CakePHP Cookbook Documentation

Version 3.9

Cake Software Foundation

avr. 09, 2021

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) . . . 3
Cycle de Requête CakePHP . . . 3
Que le Début . . . 5
Lectures Complémentaires . . . 5

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 . . . 18

3 3.0 Guide de Migration **29**
Prérequis . . . 29
Outil d'Upgrade . . . 29
Organisation des Répertoires de l'Application . . . 30
CakePHP doit être installé avec Composer . . . 30
Namespaces (Espaces de Noms) . . . 30
Constantes retirées . . . 30
Configuration . . . 30
Nouvel ORM . . . 31
Notions de base . . . 31
Debugging . . . 31
Paramétrage/Configuration des Objets . . . 31
Cache . . . 31
Core . . . 32
Console . . . 33
Shell / Task . . . 33
Event . . . 34
Log . . . 34
Routing . . . 35
Network . . . 37

Sessions . . . 37
Network\Http . . . 38
Network\Email . . . 38
Controller . . . 38
Controller\Components . . . 39
Model . . . 41
TestSuite . . . 42
View . . . 43
View\Helper . . . 44
I18n . . . 47
L10n . . . 48
Testing . . . 49
Utility . . . 49

4 Tutoriels et exemples **53**
Tutoriel d'un système de gestion de contenu . . . 53
Tutoriel CMS - Création de la base de données . . . 55
Tutoriel CMS - Création du Controller Articles . . . 58
Tutoriel CMS - Tags et Users . . . 67
Tutoriel CMS - Authentification . . . 75
Tutoriel de Bookmarker . . . 81
Tutoriel de Bookmarker Part 2 . . . 88
Tutoriel d'un Blog . . . 96
Tutoriel d'un Blog - Partie 2 . . . 99
Tutoriel d'un Blog - Partie 3 . . . 110
Tutoriel d'un Blog - Authentification et Autorisations . . . 117

5 Contribuer **125**
Documentation . . . 125
Tickets . . . 133
Code . . . 134
Normes de codes . . . 136
Guide de Compatibilité Rétroactive . . . 147

6 Installation **151**
Exigences . . . 151
Installer CakePHP . . . 152
Permissions . . . 154
Serveur de Développement . . . 154
Production . . . 155
A vous de jouer ! . . . 156
Réécriture d'URL . . . 156

7 Configuration **161**
Configurer votre Application . . . 161
Chemins de Classe Supplémentaires . . . 163
Configuration de Inflection . . . 164
Variables d'Environnement . . . 164
Classe Configure . . . 165
Lire et Ecrire les Fichiers de Configuration . . . 167
Bootstrapping CakePHP . . . 169
Désactiver les tables génériques . . . 170

8 Routing **171**
Tour Rapide . . . 171

Connecter les Routes . . . 173
Créer des Routes RESTful . . . 186
Arguments Passés . . . 189
Générer des URLs . . . 190
Routing de Redirection . . . 191
Classes Route Personnalisées . . . 192
Créer des Paramètres d'URL Persistants . . . 193
Gérer les Paramètres Nommés dans les URLs . . . 194

9 Les Objets Request & Response **199**
ServerRequest . . . 199
Response . . . 207
Définir des Cookies . . . 214
Définir les En-têtes de Requête d'Origine Croisée (Cross Origin Request Headers = CORS) . . . 215
Erreurs Communes avec les Responses Immutables . . . 215
CookieCollections . . . 215

10 Controllers (Contrôleurs) **219**
Le Controller App . . . 220
Déroulement d'une Requête . . . 220
Les Actions du Controller . . . 221
Interactions avec les Vues . . . 221
Rediriger vers d'Autres Pages . . . 224
Chargement des Models Supplémentaires . . . 225
Paginer un Model . . . 226
Configurer les Components à Charger . . . 226
Configurer les Helpers à Charger . . . 227
Cycle de Vie des Callbacks de la Requête . . . 227
Plus sur les Controllers . . . 228

11 Views (Vues) **273**
La View App . . . 273
Templates de Vues . . . 274
Utiliser les Blocks de Vues . . . 277
Layouts . . . 279
Elements . . . 282
Events de View . . . 284
Créer vos propres Classes de View . . . 285
En savoir plus sur les vues . . . 285

12 Accès Base de Données & ORM **385**
Exemple Rapide . . . 386
Pour en savoir plus sur les Models . . . 387

13 La mise en cache **549**
Configuration de la classe Cache . . . 550
Ecrire dans un Cache . . . 552
Lire depuis un Cache . . . 553
Suppression d'un Cache . . . 554
Effacer les Données du Cache . . . 554
Utiliser le Cache pour Stocker les Compteurs . . . 555
Utiliser le Cache pour Stocker les Résultats de Requêtes Courantes . . . 555
Utilisation des Groupes . . . 556
Activer ou Désactiver Globalement le Cache . . . 557
Création d'un moteur de stockage pour le Cache . . . 557

14 Console Bake **559**

15 Outils de Console, Shells, & Tasks **561**
La console de CakePHP . . . 561
Créer un Shell . . . 563
Les Tâches Shell . . . 564
Shell Helpers . . . 566
Appeler d'autres Shells à partir de votre Shell . . . 566
Récupérer les Entrées de l'Utilisateur . . . 567
Créer des Fichiers . . . 567
Sortie de la Console . . . 568
Niveaux de sortie de la Console . . . 569
Style de sortie . . . 569
Arrêter l'Exécution d'un Shell . . . 570
Méthodes Hook . . . 572
Configurer les options et générer de l'aide . . . 572
Renommer des Commandes . . . 581
Routing dans shells / CLI . . . 582
Plus de sujets . . . 582

16 Debugger **595**
Debug Basique . . . 595
Utiliser la Classe Debugger . . . 596
Affichage des Valeurs . . . 596
Logging With Stack Traces . . . 597
Generating Stack Traces . . . 597
Getting an Excerpt From a File . . . 597
Utiliser les Logs pour Debugger . . . 598
Kit de Debug . . . 598

17 Déploiement **599**
Déplacer les Fichiers . . . 599
Modifier le fichier config/app.php . . . 599
Vérifier Votre Sécurité . . . 600
Définir le Document Root . . . 600
Améliorer les Performances de votre Application . . . 600
Déployer une Mise à Jour . . . 601

18 Email **603**
Utilisation basique . . . 603
Configuration . . . 604
Envoyer les pièces jointes . . . 608
Utiliser les Transports . . . 609
Envoyer des Messages Rapidement . . . 610
Envoyer des Emails depuis CLI . . . 611
Créer des emails réutilisables . . . 611

19 Gestion des Erreurs & Exceptions **613**
Configuration des Erreurs et des Exceptions . . . 613
Créer vos Propres Gestionnaires d'Erreurs . . . 614
Changer le Comportement des Erreurs Fatales . . . 614
Classes des Exceptions . . . 615
Exceptions Intégrées de CakePHP . . . 615
Utiliser les Exceptions HTTP dans vos Controllers . . . 618
Exception Renderer . . . 618

Créer vos Propres Exceptions dans votre Application . . . 619
Etendre et Implémenter vos Propres Gestionnaires d'Exceptions . . . 620

20 Événements système **623**
Exemple d'Utilisation d'Événement . . . 624
Accéder aux Gestionnaires d'Événements . . . 624
Events du Cœur . . . 626
Enregistrer les Listeners . . . 626
Dispatcher les Events . . . 629
Lecture Supplémentaire . . . 632

21 Internationalisation & Localisation **633**
Internationaliser Votre Application . . . 633
Utiliser les Fonctions de Traduction . . . 635
Créer Vos Propres Traducteurs . . . 639
Localiser les Dates et les Nombres . . . 643
Sélection Automatique de Locale Basée sur les Données de Requêtes . . . 644

22 Journalisation (logging) **645**
Configuration des flux d'un log (journal) . . . 645
Journalisation des Erreurs et des Exception . . . 647
Interagir avec les Flux de Log . . . 648
Utilisation de l'Adaptateur FileLog . . . 648
Logging vers Syslog . . . 648
Ecrire dans les logs . . . 649
Scopes de Journalisation . . . 650
l'API de Log . . . 651
Logging Trait . . . 652
Utiliser Monolog . . . 652

23 Formulaires Sans Models **653**
Créer un Formulaire . . . 653
Traiter les Données de Requêtes . . . 654
Définir des Valeurs pour le Formulaire . . . 655
Récupérer les Erreurs d'un Formulaire . . . 655
Invalider un Champ de Formulaire depuis un Controller . . . 656
Créer le HTML avec FormHelper . . . 656

24 Plugins **657**
Installer un Plugin Avec Composer . . . 657
Charger un Plugin . . . 658
Configuration du Plugin . . . 659
Utiliser un Plugin . . . 660
Créer Vos Propres Plugins . . . 661
Routes de Plugins . . . 662
Controllers du Plugin . . . 662
Models du Plugin . . . 664
Vues du Plugin . . . 665
Assets de Plugin . . . 666
Components, Helpers et Behaviors . . . 666
Etendez votre Plugin . . . 667
Publiez votre Plugin . . . 667

25 REST **669**
Mise en place Simple . . . 669

Accepter l'Input dans d'Autres Formats . . . 672
RESTful Routing . . . 672

26 Sécurité **673**
Security . . . 673

27 Sessions **677**
Configuration de Session . . . 677
Gestionnaires de Session intégrés & Configuration . . . 678
Configurer les Directives ini . . . 680
Créer un Gestionnaire de Session Personnalisé . . . 681
Accéder à l'Objet Session . . . 682
Lire & Ecrire les Données de Session . . . 683
Détruire la Session . . . 683
Faire une Rotation des Identificateurs de Session . . . 684
Messages Flash . . . 684

28 Testing **685**
Installer PHPUnit . . . 685
Tester la Configuration de la Base de Données . . . 686
Vérifier la Configuration Test . . . 687
Conventions des Cas de Test (TestCase) . . . 687
Créer Votre Premier Cas de Test . . . 687
Lancer les Tests . . . 689
Les Callbacks du Cycle de Vie des Cas de Test . . . 691
Fixtures . . . 691
Tester les Classes Table . . . 697
Test d'Intégrations des Controllers . . . 700
Tests d'Intégration de la Console . . . 710
Tester les Views . . . 715
Tester les Components . . . 715
Tester les Helpers . . . 717
Tester les Events . . . 719
Créer des Suites de Test (Test Suites) . . . 720
Créer des Tests pour les Plugins . . . 721
Générer des Tests avec Bake . . . 722
Intégration avec Jenkins . . . 722

29 Validation **725**
Créer les Validators . . . 725
Valider les Données . . . 732
Valider les Entities . . . 733
Règles de Validation du Cœur . . . 734

30 Classe App **735**
Trouver les Classes . . . 735
Trouver les Chemins vers les Namespaces . . . 736
Localiser les Plugins . . . 736
Localiser les Themes . . . 736
Charger les Fichiers de Vendor . . . 736

31 Collections **739**
Exemple Rapide . . . 739
Liste des Méthodes . . . 740
Faire une Itération . . . 740

Filtrer . . . 744
Agrégation . . . 746
Trier . . . 749
Utiliser des Données en Arbre . . . 750
Autres Méthodes . . . 752

32 Folder & File **759**
Utilisation Basique . . . 759
API de Folder . . . 760
L'API de File . . . 763

33 Hash **767**
Syntaxe de chemin Hash . . . 767

34 Client Http **783**
Faire des Requêtes . . . 783
Créer des Requêtes Multipart avec des Fichiers . . . 784
Envoyer des Corps de Requête . . . 785
Options de la Méthode Request . . . 786
Authentification . . . 786
Créer des Scoped Clients . . . 788
Configurer et Gérer les Cookies . . . 788
Objets Response . . . 789

35 Inflector **793**
Résumé des Méthodes d'Inflector et de leurs Sorties . . . 793
Créer des Formes Pluriel et Singulier . . . 794
Créer des Formes en CamelCase et en Underscore . . . 795
Créer des Formes Lisibles par l'Homme . . . 795
Créer des Formes pour les Tables et les Noms de Classe . . . 795
Créer des Noms de Variable . . . 796
Créer des Chaînes d'URL Safe . . . 796
Configuration d'Inflexion . . . 796

36 Number **797**
Formatage des Devises . . . 798
Paramétrage de la Devise par Défaut . . . 798
Formatage Des Nombres A Virgules Flottantes . . . 799
Formatage Des Pourcentages . . . 799
Interagir Avec Des Valeurs Lisibles Par L'Homme . . . 799
Formatage Des Nombres . . . 800
Formatage Des Différences . . . 801
Configurer le Formatage . . . 802

37 Objets Registry **803**
Charger les Objets . . . 803
Attraper les Callbacks . . . 804
Désactiver les Callbacks . . . 804

38 Text **805**
Convertion des Chaînes de Caractères en ASCII . . . 806
Créer des chaînes saines pour les URL . . . 806
Générer des UUIDs . . . 807
Parseur de chaînes simples . . . 807
Formater une chaîne . . . 807

Fixer la largeur d'un texte . . . 808
Subrillance de Sous-Chaîne . . . 809
Retirer les Liens . . . 809
Tronquer le Texte . . . 809
Tronquer une chaîne par la fin . . . 810
Générer un Extrait . . . 811
Convertir un tableau sous la forme d'une phrase . . . 811

39 Date & Time **813**
Créer des Instances Time . . . 814
Manipulation . . . 814
Formatage . . . 815
Conversion . . . 818
Comparer Avec le Present . . . 818
Comparer Avec les Intervals . . . 819
Dates . . . 819
Dates et Heures Immutables . . . 820
Accepter des Données Requêtées Localisées . . . 820

40 Xml **821**
Importer les données vers la classe Xml . . . 821
Transformer une Chaîne de Caractères XML en Tableau . . . 822
Transformer un tableau en une chaîne de caractères XML . . . 822

41 Globales & Fonctions **825**
Fonctions Globales . . . 825
Définitions des constantes du noyau . . . 827
Définition de Constantes de Temps . . . 828

42 Chronos **829**

43 Debug Kit **831**

44 Migrations **833**

45 ElasticSearch **835**

46 Annexes **837**
3.x Guide de Migration . . . 837
Informations générales . . . 874

PHP Namespace Index **877**

Index **879**

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\TableRegistry;

// Prior to 3.6 use TableRegistry::get('Users')
$users = TableRegistry::getTableLocator()->get('Users');
$query = $users->find();
foreach ($query 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\TableRegistry;

// Prior to 3.6 use TableRegistry::get('Users')
$users = TableRegistry::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 qu'elle interface de présentation nécessaire à votre application.

