKey() (*Cake\View\Helper\PaginatorHelper method*), 376
stackTrace() (*global function*), 623
startup() (*Cake\Console\Shell method*), 621
stopWhen() (*Cake\Collection\Collection method*), 784
store() (*Cake\Core\Configure method*), 183
stripLinks() (*Cake\Utility\Text method*), 853
stripLinks() (*Cake\View\Helper\TextHelper method*), 383
style() (*Cake\View\Helper\HtmlHelper method*), 356
subcommands, 584
subdomains() (*Cake\Http\ServerRequest method*), 222
submit() (*Cake\View\Helper\FormHelper method*), 343
sumOf() (*Cake\Collection\Collection method*), 788
syntaxe de plugin, 888

T

Table (*classe dans Cake\ORM*), 471
tableau de routing, 888
tableCells() (*Cake\View\Helper\HtmlHelper method*), 365
tableHeaders() (*Cake\View\Helper\HtmlHelper method*), 364
tableize() (*Cake\Utility\Inflector method*), 839
TableLocator (*classe dans Cake\ORM*), 460
TableSchema (*classe dans Cake\Database\Schema*), 565
tail() (*Cake\Utility\Text method*), 854
tail() (*Cake\View\Helper\TextHelper method*), 384
take() (*Cake\Collection\Collection method*), 796
TESTS (*global constant*), 874
Text (*classe dans Cake\Utility*), 849
text() (*Cake\View\Helper\FormHelper method*), 326
textarea() (*Cake\View\Helper\FormHelper method*), 327
TextHelper (*classe dans Cake\View\Helper*), 381
through() (*Cake\Collection\Collection method*), 799
Time (*classe dans Cake\I18n*), 865
time() (*Cake\View\Helper\FormHelper method*), 340
TIME_START (*global constant*), 874
timeAgoInWords() (*Cake\I18n\FrozenTime method*), 863
TimeHelper (*classe dans Cake\View\Helper*), 385
TimestampBehavior (*classe dans Cake\ORM\Behavior*), 545
TMP (*global constant*), 874
toArray() (*Cake\Utility\Xml method*), 868
tokenize() (*Cake\Utility\Text method*), 851
toList() (*Cake\Utility\Text method*), 855
toList() (*Cake\View\Helper\TextHelper method*), 385
toPercentage() (*Cake\I18n\Number method*), 843
toPercentage() (*Cake\View\Helper\NumberHelper method*), 370
toQuarter() (*Cake\I18n\FrozenTime method*), 863
toReadableSize() (*Cake\I18n\Number method*), 843
toReadableSize() (*Cake\View\Helper\NumberHelper method*), 370
total() (*Cake\View\Helper\PaginatorHelper method*), 378
trace() (*Cake\Error\Debugger method*), 625
trailing star, 186
transactional() (*Cake\Database\Connection method*), 411
TranslateBehavior (*classe dans Cake\ORM\Behavior*), 547
transliterate() (*Cake\Utility\Text method*), 850
transpose() (*Cake\Collection\Collection method*), 795
tree() (*Cake\Filesystem\Folder method*), 807
TreeBehavior (*classe dans Cake\ORM\Behavior*), 555
truncate() (*Cake\Utility\Text method*), 853
truncate() (*Cake\View\Helper\TextHelper method*), 383
TypeFactory (*classe dans Cake\Database*), 402, 404

U

UnauthorizedException, 653
underscore() (*Cake\Utility\Inflector method*), 839
unfold() (*Cake\Collection\Collection method*), 785
unlockField() (*Cake\View\Helper\FormHelper method*), 354
updateAll() (*Cake\ORM\Table method*), 528
url() (*Cake\Routing\RouterBuilder method*), 206
UrlHelper (*classe dans Cake\View\Helper*), 386
user() (*méthode AuthComponent*), 258
uuid() (*Cake\Utility\Text method*), 851

V

Validator (*classe dans Cake\Validation*), 767
variable() (*Cake\Utility\Inflector method*), 839
View (*classe dans Cake\View*), 289

W

warning() (*Cake\Log\Log method*), 688
wasWithinLast() (*Cake\I18n\FrozenTime method*), 864
WEEK (*global constant*), 874
withBody() (*Cake\Http\Response method*), 228
withCache() (*Cake\Http\Response method*), 229
withCharset() (*Cake\Http\Response method*), 228
withDisabledCache() (*Cake\Http\Response method*), 229
withEtag() (*Cake\Http\Response method*), 230
withExpires() (*Cake\Http\Response method*), 230

withFile() (*Cake\Http\Response method*), 226
withHeader() (*Cake\Http\Response method*), 227
withModified() (*Cake\Http\Response method*), 231
withSharable() (*Cake\Http\Response method*), 229
withStringBody() (*Cake\Http\Response method*), 228
withType() (*Cake\Http\Response method*), 226
withUploadedFiles() (*Cake\Http\ServerRequest method*), 218
withVary() (*Cake\Http\Response method*), 231
wrap() (*Cake\Utility\Text method*), 852
wrapBlock() (*Cake\Utility\Text method*), 852
writable() (*Cake\Filesystem\File method*), 809
write() (*Cake\Cache\Cache method*), 574
write() (*Cake\Cache\CacheEngine method*), 579
write() (*Cake\Core\Configure method*), 180
write() (*Cake\Filesystem\File method*), 809
write() (*Cake\Log\Log method*), 688
write() (*méthode Session*), 729
writeMany() (*Cake\Cache\Cache method*), 574
WWW_ROOT (*global constant*), 874

X

Xml (*classe dans Cake\Utility*), 867
XmlView (*class*), 309

Y

YEAR (*global constant*), 874
year() (*Cake\View\Helper\FormHelper method*), 340

Z

zip() (*Cake\Collection\Collection method*), 790