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

```
// Dans un fichier de template de view, nous afficherons un 'element' pour chaque␣
↪utilisateur (user).
<?php foreach ($users as $user): ?>
    <li class="user">
        <?= $this->element('user', ['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 = $this->Users->newEntity();
    if ($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);
}
```

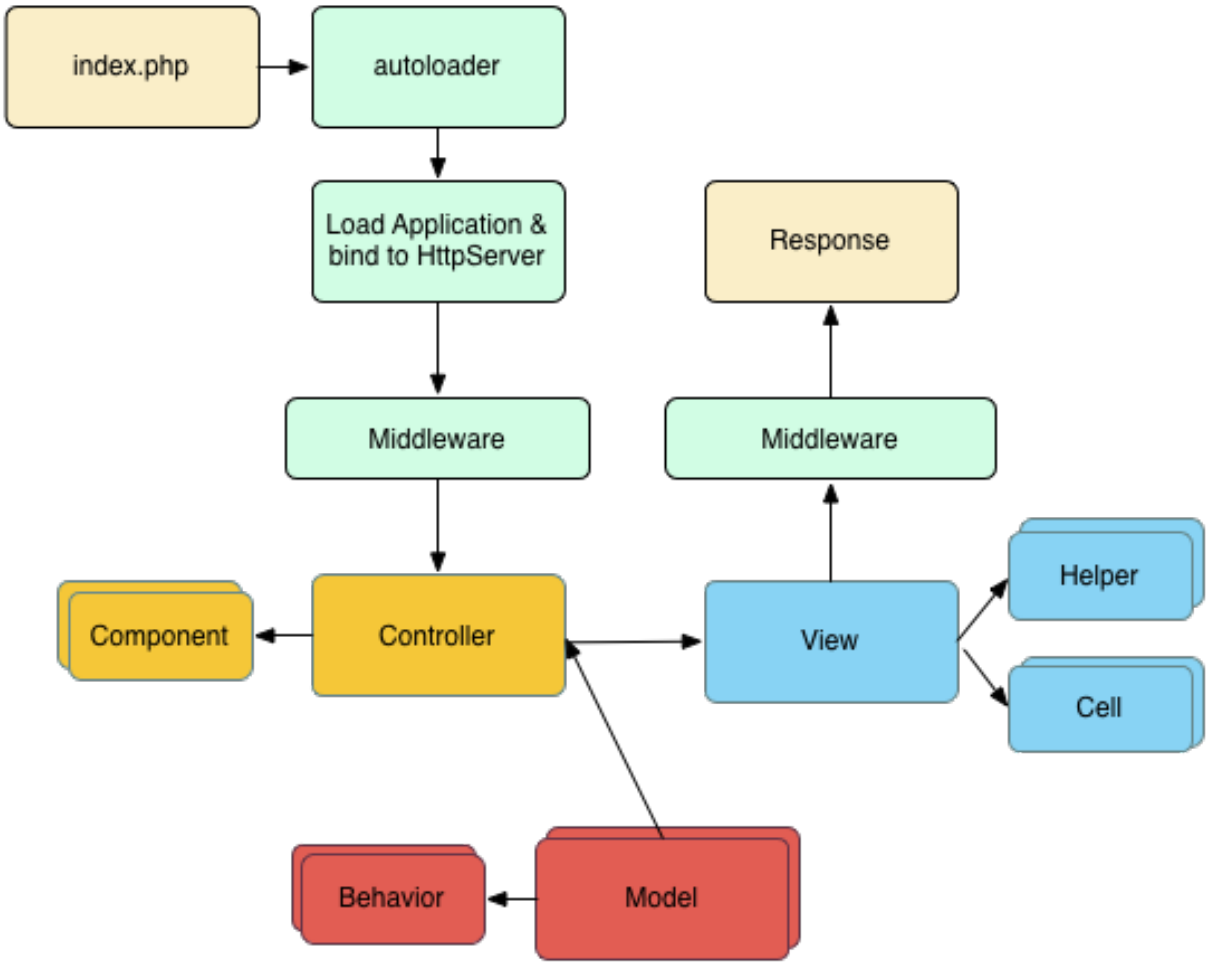
Notez que nous n'avons jamais explicitement rendu de view. Les conventions de CakePHP prendront soin de sélectionner la bonne view et de la rendre avec les données préparées avec `set()`.

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.


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

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

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.

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]

4. https://github.com/cakephp?tab=members
5. http://discourse.cakephp.org
6. https://stackoverflow.com/questions/tagged/cakephp/
7. http://cakephp-br.org

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]

Portugaise

— Groupe Google CakePHP protugais [16]

8. https://cakesf.slack.com/messages/denmark/
9. http://cakephp-fr.org
10. https://cakesf.slack.com/messages/german/
11. https://www.facebook.com/groups/146324018754907/
12. http://cakephp.ir
13. https://cakesf.slack.com/messages/netherlands/
14. https://cakesf.slack.com/messages/japanese/
15. https://www.facebook.com/groups/304490963004377/
16. http://groups.google.com/group/cakephp-pt

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 dans le code plus.

Les Conventions des Controllers

Les noms des classes de controller sont au pluriel, en CamelCase et se terminent par `Controller`. `UsersController` et `ArticleCategoriesController` 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` correspond à la méthode `view()` de `UsersController` sans rien modifier. 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 à mot unique renvoie vers un chemin URL en minuscules. Par exemple, `UsersController` (qui serait défini dans le nom de fichier **UsersController.php**) est accessible à l'adresse http://exemple.com/users.

Alors que vous pouvez router des controllers qui ont 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 `/article-categories/view-all` est la bonne forme pour accéder à l'action `ArticleCategoriesController::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*.

17. https://cakesf.slack.com/messages/spanish/
18. http://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-0 et 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.ctp**.
- 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 classe de model sont au pluriel, en CamelCase et finissent par `Table`. `UsersTable`, `ArticleCategoriesTable` et `UserFavoritePagesTable` en sont des exemples.

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`, `article_categories` et `user_favorite_pages`.

La convention est d'utiliser des mots anglais pour les noms de colonne et de table. Si vous utilisez des mots dans une autre langue, CakePHP ne va pas pouvoir convertir correctement les bonnes inflections (du singulier vers le pluriel et vice-versa). Dans certains cas, si vous souhaitez ajouter vos propres règles pour des mots d'une autre langue, vous pouvez utiliser la classe utilitaire *`Cake\Utility\Inflector`*. En plus de définir ces règles d'inflections personnalisées, cette classe va aussi vous permettre de vérifier que CakePHP comprend votre syntaxe personnalisée pour les mots pluriels et singuliers. Vous pouvez consulter la documentation sur *Inflector* pour plus d'informations.

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 `article_categories`, la clé étrangère sera `article_category_id`.

Les tables de jointure utilisées dans les relations BelongsToMany entre models doivent être nommées d'après le nom des tables qu'elles unissent ou la commande bake ne fonctionnera pas, dans l'ordre alphabétique (`articles_tags` plutôt que `tags_articles`). Dans le cas où vous souhaitez ajouter des données supplémentaires à la table intermédiaire, vous devrez ajouter un champ `id` à la table de jointure pour que `articles_tags` devienne une entity et la table aura au moins les clés `article_id`, `tag_id` et `id`.

En plus de l'utilisation des clés auto-incrémentées en tant que clés primaires, vous voudrez peut-être 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()`.

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 **src/Template/Articles/view_all.ctp**.

Le schéma classique est **src/Template/Controller/nom_de_fonction_avec_underscore.ctp**.

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 »
- 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 **src/Template/Articles/index.ctp**

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 (et automatiquement lié à la table "articles" dans la base) et rendue dans un fichier. 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.

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 avec 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 de votre application.
- 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. Modifier ces fichiers est déconseillé car composer écraser 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*.

Controller Contient les controllers et les components de votre application.

Locale Stocke les fichiers pour l'internationalisation.

Model Pour 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 *Outils de Console, Shells, & Tasks*.

Template Les fichiers de présentation se trouvent ici : elements, pages d'erreur, les layouts, et les fichiers de template de vue.

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

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 5.6 installé (en CLI). Votre version serveur de PHP doit au moins être aussi 5.6 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, il suffit simplement de lancer la commande suivante :

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

Ou vous pouvez télécharger `composer.phar` depuis le site de Composer [19].

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:^3.8 cms
```

Si vous avez téléchargé et utilisé l'Installer de Composer pour Windows [20], tapez la commande suivante dans votre terminal depuis le dossier d'installation (par exemple C :\wamp\www\dev\cakephp3) :

```
composer self-update && composer create-project --prefer-dist cakephp/app:^3.8 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
  /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 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.

19. https://getcomposer.org/download/
20. https://getcomposer.org/Composer-Setup.exe

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`). Exécutez ensuite la requête suivante pour créer les premières tables nécessaires au tutoriel :

```
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)
) CHARSET=utf8mb4;
```

(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_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());
```

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. Remplacer 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' => 'Cake\Database\Connection',
            'driver' => 'Cake\Database\Driver\Mysql',
            'persistent' => false,
            'host' => 'localhost',
            'username' => 'cakephp',
            'password' => 'AngelF00dC4k3~',
            'database' => 'cake_cms',
            '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 : Une copie du fichier de configuration par défaut peut être trouvée dans **config/app.default.php**.

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)
    {
        $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 notre 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
{
    protected $_accessible = [
        '*' => true,
        'id' => false,
        'slug' => false,
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ];
}
```

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 maintenant 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 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 **src/Template** et dans un dossier au nom du controller auquel ils sont attachés. Nous devons donc créer un dossier nommé "Articles" dans notre cas. Ajouter le code suivant dans ce fichier :

```
<!-- Fichier : src/Template/Articles/index.ctp -->

<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 rendent les tâches comme créer des liens, des formulaires et des éléments de paginations très faciles. 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 corrigez 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 utilisez 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 `$slug`, mais d'où vient-il ? Si un utilisateur requête `/articles/view/first-post`, alors la valeur "first-post" sera passé à `$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.

Création du template View

Créons le template de view pour notre action « view » dans **src/Template/Articles/view.ctp**.

```
<!-- Fichier : src/Template/Articles/view.ctp -->

<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 :

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

namespace App\Controller;

use App\Controller\AppController;

class ArticlesController extends AppController
{

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

            // L'écriture de 'user_id' en dur est temporaire et
            // sera supprimé 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 *FlashComponent* 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 magiquesde PHP [21]. 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 de 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 [22] 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 » :

```
<!-- Fichier : src/Template/Articles/add.ctp -->

<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 l'ouverture du form 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

21. http://php.net/manual/en/language.oop5.overloading.php#object.call
22. https://api.cakephp.org

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 **src/Template/Articles/index.ctp** 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 :

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

// Ajoutez ce "use" juste sous la déclaration du namespace
// pour importer la classe Text
use Cake\Utility\Text;

// 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. 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.'));
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    $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 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 : src/Template/Articles/edit.ctp -->

<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.

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

```
<!-- Fichier : src/Template/Articles/index.ctp (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>
```

(suite sur la page suivante)

(suite de la page précédente)

```
            <?= $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
        ->notEmpty('title')
        ->minLength('title', 10)
        ->maxLength('title', 255)

        ->notEmpty('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 taille.

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

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 : src/Template/Articles/index.ctp (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) ?>
```

(suite sur la page suivante)

(suite de la page précédente)

```
        </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 `View\Helper\FormHelper::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

3.0 Guide de Migration

Cette page résume les changements de CakePHP 2.x qui aidera à la migration d'un projet vers la version 3.0, ainsi qu'une référence pour être à jour des changements faits dans le cœur depuis la branche CakePHP 2.x. Assurez-vous de bien lire les autres pages de ce guide pour toutes les nouvelles fonctionnalités et les changements de l'API.

Prérequis

- CakePHP 3.x a besoin de la Version 5.4.16 ou supérieur de PHP.
- CakePHP 3.x a besoin de l'extension mbstring.
- CakePHP 3.x a besoin de l'extension intl.

Avertissement : CakePHP 3.0 ne fonctionnera pas si vous n'avez pas les prérequis ci-dessus.

Outil d'Upgrade

Alors que ce document couvre tous les changements non rétro-compatibles et les évolutions faites dans CakePHP 3.0, nous avons également créé une application de console pour vous aider à réaliser quelques changements qui consomment du temps. Vous pouvez Récupérer l'outil d'upgrade depuis Github [23].

23. https://github.com/cakephp/upgrade

Organisation des Répertoires de l'Application

L'organisation des répertoires de l'application a changé et suit maintenant PSR-4 [24]. Vous devez utiliser le projet de squelette d'application [25] comme point de référence lors de la mise à jour de votre application.

CakePHP doit être installé avec Composer

Puisque CakePHP ne peut plus être installé via PEAR, ou dans un répertoire partagé, ces options ne sont plus supportées. A la place, vous devez utiliser Composer [26] pour installer CakePHP dans votre application.

Namespaces (Espaces de Noms)

Toutes les classes du cœur de CakePHP sont maintenant dans des namespaces et suivent les spécifications du chargement PSR-4. Par exemple **src/Cache/Cache.php** est dans le namespace `Cake\Cache\Cache`. Les constantes globales et les méthodes de helper comme *__()* et *debug()* ne sont pas mis dans un namespace pour des raisons de commodité.

Constantes retirées

Les constantes dépréciées suivantes ont été retirées :

- `IMAGES`
- `CSS`
- `JS`
- `IMAGES_URL`
- `JS_URL`
- `CSS_URL`
- `DEFAULT_LANGUAGE`

Configuration

La configuration dans CakePHP 3.0 est significativement différente des versions précédentes. Vous devriez lire la documentation *Configuration* sur la façon dont la configuration est faite dans la version 3.0.

Vous ne pouvez plus utiliser `App::build()` pour configurer les chemins de classe supplémentaires. A la place, vous devez mapper les chemins supplémentaires en utilisant l'autoloader de votre application. Regardez la section sur *Chemins de Classe Supplémentaires* pour plus d'informations.

Trois nouvelles variables de configuration fournissent la configuration de chemin pour les plugins, les views et les fichiers de locales. Vous pouvez ajouter plusieurs chemins à `App.paths.templates`, `App.paths.plugins` et `App.paths.locales` pour configurer des chemins multiples pour respectivement les templates, les plugins et les fichiers de locales.

La clé de configuration `www_root` a été renommée `wwwRoot` par cohérence. Merci d'ajuster votre fichier de configuration **app.php** ainsi que chaque utilisation de``Configure : :read("App.wwwRoot")``.

24. http://www.php-fig.org/psr/psr-4/
25. https://github.com/cakephp/app
26. http://getcomposer.org

Nouvel ORM

CakePHP 3.0 dispose d'un nouvel ORM qui a été reconstruit de zéro. Le nouvel ORM est significativement différent et incompatible avec la version précédente. Mettre à jour vers le nouvel ORM nécessite des changements importants dans toute application qui souhaite être mise à jour. Regardez la nouvelle documentation *Accès Base de Données & ORM* pour des informations sur la façon d'utiliser le nouvel ORM.

Notions de base

- `LogError()` a été retirée, elle ne fournissait aucun bénéfice et n'était rarement/jamais utilisée.
- Les fonctions globales suivantes ont été retirées : `config()`, `cache()`, `clearCache()`, `convertSlashes()`, `am()`, `fileExistsInPath()`, `sortByKey()`.

Debugging

- `Configure::write('debug', $bool)` n'accepte plus 0/1/2. Un simple booléen est utilisé à la place pour changer entre le mode debug on et off.

Paramétrage/Configuration des Objets

- Les Objets utilisés dans CakePHP ont maintenant un système d'instance-configuration de stockage/récupération cohérent. Le code qui était auparavant accessible par exemple comme ceci : `$object->settings` devra être mis à jour en utilisant à la place `$object->config()`.

Cache

- Le moteur `Memcache` a été retiré, utilisez `Cake\Cache\Cache\Engine\Memcached` à la place.
- Les moteurs de Cache sont maintenant chargés automatiquement à la première utilisation.
- `Cake\Cache\Cache::engine()` a été ajoutée.
- *`Cake\Cache\Cache::enabled()`* a été ajoutée. celle-ci remplace l'option de configuration `Cache.disable`.
- *`Cake\Cache\Cache::enable()`* a été ajoutée.
- *`Cake\Cache\Cache::disable()`* a été ajoutée.
- Les configurations de Cache sont maintenant immutables. Si vous avez besoin de changer la configuration, vous devez d'abord retirer la configuration et la recréer. Cela évite les problèmes de synchronisation avec les options de configuration.
- `Cache::set()` a été retirée. Il est recommandé que vous créiez des configurations de cache multiples pour remplacer les réglages de configuration d'exécution, ce qui était auparavant possible avec `Cache::set()`.
- Toutes les sous-classes `CacheEngine` intègrent maintenant une méthode `config()`.
- *`Cake\Cache\Cache::readMany()`*, *`Cake\Cache\Cache::deleteMany()`*, et *`Cake\Cache\Cache::writeMany()`* ont été ajoutées.

Toutes les méthodes de `Cake\Cache\Cache\CacheEngine` sont maintenant responsables de la gestion du préfix de clé configuré. *`Cake\Cache\CacheEngine::write()`* ne permet plus de définir la durée d'écriture - la durée est prise par la configuration d'exécution du moteur de cache. Appeler une méthode de cache avec une clé vide va maintenant lancer `InvalidArgumentException`, au lieu de retourner `false`.

Core

App

- `App::pluginPath()` a été retirée. Utilisez `CakePlugin::path()` à la place.
- `App::build()` a été retirée.
- `App::location()` a été retirée.
- `App::paths()` a été retirée.
- `App::load()` a été retirée.
- `App::objects()` a été retirée.
- `App::RESET` a été retirée.
- `App::APPEND` a été retirée.
- `App::PREPEND` a été retirée.
- `App::REGISTER` a été retirée.

Plugin

- `Cake\Core\Plugin::load()` ne configure pas d'autoloader à moins que vous définissiez l'option `autoload` à `true`.
- Lors du chargement des plugins, vous ne pouvez plus fournir de callable.
- Lors du chargement des plugins, vous ne pouvez plus fournir un tableau de fichiers de configuration à charger.

Configure

- `Cake\Configure\PhpReader` renommé en `Cake\Core\Configure\EnginePhpConfig`
- `Cake\Configure\IniReader` renommé en `Cake\Core\Configure\EngineIniConfig`
- `Cake\Configure\ConfigReaderInterface` renommé en `Cake\Core\Configure\ConfigEngineInterface`
- *`Cake\Core\Configure::consume()`* a été ajoutée.
- *`Cake\Core\Configure::load()`* attend maintenant un suffixe du nom du fichier sans extension puisque celui-ci peut venir d'un moteur. Par exemple, l'utilisation de PhpConfig utilise `app` pour charger **app.php**.
- Définir une variable `$config` dans un fichier PHP de config est déprécié. `Cake\Core\Configure\EnginePhpConfig` attend maintenant le fichier de config pour retourner un tableau.
- Un nouveau moteur de config `Cake\Core\Configure\EngineJsonConfig` a été ajouté.

Object

La classe `Object` a été retirée. Elle contenait au début un tas de méthodes qui étaient utilisées dans plusieurs endroits à travers le framework. Les méthodes les plus utiles qui étaient utilisées ont été extraites dans des traits. Vous pouvez utiliser *`Cake\Log\LogTrait`* pour accéder à la méthode `log()`. `Cake\Routing\RequestActionTrait` fournit `requestAction()`.

Console

L'exécutable `cake` a été déplacé du répertoire `app/Console` vers le répertoire `bin` dans le squelette de l'application. Vous pouvez maintenant lancer la console de CakePHP avec `bin/cake`.

TaskCollection Remplacée

Cette classe a été renommée en `Cake\Console\TaskRegistry`. Regardez la section sur *Objets Registry* pour plus d'informations sur les fonctionnalités fournies par la nouvelle classe. Vous pouvez utiliser `cake upgrade rename_collections` pour vous aider à mettre à niveau votre code. Les Tasks n'ont plus accès aux callbacks, puisqu'il n'y avait jamais de callbacks à utiliser.

Shell

— `Shell::__construct()` a changé. Il prend maintenant une instance de `Cake\Console\ConsoleIo`.
— `Shell::param()` a été ajoutée pour un accès pratique aux paramètre.

De plus, toutes les méthodes du shell vont être transformées en camel case lors de leur appel. Par exemple, si vous avez une méthode `hello_world()` dans un shell et que vous l'appelez avec `bin/cake my_shell hello_world`, vous devez renommer la méthode en `helloWorld`. Il n'y a pas de changements nécessaires dans la façon d'appeler les commandes.

ConsoleOptionParser

— `ConsoleOptionParser::merge()` a été ajoutée pour fusionner les parsers.

ConsoleInputArgument

— `ConsoleInputArgument::isEqualTo()` a été ajoutée pour comparer deux arguments.

Shell / Task

Shells et Tasks ont été déplacés de `Console/Command` et `Console/Command/Task` vers `Shell` et `Shell/Task`.

ApiShell Retiré

ApiShell a été retiré puisqu'il ne fournit aucun bénéfice sur le fichier source lui-même et sur la documentation/l'API [27] en-ligne.

27. https://api.cakephp.org/

SchemaShell Removed

SchemaShell a été retiré puisqu'il n'a jamais été une implémentation de migrations de base de données complète et de meilleurs outils comme Phinx [28] ont émergé. Il a été remplacé par le Plugin de Migrations pour CakePHP [29] qui permet l'utilisation de Phinx [30] avec CakePHP.

ExtractTask

— `bin/cake i18n extract` n'inclut plus les messages de validation non traduits. Si vous voulez traduire les messages de validation, vous devez entourer ces messages dans des appels *__()* comme tout autre contenu.

BakeShell / TemplateTask

— Bake ne fait plus partie du code source du core et est remplacé par le Plugin CakePHP Bake [31]
— Les templates de bake ont été déplacés vers **src/Template/Bake**.
— La syntaxe des templates Bake utilise maintenant des balises de type erb (`<% %>`) pour désigner le templating.
— La commande `bake view` a été renommée `bake template`.

Event

La méthode `getEventManager()` a été retirée pour tous les objets qui l'avaient. Une méthode `eventManager()` est maintenant fournie par `EventManagerTrait`. `EventManagerTrait` contient la logique pour instancier et garder une référence d'un gestionnaire d'évènement local.

Le sous-système d'Event s'est vu retiré un certain nombre de fonctionnalités Lors du dispatching des événements, vous ne pouvez plus utiliser les options suivantes :

— `passParams` Cette option est maintenant toujours activée implicitement. Vous ne pouvez pas l'arrêter.
— `break` Cette option a été retirée. Vous devez maintenant arrêter les events.
— `breakOn` Cette option a été retirée. Vous devez maintenant arrêter les events.

Log

— Les configurations des logs sont maintenant immutables. Si vous devez changer la configuration, vous devez d'abord supprimer la configuration et la récréer. Cela évite les problèmes de synchronisation avec les options de configuration.
— Les moteurs de Log se chargent maintenant automatiquement à la première écriture dans les logs.
— `Cake\Log\Log::engine()` a été ajoutée.
— Les méthodes suivantes ont été retirées de *`Cake\Log\Log`* : : `defaultLevels()`, `enabled()`, `enable()`, `disable()`.
— Vous ne pouvez plus créer de niveaux personnalisés en utilisant `Log::levels()`.
— Lors de la configuration des loggers, vous devez utiliser `'levels'` au lieu de `'types'`.
— Vous ne pouvez plus spécifier de niveaux de log personnalisé. Vous devez utiliser les niveaux de log définis par défaut. Pour créer des fichiers de log personnalisés ou spécifiques à la gestion de différentes sections de votre application, vous devez utiliser les logging scopes. L'utilisation d'un niveau de log non-standard lancera maintenant une exception.

28. https://phinx.org/
29. https://github.com/cakephp/migrations
30. https://phinx.org/
31. https://github.com/cakephp/bake

- *Cake\Log\LogTrait* a été ajoutée. Vous pouvez utiliser ce trait dans vos classes pour ajouter la méthode `log()`.
- Le logging scope passé à *Cake\Log\Log::write()* est maintenant transmis à la méthode `write()` du moteur de log afin de fournir un meilleur contexte aux moteurs.
- Les moteurs de Log sont maintenant nécessaires pour intégrer `Psr\Log\LogInterface` plutôt que la propre `LogInterface` de CakePHP. En général, si vous étendez `Cake\Log\Engine\BaseEngine` vous devez juste renommer la méthode `write()` en `log()`.
- `Cake\Log\Engine\FileLog` écrit maintenant les fichiers dans `ROOT/logs` au lieu de `ROOT/tmp/logs`.

Routing

Paramètres Nommés

Les paramètres nommés ont été retirés dans 3.0. Les paramètres nommés ont été ajoutés dans 1.2.0 comme un "belle" version des paramètres query strings. Alors que le bénéfice visuel est discutable, les problèmes engendrés par les paramètres nommés ne le sont pas.

Les paramètres nommés nécessitaient une gestion spéciale dans CakePHP ainsi que toute librairie PHP ou JavaScript qui avaient besoin d'interagir avec eux, puisque les paramètres nommés ne sont implémentés ou compris par aucune librairie *exceptée* CakePHP. La complexité supplémentaire et le code nécessaire pour supporter les paramètres nommés ne justifiaient pas leur existence, et elles ont été retirées. A la place, vous devrez utiliser les paramètres standards de query string, ou les arguments passés. Par défaut `Router` traitera tous les paramètres supplémentaires de `Router::url()` comme des arguments de query string.

Puisque beaucoup d'applications auront toujours besoin de parser des URLs entrantes contenant des paramètres nommés, `Cake\Routing\Router::parseNamedParams()` a été ajoutée pour permettre une rétrocompatibilité avec les URLs existantes.

RequestActionTrait

- `Cake\Routing\RequestActionTrait::requestAction()` a connu quelques changements sur des options supplémentaires :
  - `options[url]` est maintenant `options[query]`.
  - `options[data]` est maintenant `options[post]`.
  - Les paramètres nommés ne sont plus supportés.

Router

- Les paramètres nommés ont été retirés, regardez ci-dessus pour plus d'informations.
- L'option `full_base` a été remplacée par l'options `_full`.
- L'option `ext` a été remplacée par l'option `_ext`.
- Les options `_scheme`, `_port`, `_host`, `_base`, `_full` et `_ext` ont été ajoutées.
- Les chaînes URLs ne sont plus modifiées en ajoutant les noms de plugin/controller/prefix.
- La gestion de route fallback par défaut a été retirée. Si aucune route ne correspond à un paramètre défini, `/` sera retourné.
- Les classes de route sont responsables pour *toutes* les générations d'URL incluant les paramètres de query string. Cela rend les routes bien plus puissantes et flexibles.
- Les paramètres persistants ont été retirés. Ils ont été remplacés par `Cake\Routing\Router::urlFilter()` qui est une meilleur façon plus flexible pour changer les URLs étant routées inversement.

- La signature de `Cake\Routing\Router::parseExtensions()` a changé en `parseExtensions(string|array $extensions = null, $merge = true)`. Elle ne prend plus d'arguments variables pour la spécification des extensions. Aussi, vous ne pouvez plus l'appeler sans paramètre pour parser toutes les extensions (en faisant cela, cela va retourner des extensions existantes qui sont définies). Vous avez besoin de faire une liste blanche des extensions que votre application accepte.
- `Router::parseExtensions()` **doit** être appelée avant que les routes ne soient connectées. Il ne modifie plus les routes existantes lors de son appel.
- `Router::setExtensions()` a été retirée. Utilisez `Cake\Routing\Router::parseExtensions()` à la place.
- `Router::resourceMap()` a été retirée.
- L'option `[method]` a été renommée en `_method`.
- La capacité pour faire correspondre les en-têtes arbitraires avec les paramètres de style `[]` a été retirée. Si vous avez besoin de parser/faire correspondre sur les conditions arbitraires, pensez à utiliser les classes de route personnalisées.
- `Router::promote()` a été retirée.
- `Router::parse()` va maintenant lancer une exception quand une URL ne peut pas être gérée par aucune route.
- `Router::url()` va maintenant lancer une exception quand aucune route ne matche un ensemble de paramètres.
- Les scopes de Routing ont été introduits. Les scopes de Routing vous permettent de garder votre fichier de routes DRY et donne au Router des indices sur la façon d'optimiser le parsing et le routing inversé des URLs.

Route

- `CakeRoute` a été renommée en `Route`.
- La signature de `match()` a changé en `match($url, $context = [])` Consultez `Cake\Routing\Route::match()` pour plus d'informations sur la nouvelle signature.

La Configuration des Filtres de Dispatcher a Changé

Les filtres de Dispatcher ne sont plus ajoutés à votre application en utilisant `Configure`. Vous les ajoutez maintenant avec `Cake\Routing\DispatcherFactory`. Cela signifie que si votre application utilisait `Dispatcher.filters`, vous devrez maintenant utiliser `Cake\Routing\DispatcherFactory::add()`.

En plus des changements de configuration, les filtres du dispatcher ont vu leurs conventions mises à jour, et des fonctionnalités ont été ajoutées. Consultez la documentation *Filtres du Dispatcher* pour plus d'informations.

FilterAssetFilter

- Les assets des Plugin & theme gérés par AssetFilter ne sont plus lus via `include`, à la place ils sont traités comme de simples fichiers texte. Cela règle un certain nombre de problèmes avec les librairies JavaScript comme TinyMCE et les environnements avec short_tags activé.
- Le support pour la configuration de `Asset.filter` et les hooks ont été retirés. Cette fonctionnalité peut être remplacée par un plugin ou un filtre dispatcher.

Network

Request

- `CakeRequest` a été renommé en `Cake\Network\Request`.
- `Cake\Network\Request::port()` a été ajoutée.
- `Cake\Network\Request::scheme()` a été ajoutée.
- `Cake\Network\Request::cookie()` a été ajoutée.
- `Cake\Network\Request::$trustProxy` a été ajoutée. Cela rend la chose plus facile pour mettre les applications CakePHP derrière les load balancers.
- `Cake\Network\Request::$data` n'est plus fusionnée avec la clé de données préfixés, puisque ce préfixe a été retiré.
- `Cake\Network\Request::env()` a été ajoutée.
- `Cake\Network\Request::acceptLanguage()` a été changée d'une méthode static en non static.
- Le détecteur de Request pour « mobile » a été retiré du cœur. A la place le template de app ajoute des détecteurs pour « mobile » et « tablet » en utilisant la lib `MobileDetect`.
- La méthode `onlyAllow()` a été renommée en `allowMethod()` et n'accepte plus « var args ». Tous les noms de méthode doivent être passés en premier argument, soit en chaîne de caractère, soit en tableau de chaînes.

Response

- Le mapping du mimetype `text/plain` en extension `csv` a été retiré. En conséquence, `Cake\Controller\Component\RequestHandlerComponent` ne définit pas l'extension en `csv` si l'en-tête `Accept` contient le mimetype `text/plain` ce qui était une nuisance habituelle lors de la réception d'une requête jQuery XHR.

Sessions

La classe session n'est plus statique, à la place, la session est accessible à travers l'objet request. Consultez la documentation *Sessions* sur l'utilisation de l'objet session.

- `Cake\Network\Session` et les classes de session liées ont été déplacées dans le namespace `Cake\Network`.
- `SessionHandlerInterface` a été retirée en faveur de celui fourni par PHP.
- La propriété `Session::$requestCountdown` a été retirée.
- La fonctionnalité de session checkAgent a été retirée. Elle entrainait un certain nombre de bugs quand le chrome frame, et flash player sont impliqués.
- Le nom de la table de la base de données des sessions est maintenant `sessions` plutôt que `cake_sessions`.
- Le timeout du cookie de session est automatiquement mis à jour en tandem avec le timeout dans les données de session.
- Le chemin pour le cookie de session est maintenant par défaut le chemin de l'application plutôt que « / ». De plus, une nouvelle variable de configuration `Session.cookiePath` a été ajoutée pour personnaliser le chemin du cookie.
- Une nouvelle méthode `Cake\Network\Session::consume()` a été ajoutée pour permettre de lire et supprimer les données de session en une seule étape.

Network\Http

— `HttpSocket` est maintenant `Cake\Network\Http\Client`.
— HttpClient a été réécrit de zéro. Il a une API plus simple/facile à utiliser, le support pour les nouveaux systèmes d'authentification comme OAuth, et les uploads de fichier. Il utilise les APIs de PHP en flux donc il n'est pas nécessaire d'avoir cURL. Regardez la documentation *Client Http* pour plus d'informations.

Network\Email

— `Cake\Network\Email\Email::config()` est utilisée maintenant pour définir les profiles de configuration. Ceci remplace les classes `EmailConfig` des précédentes versions. versions.
— `Cake\Network\Email\Email::profile()` remplace `config()` comme façon de modifier les options de configuration par instance.
— `Cake\Network\Email\Email::drop()` a été ajoutée pour permettre le retrait de la configuration d'email.
— `Cake\Network\Email\Email::configTransport()` a été ajoutée pour permettre la définition de configurations de transport. Ce changement retire les options de transport des profiles de livraison et vous permet de réutiliser les transports à travers les profiles d'email.
— `Cake\Network\Email\Email::dropTransport()` a été ajoutée pour permettre le retrait de la configuration du transport.

Controller

Controller

— Les propriétés `$helpers`, `$components` sont maintenant fusionnées avec **toutes** les classes parentes, pas seulement `AppController` et le app controller du plugin. Les propriétés sont fusionnées de manière différente par rapport à aujourd'hui. Plutôt que d'avoir comme actuellement les configurations de toutes les classes fusionnées, la configuration définie dans la classe enfante sera utilisée. Cela signifie que si vous avez une configuration définie dans votre AppController, et quelques configurations définies dans une sous-classe, seule la configuration de la sous-classe sera utilisée.
— `Controller::httpCodes()` a été retirée, utilisez `Cake\Network\Response::httpCodes()` à la place.
— `Controller::disableCache()` a été retirée, utilisez `Cake\Network\Response::disableCache()` à la place.
— `Controller::flash()` a été retirée. Cette méthode était rarement utilisée dans les vraies applications et ne n'avait plus aucun intérêt.
— `Controller::validate()` et `Controller::validationErrors()` ont été retirées. Il y avait d'autres méthodes laissées depuis l'époque de 1.x, où les préoccupations des models + controllers étaient bien plus étroitement liées.
— `Controller::loadModel()` charge maintenant les objets table.
— La propriété `Controller::$scaffold` a été retirée. Le scaffolding dynamique a été retiré du cœur de CakePHP. Un plugin de scaffolding appelé CRUD se trouve ici : https://github.com/FriendsOfCake/crud
— La propriété `Controller::$ext` a été retirée. Vous devez maintenant étendre et surcharger la propriété `View::$_ext` si vous voulez utiliser une extension de fichier de view autre que celle par défaut.
— La propriété `Controller::$Components` a été retirée et remplacée par `_components`. Si vous avez besoin de charger les components à la volée, vous devez utiliser `$this->loadComponent()` dans votre controller.

— La signature de *Cake\Controller\Controller::redirect()* a été changée en `Controller::redirect(string|array $url, int $status = null)`. Le 3ème argument `$exit` a été retiré. La méthode ne peut plus envoyer la réponse et sortir du script, à la place elle retourne une instance de `Response` avec les en-têtes appropriés définis.
— Les propriétés magiques `base`, `webroot`, `here`, `data`, `action` et `params` ont été retirées. Vous pouvez accéder à toutes ces propriétés dans `$this->request` à la place.
— Les méthodes préfixées avec underscore des controllers comme `_someMethod()` ne sont plus considérées comme des méthodes privées. Utilisez les bons mots clés de visibilité à la place. Seules les méthodes publiques peuvent être utilisées comme action de controller.

Scaffold retiré

Le scaffolding dynamique dans CakePHP a été retiré du cœur de CakePHP. Il était peu fréquemment utilisé, et n'avait jamais pour intention d'être utilisé en mode production. Un plugin de scaffolding appelé CRUD se trouve ici : https://github.com/FriendsOfCake/crud

ComponentCollection remplacée

Cette classe a été remplacée en `Cake\Controller\ComponentRegistry`. Regardez la section sur *Objets Registry* pour plus d'informations sur les fonctionnalités fournies par cette nouvelle classe. Vous pouvez utiliser `cake upgrade rename_collections` pour vous aider à mettre à niveau votre code.

Component

— La propriété `_Collection` est maintenant `_registry`. Elle contient maintenant une instance de `Cake\Controller\ComponentRegistry`.
— Tous les components doivent maintenant utiliser la méthode `config()` pour récupérer/définir la configuration.
— La configuration par défaut pour les components doit être définie dans la propriété `$_defaultConfig`. Cette propriété est automatiquement fusionnée avec toute configuration fournie au constructeur.
— Les options de configuration ne sont plus définies en propriété public.
— La méthode `Component::initialize()` n'est plus un listener d'event. A la place, c'est un hook post-constructeur comme `Table::initialize()` et `Controller::initialize()`. La nouvelle méthode `Component::beforeFilter()` est liée au même évènement que `Component::initialize()`. La méthode initialize devrait avoir la signature suivante `initialize(array $config)`.

Controller\Components

CookieComponent

— Utilise `Cake\Network\Request::cookie()` pour lire les données de cookie, ceci facilite les tests, et permet de définir les cookies pour ControllerTestCase.
— Les Cookies chiffrés dans les versions précédentes de CakePHP utilisant la méthode `cipher()` sont maintenant illisible parce que `Security::cipher()` a été retirée. Vous aurez besoin de re-chiffrer les cookies avec la méthode `rijndael()` ou `aes()` avant mise à jour.
— `CookieComponent::type()` a été retirée et remplacée par la donnée de configuration accessible avec `config()`.
— `write()` ne prend plus de paramètres `encryption` ou `expires`. Ces deux-là sont maintenant gérés avec des données de config. Consultez *CookieComponent* pour plus d'informations.

— Le chemin pour les cookies sont maintenant par défaut le chemin de l'app plutôt que « / ».

AuthComponent

— `Default` est maintenant le hasher de mot de passe par défaut utilisé par les classes d'authentification. Si vous voulez continuer à utiliser le hashage SHA1 utilisé dans 2.x utilisez `'passwordHasher' => 'Weak'` dans votre configuration d'authenticator.
— Un nouveau `FallbackPasswordHasher` a été ajouté pour aider les utilisateurs à migrer des vieux mots de passe d'un algorithm à un autre. Consultez la documentation d'AuthComponent pour plus d'informations.
— La classe `BlowfishAuthenticate` a été retirée. Utilisez juste `FormAuthenticate`.
— La classe `BlowfishPasswordHasher` a été retirée. Utilisez `DefaultPasswordHasher` à la place.
— La méthode `loggedIn()` a été retirée. Utilisez `user()` à la place.
— Les options de configuration ne sont plus définies en propriété public.
— Les méthodes `allow()` et `deny()` n'acceptent plus « var args ». Tous les noms de méthode ont besoin d'être passés en premier argument, soit en chaîne, soit en tableau de chaînes.
— La méthode `login()` a été retirée et remplacée par `setUser()`. Pour connecter un utilisateur, vous devez maintenant appeler `identify()` qui retourne les informations d'utilisateur en cas de succès d'identification et utilise ensuite `setUser()` pour sauvegarder les informations de session pour la persistance au cours des différentes requêtes.
— `BaseAuthenticate::_password()` a été retirée. Utilisez `PasswordHasher` à la place.
— `BaseAuthenticate::logout()` a été retirée.
— `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.

Les classes liées à ACL ont été déplacées dans un plugin séparé. Les hashers de mot de passe, l'Authentification et les fournisseurs d'Authorisation ont été déplacés vers le namespace `\Cake\Auth`. Vous devez aussi déplacer vos providers et les hashers dans le namespace `App\Auth`.

RequestHandlerComponent

— Les méthodes suivantes ont été retirées du component RequestHandler : : `isAjax()`, `isFlash()`, `isSSL()`, `isPut()`, `isPost()`, `isGet()`, `isDelete()`. Utilisez la méthode `Cake\Network\Request::is()` à la place avec l'argument pertinent.
— `RequestHandler::setContent()` a été retirée, utilisez `Cake\Network\Response::type()` à la place.
— `RequestHandler::getReferer()` a été retirée, utilisez `Cake\Network\Request::referer()` à la place.
— `RequestHandler::getClientIP()` a été retirée, utilisez `Cake\Network\Request::clientIp()` à la place.
— `RequestHandler::mapType()` a été retirée, utilisez `Cake\Network\Response::mapType()` à la place.
— Les options de configuration ne sont plus définies en propriété public.

SecurityComponent

- Les méthodes suivantes et leurs propriétés liées ont été retirées du component Security : `requirePost()`, `requireGet()`, `requirePut()`, `requireDelete()`. Utilisez `Cake\Network\Request::onlyAllow()` à la place.
- `SecurityComponent::$disabledFields()` a été retirée, utilisez `SecurityComponent::$unlockedFields()`.
- Les fonctionnalités liées au CSRF dans SecurityComponent ont été extraites et déplacées dans un CsrfComponent séparé. Ceci vous permet d'utiliser une protection CSRF sans avoir à utiliser la prévention de falsification de formulaire.
- Les options de Configuration ne sont plus définies comme des propriétés publiques.
- Les méthodes `requireAuth()` et `requireSecure()` n'acceptent plus « var args ». Tous les noms de méthode ont besoin d'être passés en premier argument, soit en chaîne, soit en tableau de chaînes.

SessionComponent

- `SessionComponent::setFlash()` est déprécié. Vous devez utiliser *FlashComponent* à la place.

Error

Les ExceptionRenderers personnalisées doivent maintenant soit retourner un objet `Cake\\Network\\Response`, soit une chaîne de caractère lors du rendu des erreurs. Cela signifie que toutes les méthodes gérant des exceptions spécifiques doivent retourner une réponse ou une valeur.

Model

La couche Model de 2.x a été entièrement réécrite et remplacée. Vous devriez regarder le *Guide de Migration du Nouvel ORM* pour plus d'informations sur la façon d'utiliser le nouvel ORM.

- La classe `Model` a été retirée.
- La classe `BehaviorCollection` a été retirée.
- La classe `DboSource` a été retirée.
- La classe `Datasource` a été retirée.
- Les différentes sources de données des classes ont été retirées.

ConnectionManager

- ConnectionManager a été déplacée dans le namespace `Cake\\Datasource`.
- ConnectionManager a eu les méthodes suivantes retirées :
  - `sourceList`
  - `getSourceName`
  - `loadDataSource`
  - `enumConnectionObjects`
- `Database\ConnectionManager::config()` a été ajoutée et est maintenant la seule façon de configurer les connections.
- `Database\ConnectionManager::get()` a été ajoutée. Elle remplace `getDataSource()`.
- `Database\ConnectionManager::configured()` a été ajoutée. Celle-ci avec `config()` remplace `sourceList()` & `enumConnectionObjects()` avec une API plus standard et cohérente.
- `ConnectionManager::create()` a été retirée. Il peut être remplacé par `config($name, $config)` et `get($name)`.

Behaviors

— Les méthodes préfixées avec underscore des behaviors comme `_someMethod()` ne sont plus considérées comme des méthodes privées. Utilisez les bons mots clés à la place.

TreeBehavior

TreeBehavior a été complètement réécrit pour utiliser le nouvel ORM. Bien qu'il fonctionne de la même manière que dans 2.x, certaines méthodes ont été renommées ou retirées :

— `TreeBehavior::children()` est maintenant un finder personnalisé `find('children')`.
— `TreeBehavior::generateTreeList()` est maintenant un finder personnalisé `find('treeList')`.
— `TreeBehavior::getParentNode()` a été retirée.
— `TreeBehavior::getPath()` est maintenant un finder personnalisé `find('path')`.
— `TreeBehavior::reorder()` a été retirée.
— `TreeBehavior::verify()` a été retirée.

TestSuite

TestCase

— `_normalizePath()` a été ajoutée pour permettre aux tests de comparaison de chemin de se lancer pour tous les systèmes d'exploitation selon la configuration de leur DS (\ dans Windows vs / dans UNIX, par exemple).

Les méthodes d'assertion suivantes ont été retirées puisque cela faisait longtemps qu'elles étaient dépréciées et remplacées par leurs nouvelles homologues de PHPUnit :

— `assertEqual()` en faveur de `assertEquals()`
— `assertNotEqual()` en faveur de `assertNotEquals()`
— `assertIdentical()` en faveur de `assertSame()`
— `assertNotIdentical()` en faveur de `assertNotSame()`
— `assertPattern()` en faveur de `assertRegExp()`
— `assertNoPattern()` en faveur de `assertNotRegExp()`
— `assertReference()` if favor of `assertSame()`
— `assertIsA()` en faveur de `assertInstanceOf()`

Notez que l'ordre des paramètres de certaines méthodes a été changé, par ex : `assertEqual($is, $expected)` devra maintenant être `assertEquals($expected, $is)`.

Les méthodes d'assertion suivantes ont été dépréciées et seront retirées dans le futur :

— `assertWithinMargin()` en faveur de `assertWithinRange()`
— `assertTags()` en faveur de `assertHtml()`

Les deux méthodes de remplacement changent aussi l'ordre des arguments pour avoir une méthode d'API assert cohérente avec `$expected` en premier argument.

Les méthodes d'assertion suivantes ont été ajoutées :

— `assertNotWithinRange()` comme contrepartie de `assertWithinRange()`

View

Les Themes sont maintenant purement des Plugins

Avoir des themes et des plugins comme façon de créer des composants d'applications modulaires nous semblait limité et cela apportait de la confusion. Dans CakePHP 3.0, les themes ne se trouvent plus **dans** l'application. A la place, ce sont des plugins autonomes. Cela résout certains problèmes liés aux themes :

- Vous ne pouviez pas mettre les themes *dans* les plugins.
- Les Themes ne pouvaient pas fournir de helpers, ou de classes de vue personnalisée.

Ces deux problèmes sont résolus en convertissant les themes en plugins.

Les Dossiers de View Renommés

Les dossiers contenant les fichiers de vue vont maintenant dans **src/Template** à la place de **src/View**. Ceci a été fait pour séparer les fichiers de vue des fichiers contenant des classes php (par ex les classes Helpers et View).

Les dossiers de View suivants ont été renommés pour éviter les collisions avec les noms de controller :

- `Layouts` est maintenant `Layout`
- `Elements` est maintenant `Element`
- `Errors` est maintenant `Error`
- `Emails` est maintenant `Email` (idem pour `Email` dans `Layout`)

HelperCollection remplacée

Cette classe a été renommée en `Cake\View\HelperRegistry`. Regardez la section sur *Objets Registry* pour plus d'informations sur les fonctionnalités fournies par la nouvelle classe. Vous pouvez utiliser `cake upgrade rename_collections` pour vous aider à mettre à niveau votre code.

View Class

- La clé `plugin` a été retirée de l'argument `$options` de *`Cake\View\View::element()`*. Spécifiez le nom de l'element comme `SomePlugin.element_name` à la place.
- `View::getVar()` a été retirée, utilisez `Cake\View\View::get()` à la place.
- `View::$ext` a été retirée et à la place une propriété protected `View::$_ext` a été ajoutée.
- `View::addScript()` a été retirée. Utilisez *Utiliser les Blocks de Vues* à la place.
- Les propriétés magiques `base`, `webroot`, `here`, `data`, `action` et `params` ont été retirées. Vous pouvez accéder à toutes ces propriétés dans `$this->request` à la place.
- `View::start()` n'ajoute plus à un block existant. A la place, il va écraser le contenu du block quand end est appelé. Si vous avez besoin de combiner les contenus de block, vous devrez récupérer le contenu du block lors de l'appel de start une deuxième fois ou utiliser le mode de capture `append()`.
- `View::prepend()` n'a plus de mode de capture.
- `View::startIfEmpty()` a été retirée. maintenant que start() écrase toujours startIfEmpty n'a plus d'utilité.
- La propriété `View::$Helpers` a été retirée et remplacée par `_helpers`. Si vous avez besoin de charger les helpers à la volée, vous devrez utiliser `$this->addHelper()` dans vos fichiers de view.
- `View` lancera des `Cake\View\Exception\MissingTemplateException` lorsque des templates sont au lieu de `MissingViewException`.

ViewBlock

— `ViewBlock::append()` a été retirée, utilisez `Cake\ViewViewBlock::concat()` à la place. Cependant, `View::append()` existe encore.

JsonView

— Par défaut les données JSON vont maintenant avoir des entités HTML encodées. Ceci empêche les problèmes possible de XSS quand le contenu de la view JSON est intégrée dans les fichiers HTML.
— `Cake\View\JsonView` supporte maintenant la variable de view `_jsonOptions`. Ceci vous permet de configurer le masque utilisé lors de la génération de JSON.

XmlView

— `Cake\View\XmlView` supporte maintenant la variable de view `_xmlOptions`. Ceci vous permet de configurer les options utilisées lors de la génération de XML.

View\Helper

— La propriété `$settings` est maintenant appelée `$_config` et peut être accessible via la méthode `config()`.
— Les options de configuration ne sont plus définies en propriété public.
— `Helper::clean()` a été retirée. Il n'était jamais assez robuste pour complètement empêcher XSS. A la place, vous devez échapper le contenu avec *h* ou utiliser une librairie dédiée comme HTMLPurifier.
— `Helper::output()` a été retirée. Cette méthode a été dépréciée dans 2.x.
— Les méthodes `Helper::webroot()`, `Helper::url()`, `Helper::assetUrl()`, `Helper::assetTimestamp()` ont été déplacées vers le nouveau helper *Cake\View\Helper\UrlHelper*. `Helper::url()` est maintenant disponible dans *Cake\View\Helper\UrlHelper::build()*.
— Les accesseurs magiques pour les propriétés dépréciées ont été retirés. Les propriétés suivantes ont maintenant besoin d'être accédées à partir de l'objet request :
  — base
  — here
  — webroot
  — data
  — action
  — params

Helper

Les méthodes suivantes de Helper ont été retirées :
— `Helper::setEntity()`
— `Helper::entity()`
— `Helper::model()`
— `Helper::field()`
— `Helper::value()`
— `Helper::_name()`
— `Helper::_initInputField()`
— `Helper::_selectedArray()`

Ces méthodes étaient des parties uniquement utilisées par FormHelper, et faisaient partie des continuelles fonctionnalités des champs qui devenaient problématiques au fil du temps. FormHelper ne s'appuie plus sur ces méthodes et leur complexité n'est plus nécessaire.

Les méthodes suivantes ont été retirées :
- `Helper::_parseAttributes()`
- `Helper::_formatAttribute()`

Ces méthodes se trouvent dans la classe `StringTemplate` que les helpers utilisent fréquemment. Regardez `StringTemplateTrait` pour intégrer les templates de chaîne dans vos propres helpers.

FormHelper

FormHelper a été entièrement réécrite pour 3.0. Il amène quelques grands changements :
- FormHelper fonctionne avec le nouvel ORM. Mais il a un système extensible pour être intégré avec d'autres ORMs ou sources de données.
- FormHelper dispose d'une fonctionnalité de système de widget extensible qui vous permet de créer de nouveaux widgets d'input personnalisés et d'améliorer ceux intégrés.
- Les templates de chaîne sont un élément fondateur du helper. Au lieu de tableaux imbriqués ensemble partout, la plupart du HTML que FormHelper génère peut être personnalisé dans un endroit central en utilisant les ensembles de template.

En plus de ces grands changements, quelques plus petits changements finaux ont été aussi faits. Ces changements devraient aider le streamline HTML que le FormHelper génère et réduire les problèmes que les gens ont eu dans le passé :
- Le prefix `data[` a été retiré de tous les inputs générés. Le prefix n'a plus de réelle utilité.
- Les diverses méthodes d'input autonomes comme `text()`, `select()` et autres ne génèrent plus d'attributs id.
- L'option `inputDefaults` a été retirée de `create()`.
- Les options `default` et `onsubmit` de `create()` ont été retirées. A la place, vous devez utiliser le binding d'event Javascript ou définir tout le code js nécessaire pour `onsubmit`.
- `end()` ne peut plus faire des boutons. Vous devez créer des buttons avec `button()` ou `submit()`.
- `FormHelper::tagIsInvalid()` a été retirée. Utilisez `isFieldError()` à la place.
- `FormHelper::inputDefaults()` a été retirée. Vous pouvez utiliser `templates()` pour définir/améliorer les templates que FormHelper utilise.
- Les options `wrap` et `class` ont été retirées de la méthode `error()`.
- L'option `showParents` a été retirée de select().
- Les options `div`, `before`, `after`, `between` et `errorMessage` ont été retirées de `input()`. Vous pouvez utiliser les templates pour mettre à jour le HTML qui l'entoure. L'option `templates` vous permet de surcharger les templates chargés pour un input.
- Les options `separator`, `between`, et `legend` ont été retirées de `radio()`. Vous pouvez maintenant utiliser les templates pour changer le HTML qui l'entoure.
- Le paramètre `format24Hours` a été retiré de `hour()`. Il a été remplacé par l'option `format`.
- Les paramètres `minYear` et `maxYear` ont été retirés de `year()`. Ces deux paramètres peuvent maintenant être fournis en options.
- Les paramètres `dateFormat` et `timeFormat` ont été retirés de `datetime()`. Vous pouvez maintenant utiliser les templates pour définir l'ordre dans lequel les inputs doivent être affichés.
- `submit()` a eu les options `div`, `before` et `after` retirées. Vous pouvez personnaliser le template `submitContainer` pour modifier ce contenu.
- La méthode `inputs()` n'accepte plus `legend` et `fieldset` dans le paramètre `$fields`, vous devez utiliser le paramètre `$options`. Elle nécessite aussi que le paramètre `$fields` soit un tableau. Le paramètre `$blacklist` a été retiré, la fonctionnalité a été remplacée en spécifiant `'field' => false` dans le paramètre `$fields`.
- Le paramètre `inline` a été retiré de la méthode postLink(). Vous devez utiliser l'option `block` à la place. Définir `block => true` va émuler le comportement précédent.

— Le paramètre `timeFormat` pour `hour()`, `time()` et `dateTime()` est par maintenant par défaut à 24, en accord avec l'ISO 8601.
— L'argument `$confirmMessage` de *`Cake\View\Helper\FormHelper::postLink()`* a été retiré. Vous devez maintenant utiliser la clé `confirm` dans `$options` pour spécifier le message.
— Les inputs de type checkbox et boutons radios types sont maintenant générées *à l'intérieur* de balises label par défaut. Ceci aide à accroître la compatibilité avec les librairies CSS populaires telles que Bootstrap [32] et Foundation [33].
— Les tags de templates sont maintenant tous écrits en *camelBack*. Les tags pre-3.0 `formstart`, `formend`, `hiddenblock` et `inputsubmit` sont maintenant `formStart`, `formEnd`, `hiddenBlock` et `inputSubmit`. Pensez à bien les changer s'ils sont personnalisés dans votre application.

Il est recommandé que vous regardiez la documentation *Form* pour plus de détails sur la façon d'utiliser le FormHelper dans 3.0.

HtmlHelper

— `HtmlHelper::useTag()` a été retirée, utilisez `tag()` à la place.
— `HtmlHelper::loadConfig()` a été retirée. La personnalisation des tags peut être faîte en utilisant `templates()` ou la configuration `templates`.
— Le deuxième paramètre `$options` pour `HtmlHelper::css()` doit maintenant toujours être un tableau comme c'est écrit dans la documentation.
— Le premier paramètre `$data` pour `HtmlHelper::style()` doit maintenant toujours être un tableau comme c'est écrit dans la documentation.
— Le paramètre `inline` a été retiré des méthodes meta(), css(), script(), scriptBlock(). Vous devrez utiliser l'option `block` à la place. Définir `block => true` va émuler le comportement précédent.
— `HtmlHelper::meta()` nécessite maintenant que `$type` soit une chaîne de caractère. Les options supplémentaires peuvent en outre être passées dans `$options`.
— `HtmlHelper::nestedList()` nécessite maintenant que `$options` soit un tableau. Le quatrième argument pour le niveau de tag a été retiré et il a été inclus dans le tableau `$options`.
— L'argument `$confirmMessage` de *`Cake\View\Helper\HtmlHelper::link()`* a été retiré. Vous devez maintenant utiliser la clé `confirm` dans `$options` pour spécifier le message.

PaginatorHelper

— `link()` a été retirée. Il n'était plus utilisé par le helper en interne. Il était peu utilisé dans le monde des utilisateurs de code, et ne correspondait plus aux objectifs du helper.
— `next()` n'a plus les options "class", ou "tag". Il n'a plus d'arguments désactivés. A la place, les templates sont utilisés.
— `prev()` n'a plus les options "class", ou "tag". Il n'a plus d'arguments désactivés. A la place, les templates sont utilisés.
— `first()` n'a plus les options "after", "ellipsis", "separator", "class", ou "tag".
— `last()` n'a plus les options "after", "ellipsis", "separator", "class", ou "tag".
— `numbers()` n'a plus les options "separator", "tag", "currentTag", "currentClass", "class", "tag", "ellipsis". Ces options sont maintenant accessibles à travers des templates. Le paramètre `$options` doit maintenant être un tableau.
— Les placeholders de style `%page%` ont été retirés de *`Cake\View\Helper\PaginatorHelper::counter()`*. Utilisez les placeholders de style `{{page}}` à la place.
— `url()` a été renommée en `generateUrl()` pour éviter des clashes de déclaration de méthode avec `Helper::url()`.

Par défaut, tous les liens et le texte inactif sont entourés d'elements `<li>`. Ceci aide à écrire plus facilement du CSS, et améliore la compatibilité avec des frameworks populaires.

32. http://getbootstrap.com/
33. http://foundation.zurb.com/

A la place de ces diverses options dans chaque méthode, vous devez utiliser la fonctionnalité des templates. Regardez les informations de la documentation *Templates de PaginatorHelper* sur la façon d'utiliser les templates.

TimeHelper

- `TimeHelper::__set()`, `TimeHelper::__get()`, et `TimeHelper::__isset()` ont été retirées. Celles-ci étaient des méthodes magiques pour des attributs dépréciés.
- `TimeHelper::serverOffset()` a été retirée. Elle entraînait de mauvaises utilisations mathématiques de time.
- `TimeHelper::niceShort()` a été retirée.

NumberHelper

- `NumberHelper::format()` nécessite maintenant que `$options` soit un tableau.

SessionHelper

- `SessionHelper` est déprécié. Vous pouvez utiliser `$this->request->session()` directement, et la fonctionnalité de message flash a été déplacée dans *Flash* à la place.

JsHelper

- `JsHelper` et tous les moteurs associés ont été retirés. il pouvait seulement générer un tout petit nombre de code Javascript pour la librairie sélectionnée et essayer de générer tout le code Javascript en utilisant le helper devenait souvent difficile. Il est maintenant recommandé d'utiliser directement la librairie Javascript de votre choix.

CacheHelper Retiré

CacheHelper a été retiré. La fonctionnalité de cache quelle fournissait n'était pas standard, limitée et incompatible avec les mises en page non-HTML et les vues de données. Ces limitations signifiaient qu'une réécriture complète était nécessaire. Edge Side Includes est devenu un moyen standard d'implémenter les fonctionnalités que CacheHelper fournissait. Cependant, implémenter Edge Side Includes [34] en PHP présente un grand nombre de limitations. Au lieu de construire une solution de qualité inférieure, nous recommandons aux développeurs ayant besoin d'un cache global d'utiliser Varnish [35] ou Squid [36] à la place.

I18n

Le sous-système I18n a été complètement réécrit. En général, vous pouvez vous attendre au même comportement que dans les versions précédentes, spécialement si vous utilisez la famille de fonctions `__()`.

En interne, la classe `I18n` utilise `Aura\Intl`, et les méthodes appropriées sont exposées pour accéder aux fonctionnalités spécifiques de cette librairie. Pour cette raison, la plupart des méthodes dans `I18n` a été retirée ou renommée.

Grâce à l'utilisation de `ext/intl`, la classe L10n a été complètement retirée. Elle fournissait des données dépassées et incomplètes en comparaison avec les données disponibles dans la classe `Locale` de PHP.

34. http://fr.wikipedia.org/wiki/Edge_Side_Includes
35. http://varnish-cache.org
36. http://squid-cache.org

La langue de l'application par défaut ne sera plus changée automatiquement par la langue du navigateur ou en ayant la valeur `Config.language` définie dans la session du navigateur. Vous pouvez cependant utiliser un filtre du dispatcher pour récupérer automatiquement la langue depuis l'en-tête `Accept-Language` envoyé dans par le navigateur :

```
// Dans config/bootstrap.php
DispatcherFactory::addFilter('LocaleSelector');
```

Il n'y a pas de remplacement intégré en ce qui concerne la sélection de la langue en définissant une valeur dans la session de l'utilisateur.

La fonction de formatage par défaut pour les messages traduits n'est plus `sprintf`, mais la classe `MessageFormatter` la plus avancée et aux fonctionnalités riches. En général, vous pouvez réécrire les placeholders dans les messages comme suit :

```
// Avant:
__('Today is a %s day in %s', 'Sunny', 'Spain');

// Après:
__('Today is a {0} day in {1}', 'Sunny', 'Spain');
```

Vous pouvez éviter la réécriture de vos messages en utilisant l'ancien formateur `sprintf` :

```
I18n::defaultFormatter('sprintf');
```

De plus, la valeur `Config.language` a été retirée et elle ne peut plus être utilisée pour contrôler la langue courante de l'application. A la place, vous pouvez utiliser la classe `I18n` :

```
// Avant
Configure::write('Config.language', 'fr_FR');

// Maintenant
I18n::setLocale('en_US');
```

- Les méthodes ci-dessous ont été déplacées :
  - De `Cake\I18n\Multibyte::utf8()` vers `Cake\Utility\Text::utf8()`
  - De `Cake\I18n\Multibyte::ascii()` vers `Cake\Utility\Text::ascii()`
  - De `Cake\I18n\Multibyte::checkMultibyte()` vers `Cake\Utility\Text::isMultibyte()`
- Puisque l'extension mbstring est maintenant nécessaire, la classe `Multibyte` a été retirée.
- Les messages d'Error dans CakePHP ne passent plus à travers les fonctions de I18n. Ceci a été fait pour simplifier les entrailles de CakePHP et réduire la charge. Les messages auxquels font face les développeurs sont rarement, voire jamais traduits donc la charge supplémentaire n'apporte que peu de bénéfices.

L10n

- Le constructeur de `Cake\I18n\L10n` prend maintenant une instance de `Cake\Network\Request` en argument.

Testing

- `TestShell` a été retiré. CakePHP, le squelette d'application et les plugins nouvellement créés utilisent tous `phpunit` pour exécuter les tests.
- L'exécuteur via le navigateur (webroot/test.php) a été retiré. L'adoption de CLI a beaucoup augmenté depuis les premières versions de 2.x. De plus, les exécuteurs CLI ont une meilleure intégration avec les outils des IDE et autres outils automatisés.
  Si vous cherchez un moyen de lancer les tests à partir d'un navigateur, vous devriez allez voir VisualPHPUnit [37]. Il dispose de plusieurs fonctionnalités supplémentaires par rapport au vieil exécuteur via le navigateur.
- `ControllerTestCase` est dépréciée et sera supprimée de CAKEPHP 3.0.0. Vous devez utiliser les nouvelles fonctionnalités de *Test d'Intégrations des Controllers* à la place.
- Les fixtures doivent maintenant être référencées sous leur forme plurielle :

```
// Au lieu de
$fixtures = ['app.article'];

// Vous devrez utiliser
$fixtures = ['app.articles'];
```

Utility

Classe Set Retirée

La classe Set a été retirée, vous devriez maintenant utiliser la classe Hash à la place.

Folder & File

Les classes folder et file ont été renommées :
- `Cake\Utility\File` renommée *`Cake\Filesystem\File`*
- `Cake\Utility\Folder` renommée *`Cake\Filesystem\Folder`*

Inflector

- la valeur par défaut pour l'argument `$replacement` de la méthode *`Cake\Utility\Inflector::slug()`* a été modifiée de underscore (_) au tiret (-). utiliser des tirets pour séparer les mots dans les url est le choix le plus courant et également celui recommandé par Google.
- Les translitérations pour *`Cake\Utility\Inflector::slug()`* ont changé. Si vous utilisez des translitérations personnalisées, vous devrez mettre à jour votre code. A la place des expressions régulières, les translitérations utilisent le remplacement par chaîne simple. Cela a donné des améliorations de performances significatives :

```
// Au lieu de
Inflector::rules('transliteration', [
    '/ä|æ/' => 'ae',
    '/å/' => 'aa'
]);

// Vous devrez utiliser
```

(suite sur la page suivante)

37. https://github.com/NSinopoli/VisualPHPUnit

(suite de la page précédente)

```
Inflector::rules('transliteration', [
    'ä' => 'ae',
    'æ' => 'ae',
    'å' => 'aa'
]);
```

— Des ensembles de règles non inflectées et irrégulières séparés pour la pluralization et la singularization ont été retirés. Plutôt que d'avoir une liste commune pour chacun. Quand on utilise *Cake\Utility\Inflector::rules()* avec un type "singular" et "plural" vous ne pouvez plus utiliser les clés comme "uninflected", "irregular" dans le tableau d'argument `$rules`.
Vous pouvez ajouter / écraser la liste de règles non inflectées et irrégulières en utilisant *Cake\Utility\Inflector::rules()* en utilisant les valeurs "non inflectées" et "irrégulières" pour un argument `$type`.

Sanitize

— La classe `Sanitize` a été retirée.

Security

— `Security::cipher()` a été retirée. Elle est peu sûre et favorise de mauvaises pratiques en cryptographie. Vous devrez utiliser `Security::encrypt()` à la place.
— La valeur de configuration `Security.cipherSeed` n'est plus nécessaire. Avec le retrait de `Security::cipher()` elle n'est plus utilisée.
— La rétrocompatibilité de `Cake\Utility\Security::rijndael()` pour les valeurs cryptées avant CakePHP 2.3.1 a été retirée. Vous devrez rechiffrer les valeurs en utilisant `Security::encrypt()` et une version plus récente de CakePHP 2.x avant migration.
— La capacité de générer blowfish a été retirée. Vous ne pouvez plus utiliser le type « blowfish » pour `Security::hash()`. Vous devrez utiliser uniquement le *password_hash()* de PHP et *password_verify()* pour générer et vérifier les hashs de blowfish. La librairie compatible ircmaxell/password-compat [38] qui est installée avec CakePHP fournit ces fonctions pour PHP < 5.5.
— OpenSSL est maintenant utilisé à la place de mcrypt pour le chiffrement/déchiffrement des données. Ce changement fournit de meilleurs performances et une avancée dans la supression du support de mcrypt.
— `Security::rijndael()` est dépréciée et elle est seulement disponible quand vous utilisez mcrypt.

Avertissement : Les données chiffrées avec `Security::encrypt()` dans les versions précédentes sont compatibles avec l'implémentation de openssl. Vous devez *définir l'implémentation pour mcrypt* lors de la mise à jour.

Time

— `CakeTime` a été renommée en *Cake\I18n\Time*.
— `Time::__set()` et - `Time::__get()` ont été retirées. Celles-ci étaient des méthodes magiques setter/getter pour une rétrocompatibilité.
— `CakeTime::serverOffset()` a été retirée. Il incitait à des pratiques de correspondance de time incorrects.
— `CakeTime::niceShort()` a été retirée.
— `CakeTime::convert()` a été retirée.
— `CakeTime::convertSpecifiers()` a été retirée.

38. https://packagist.org/packages/ircmaxell/password-compat

— `CakeTime::dayAsSql()` a été retirée.
— `CakeTime::daysAsSql()` a été retirée.
— `CakeTime::fromString()` a été retirée.
— `CakeTime::gmt()` a été retirée.
— `CakeTime::toATOM()` a été renommée en `toAtomString`.
— `CakeTime::toRSS()` a été renommée en `toRssString`.
— `CakeTime::toUnix()` a été renommée en `toUnixString`.
— `CakeTime::wasYesterday()` a été renommée en `isYesterday` pour correspondre aux autres noms de méthode.
— `CakeTime::format()` N'utilise plus les chaînes de format `sprintf`, vous pouvez utiliser ``i18nFormat` à la place.
— `Time::timeAgoInWords()` a maintenant besoin que `$options` soit un tableau.

Time n'est plus une collection de méthodes statiques, il étend `DateTime` pour hériter de ses méthodes et ajoute la localisation des fonctions de formatage avec l'aide de l'extension `intl`.

En général, les expressions ressemblent à ceci :

```
CakeTime::aMethod($date);
```

Peut être migré en réécrivant ceci en :

```
(new Time($date))->aMethod();
```

Number

Number a été réécrite pour utiliser en interne la classe `NumberFormatter`.
— `CakeNumber` a été renommée en *`Cake\I18n\Number`*.
— `Number::format()` nécessite maintenant que `$options` soit un tableau.
— `Number::addFormat()` a été retirée.
— `Number::fromReadableSize()` a été déplacée vers `Cake\Utility\Text:parseFileSize()`.

Validation

— Le range pour `Validation::range()` maintenant inclusif si `$lower` et `$upper` sont fournies.
— `Validation::ssn()` a été retirée.

Xml

— `Xml::build()` a maintenant besoin que `$options` soit un tableau.
— `Xml::build()` n'accepte plus d'URL. Si vous avez besoin de créer un document XML à partir d'une URL, utilisez *Http\Client*.

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 [39] ainsi que la Boulangerie [40] (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 5.6 installé (en CLI). Votre version serveur de PHP doit au moins être aussi 5.6 et, dans l'idéal, devrait également être la même que pour votre version en ligne de commande (CLI).

39. https://plugins.cakephp.org/
40. 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, il suffit simplement de lancer la commande suivante :

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

Ou vous pouvez télécharger `composer.phar` depuis le site de Composer[41].

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:^3.8 cms
```

Si vous avez téléchargé et utilisé l'Installer de Composer pour Windows[42], tapez la commande suivante dans votre terminal depuis le dossier d'installation (par exemple C :\wamp\www\dev\cakephp3) :

```
composer self-update && composer create-project --prefer-dist cakephp/app:^3.8 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
  /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 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.

41. https://getcomposer.org/download/
42. https://getcomposer.org/Composer-Setup.exe

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`). Exécutez ensuite la requête suivante pour créer les premières tables nécessaires au tutoriel :

```
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)
) CHARSET=utf8mb4;
```

(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_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());
```

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. Remplacer 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' => 'Cake\Database\Connection',
            'driver' => 'Cake\Database\Driver\Mysql',
            'persistent' => false,
            'host' => 'localhost',
            'username' => 'cakephp',
            'password' => 'AngelF00dC4k3~',
            'database' => 'cake_cms',
            '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 : Une copie du fichier de configuration par défaut peut être trouvée dans **config/app.default.php**.

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)
    {
        $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 notre 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
{
    protected $_accessible = [
        '*' => true,
        'id' => false,
        'slug' => false,
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ];
}
```

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 maintenant 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 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 **src/Template** et dans un dossier au nom du controller auquel ils sont attachés. Nous devons donc créer un dossier nommé "Articles" dans notre cas. Ajouter le code suivant dans ce fichier :

```
<!-- Fichier : src/Template/Articles/index.ctp -->

<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 rendent les tâches comme créer des liens, des formulaires et des éléments de paginations très faciles. 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 corrigez 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 utilisez 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 `$slug`, mais d'où vient-il ? Si un utilisateur requête `/articles/view/first-post`, alors la valeur "first-post" sera passé à `$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.

Création du template View

Créons le template de view pour notre action « view » dans **src/Template/Articles/view.ctp**.

```
<!-- Fichier : src/Template/Articles/view.ctp -->

<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 :

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

namespace App\Controller;

use App\Controller\AppController;

class ArticlesController extends AppController
{

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

            // L'écriture de 'user_id' en dur est temporaire et
            // sera supprimé 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 *FlashComponent* 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 magiquesde PHP[43]. 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 de 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[44] 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 » :

```
<!-- Fichier : src/Template/Articles/add.ctp -->

<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 l'ouverture du form 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

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

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 **src/Template/Articles/index.ctp** 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 :

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

// Ajoutez ce "use" juste sous la déclaration du namespace
// pour importer la classe Text
use Cake\Utility\Text;

// 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. 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.'));
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    $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 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 : src/Template/Articles/edit.ctp -->

<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.

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

```
<!-- Fichier : src/Template/Articles/index.ctp (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>
```

(suite sur la page suivante)

(suite de la page précédente)

```
            <?= $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
        ->notEmpty('title')
        ->minLength('title', 10)
        ->maxLength('title', 255)

        ->notEmpty('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 taille.

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

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 : src/Template/Articles/index.ctp (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) ?>
```

(suite sur la page suivante)

(suite de la page précédente)

```
        </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 `View\Helper\FormHelper::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

Il serait utile et pratique 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 :

```
# Génère tout le code d'un coup.
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)
{
    $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;

use App\Controller\AppController;

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

            // Hardcoding the user_id is temporary, and will be removed later
            // when we build authentication out.
            $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.'));
        }
```

(suite sur la page suivante)

(suite de la page précédente)

```
        // 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 **src/Template/Articles/add.ctp** :

```
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 ressemble à ceci :

```
public function edit($slug)
{
    $article = $this->Articles
        ->findBySlug($slug)
        ->contain('Tags') // charge les Tags associés
        ->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 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.ctp** au template **src/Template/Articles/edit.ctp**.

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** devra ressembler à :

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

Router::defaultRouteClass(DashedRoute::class);

// 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
Router::scope(
    '/articles',
    ['controller' => 'Articles'],
    function ($routes) {
        $routes->connect('/tagged/*', ['action' => 'tags']);
    }
);

Router::scope('/', function ($routes) {
    // Connect the default home and /pages/* routes.
    $routes->connect('/', [
        'controller' => 'Pages',
        'action' => 'display', 'home'
    ]);
    $routes->connect('/pages/*', [
        'controller' => 'Pages',
        'action' => 'display'
    ]);

    // Connect the conventions based default routes.
    $routes->fallbacks();
});

Plugin::routes();
```

Le code ci-dessus définit une nouvelle "route" qui permet de connecter le chemin URL **/articles/tagged/** à `ArticlesController::tags()`. En définissant une nouvelle route, 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 :

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

public function tags()
{
    // La clé 'pass' est fournie par CakePHP et contient tous les
```

(suite sur la page suivante)

(suite de la page précédente)

```
    // 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
    ]);

    // Passage des variable 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
    ]);

    // 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;

// 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',
    ];
```

(suite sur la page suivante)

(suite de la page précédente)

```
    $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. Créez le fichier **src/Template/Articles/tags.ctp** et ajoutez le contenu suivant :

```
<h1>
    Articles avec les tags
    <?= $this->Text->toList(h($tags), 'ou') ?>
</h1>

<section>
<?php foreach ($articles as $article): ?>
    <article>
        <!-- Utilisation du HtmlHelper pour créer le lien -->
        <h4><?= $this->Html->link(
            $article->title,
            ['controller' => 'Articles', 'action' => 'view', $article->slug]
        ) ?></h4>
        <span><?= h($article->created) ?>
    </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.ctp** 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 en-

voyées 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;

protected function _getTagString()
{
    if (isset($this->_properties['tag_string'])) {
        return $this->_properties['tag_string'];
    }
    if (empty($this->tags)) {
        return '';
    }
    $tags = new Collection($this->tags);
    $str = $tags->reduce(function ($string, $tag) {
        return $string . $tag->title . ', ';
    }, '');
    return trim($str, ', ');
}
```

Cela nous permettra d'accéder à la propriété virtuelle `$article->tag_string`. Nous utiliserons cette propriété plus tard.

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 **src/Template/Articles/add.ctp** et **src/Template/Articles/edit.ctp**, remplacez l'élément de contrôle existant `tags._ids` avec la déclaration suivante :

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

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
}

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]);

    // 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;
}
```

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.

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 et appliquer une gestion basique de contrôle d'accès à la création et à la modification d'articles.

Ajout du Hash du Mot de Passe

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.

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 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 définie une valeur dans une entity. Ajoutons un setter pour le mot de passe. Dans **src/Model/Entity/User.php**, ajoutez le code suivant :

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

use Cake\Auth\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($value)
    {
        if (strlen($value)) {
            $hasher = new DefaultPasswordHasher();

            return $hasher->hash($value);
        }
    }
}
```

Maintenant, rendez-vous sur **http ://localhost :8765/users** pour voir une liste des utilisateurs existants. 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 hashé du mot de passe à la place de la valeur par défaut sur l'action index ou view. CakePHP hash les mots de passe, par défaut, avec bcrypt [45]. Vous pouvez aussi utiliser SHA-1 ou MD5 si vous travaillez sur une base de données déjà existante mais nous vous recommandons d'utiliser bcrypt pour toutes vos nouvelles applications.

45. http://codahale.com/how-to-safely-store-a-password/

Ajouter la Connexion

Dans CakePHP, l'authentification est gérée via *Components (Composants)*. Les Components peuvent être considérés comme un moyen de créer des morceaux de code réutilisables pour les controllers en leur donnant un concept ou une fonctionnalité spécifique. Les Components peuvent se greffer aux événements du cycle de vie des événements des controllers et intéragir avec l'application de cette manière. Pour commencer, nous allons ajouter *AuthComponent* à notre application. Puisque nous voulons que les méthodes create, update et delete requièrent l'authentification, 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()
    {
        // Code existant

        $this->loadComponent('Auth', [
            'authenticate' => [
                'Form' => [
                    'fields' => [
                        'username' => 'email',
                        'password' => 'password'
                    ]
                ]
            ],
            'loginAction' => [
                'controller' => 'Users',
                'action' => 'login'
            ],
             // Si pas autorisé, on renvoit sur la page précédente
            'unauthorizedRedirect' => $this->referer()
        ]);

        // Permet à l'action "display" de notre PagesController de continuer
        // à fonctionner. Autorise également les actions "read-only".
        $this->Auth->allow(['display', 'view', 'index']);
    }
}
```

De cette manière, nous disons à CakePHP de charger les Components `Flash` et `Auth`. De plus, nous avons personnaliser la configuration de AuthComponent car notre tables d'utilisateurs (users), utilise le champ `email` comme identifiant. À partir de maintenant, si vous vous rendez sur n'importe laquelle des URLs protégées, comme `/articles/add`, vous allez être redirigé sur **/users/login**, ce qui devrait vous afficher une page d'erreur puisque nous n'avons pas encore écrit le code qui gère cette page. Créons maintenant l'action login :

```
// Dans src/Controller/UsersController.php
public function login()
{
    if ($this->request->is('post')) {
        $user = $this->Auth->identify();
        if ($user) {
            $this->Auth->setUser($user);
            return $this->redirect($this->Auth->redirectUrl());
```

(suite sur la page suivante)

(suite de la page précédente)

```
        }
        $this->Flash->error('Votre identifiant ou votre mot de passe est incorrect.');
    }
}
```

Et dans **src/Template/Users/login.ctp**, ajoutez :

```
<h1>Login</h1>
<?= $this->Form->create() ?>
<?= $this->Form->control('email') ?>
<?= $this->Form->control('password') ?>
<?= $this->Form->button('Connexion') ?>
<?= $this->Form->end() ?>
```

Note : La méthode `control()` est disponible depuis 3.4. Pour les versions précédentes, utilisez la méthode `input()` à la place.

Maintenant que nous avons un formulaire de connexion basique, nous devrions être capable de nous connecter avec un utilisateur qui a un mot de passe hashé.

Note : Si aucun de vos utilisateurs a un mot de passe hashé, commentez le bloc `loadComponent('Auth')` et les appels à `$this->Auth->allow()`. Puis allez éditer un utilisateur pour lui sauvegarder un nouveau mot de passe. Après avoir sauvegardé le mot de passe pour l'utilisateur, décommentez les lignes que vous venez tout juste de commenter.

Avant de vous connectez, visitez `/articles/add`. Puisque l'action n'est pas autorisée, vous serez redirigé sur la page de connexion. Après vous être connecté CakePHP vous redirigera automatiquement sur `/articles/add`.

Ajout de la Déconnexion

Maintenant que vos utilisateurs peuvent se connecter, il faut leur donner la possibilité de se déconnecter. Ajoutez le code suivant dans le `UsersController` :

```
public function initialize()
{
    parent::initialize();
    $this->Auth->allow(['logout']);
}

public function logout()
{
    $this->Flash->success('Vous avez été déconnecté.');
    return $this->redirect($this->Auth->logout());
}
```

Ce code ajoute l'action `logout` à la liste des actions qui ne nécessitent pas d'être authentifié et implémente la logique de déconnexion. Vous pouvez vous rendre à l'adresse `/users/logout` pour vous déconnecter. Vous serez ensuite redirigé sur 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` :

```
public function initialize()
{
    parent::initialize();
    // Ajoute l'action 'add' à la liste des actions autorisées.
    $this->Auth->allow(['logout', 'add']);
}
```

Le code ci-dessus indique à `AuthComponent` que la méthode `add()` du `UsersController` peut être visitée sans être authentifié ou avoir besoin d'autorisation. Pour avoir une page de création plus propre, nous vous invitons à retirer les liens et autres contenus qui n'ont plus de sens pour cette page de création de compte. De même, nous ne nous occuperons pas des autres actions spécifiques aux utilisateurs, mais c'est quelque chose que vous pouvez faire vous même comme exercice.

Restreindre l'Accès aux Articles

Maintenant que nos utilisateurs peuvent se connecter, nous souhaitons limiter l'édition seulement aux articles qu'ils ont rédigé. Nous allons implémenter cette fonctionnalité à l'aide d'un adapter "authorization". Puisque nos besoins sont assez limités, nous pouvons rediger cette logique dans le `ArticlesController`. Mais avant, nous devons indiquer à `AuthComponent` comment notre application va gérer l'accès à nos actions. Mettez à jour votre `AppController` avec ceci :

```
public function isAuthorized($user)
{
    // Par défaut, on refuse l'accès.
    return false;
}
```

Ensuite, nous allons indiquer à `AuthComponent` que nous voulons utiliser les méthodes de hooks des controllers pour gérer *l'authorization*. Votre méthode `AppController::initialize()` devrait maintenant ressembler à ceci :

```
public function initialize()
{
    // Code existant code

    $this->loadComponent('Flash');
    $this->loadComponent('Auth', [
        // La ligne suivante a été ajoutée
        'authorize'=> 'Controller',
        'authenticate' => [
            'Form' => [
                'fields' => [
                    'username' => 'email',
                    'password' => 'password'
                ]
            ]
        ],
        'loginAction' => [
            'controller' => 'Users',
            'action' => 'login'
        ],
```

(suite sur la page suivante)

(suite de la page précédente)

```
        // Si pas autorisé, on renvoit sur la page précédente
        'unauthorizedRedirect' => $this->referer()
    ]);

    // Permet à l'action "display" de notre PagesController de continuer
    // à fonctionner. Autorise également les actions "read-only".
    $this->Auth->allow(['display', 'view', 'index']);
}
```

Par défaut, nous empêchons l'accès et nous allons donner accès au fur et à mesure, en fonction des cas. Pour commencer, nous allons ajouter la logique d'autorisation pour les articles. Dans votre `ArticlesController`, ajoutez le code suivant :

```
public function isAuthorized($user)
{
    $action = $this->request->getParam('action');
    // Les actions 'add' et 'tags' sont toujours autorisés pour les utilisateur
    // authentifiés sur l'application
    if (in_array($action, ['add', 'tags'])) {
        return true;
    }

    // Toutes les autres actions nécessitent un slug
    $slug = $this->request->getParam('pass.0');
    if (!$slug) {
        return false;
    }

    // On vérifie que l'article appartient à l'utilisateur connecté
    $article = $this->Articles->findBySlug($slug)->first();

    return $article->user_id === $user['id'];
}
```

Maintenant, si vous essayez de modifier ou supprimer un article qui ne vous appartient pas, vous serez redirigé sur la page où vous étiez avant. Si aucun message d'erreur n'apparaît, ajoutez ceci à votre layout :

```
// Dans src/Template/Layout/default.ctp
<?= $this->Flash->render() ?>
```

Bien que le code ci-dessus soit très simple, cela démontre comment vous pouvez facilement construire des logiques d'autorisation flexibles qui impliquent l'utilisateur connecté et / ou les données de la requête.

Renforcer les Action Add & Edit

Bien que nous ayons bloqué l'accès de l'action edit, nous sommes toujours vulnérables aux utilisateurs qui changeraient l'attribut `user_id` des articles pendant la modification. Mais nous allons commencer par nous occuper de l'action `add` en premier.

Lorsque vous créez des articles, on veut forcer le `user_id` à celui de l'utilisateur actuellement connecté. Remplacer le code de votre action `add` par le code suivant :

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

public function add()
```

(suite sur la page suivante)

(suite de la page précédente)

```
{
    $article = $this->Articles->newEntity();
    if ($this->request->is('post')) {
        $article = $this->Articles->patchEntity($article, $this->request->getData());

        // Changé : On force le user_id à celui de la session
        $article->user_id = $this->Auth->user('id');

        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);
}
```

Ensuite, nous allons nous occuper de l'action `edit`. Remplacez le code de l'action par ceci :

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

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

    if ($this->request->is(['post', 'put'])) {
        $this->Articles->patchEntity($article, $this->request->getData(), [
            // Ajouté : Empêche la modification du user_id.
            'accessibleFields' => ['user_id' => false]
        ]);
        if ($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 l\'article.'));
    }
    $this->set('article', $article);
}
```

Ici, nous avons modifier les propriétés qui peuvent être assignées en masse via les options de `patchEntity()`. Référez-vous à la section *Changer les Champs Accessibles* pour plus de détails. Pensez également à retirer l'élément de contrôle de `user_id` sur **src/Templates/Articles/edit.ctp**.

Conclusion

Nous avons créer une application CMS simple qui permet à nos utilisateurs de se connecter, de poster des articles, leur ajouter des tags, récupérer les articles par leurs tags et nous avons fini par ajouter une couche de contrôle d'accès à nos articles. Nous avons également ajouté des améliorations UX en tirant avantage du FormHelper et de l'ORM.

Merci d'avoir pris le temps d'explorer CakePHP. Pour les prochaines étapes de votre apprentissage, nous vous conseillons la documentations de *l'ORM* ou bien de vous diriger vers la section 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 5.6 (CLI) de PHP ou supérieure. La version de PHP de votre serveur web doit aussi être 5.6 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 [46]. C'est parti !

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, c'est aussi facile que de lancer ce qui suit :

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

Ou alors vous pouvez télécharger `composer.phar` depuis le site de Composer [47].

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:^3.8 bookmarker
```

Si vous avez téléchargé et exécuté l'"installeur Windows de Composer [48], 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:^3.8␣
↪bookmarker
```

46. https://github.com/cakephp/bookmarker-tutorial
47. https://getcomposer.org/download/
48. https://getcomposer.org/Composer-Setup.exe

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*.

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(255),
    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),
    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 [49] par défaut. Vous pouvez aussi utiliser sha1 ou md5 si vous travaillez avec une base de 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 :

49. http://codahale.com/how-to-safely-store-a-password/

```
<?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 **src/Template/Bookmarks/tags.ctp** 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>
```

(suite sur la page suivante)

(suite de la page précédente)

```
        <!-- 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.ctp** 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()
    {
        $this->loadComponent('Flash');
        $this->loadComponent('Auth', [
```

(suite sur la page suivante)

(suite de la page précédente)

```
            '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) {
            $this->Auth->setUser($user);
            return $this->redirect($this->Auth->redirectUrl());
        }
        $this->Flash->error('Votre username ou mot de passe est incorrect.');
    }
}
```

Et dans **src/Template/Users/login.ctp**, 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() ?>
```

Note : La méthode `control()` est disponible depuis 3.4. Si vous utilisez une version précédente, utilisez la méthode `input()`.

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()
{
    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()
{
    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.ctp** 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()
{
    $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
    // 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')) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        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 src/Template/Layout/default.ctp
<?= $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 **src/Template/Bookmarks/add.ctp**. 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')) {
        $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->set('_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->set('_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'),
        ]
    ];
    $this->set('bookmarks', $this->paginate($this->Bookmarks));
    $this->set('_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->_properties['tag_string'])) {
        return $this->_properties['tag_string'];
    }
    if (empty($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 $_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 **src/Template/Bookmarks/add.ctp** et **src/Template/Bookmarks/edit.ctp**, 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 5.6 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'invité de ligne de commande. Tapez simplement les deux lignes suivantes dans votre terminal à partir de votre répertoire webroot :

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

Ou vous pouvez télécharger `composer.phar` du site de Composer[50].

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 que vous souhaitez 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:^3.8 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:^3.8 blog
```

L'avantage d'utiliser Composer est qu'il va automatiquement réaliser certaines tâches de configurations importantes, comme configurer les bonnes permissions de fichier et créer votre fichier config/app.php à votre place.

Il y a d'autres moyens d'installer CakePHP. Si vous ne pouvez ou ne voulez pas utiliser `Composer`, regardez la section *Installation*.

50. https://getcomposer.org/download/

Peu importe la façon dont vous l'avez téléchargé, placez le code à l'intérieur du « DocumentRoot » de votre serveur. Une fois terminé, 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 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 <folder> ». Si vous avez plutôt un message d'erreur ou voulez le faire manuellement, la meilleur façon de le faire est de trouver sous quel utilisateur votre serveur web tourne en faisant (`<?= `whoami`; ?>`) et en changeant le possesseur du répertoire **src/tmp** pour 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 serez informé par un avertissement quand vous n'êtes pas en mode production.

Bien que non recommandé, si vous ne pouvez pas configurer les permissions de la même façon que pour 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 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 ex `cake_blog`.Pour le moment, nous allons juste créer une simple table pour stocker nos posts. Nous allons également insérer quelques posts à des fins de tests. Exécutez les requêtes SQL suivantes dans votre base de données :

```
# D'abord, créons la table des posts
CREATE TABLE articles (
```

(suite sur la page suivante)

(suite de la page précédente)

```
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(50),
    body TEXT,
    created DATETIME DEFAULT NULL,
    modified DATETIME DEFAULT NULL
);

# Puis insérons quelques posts pour les tests
INSERT INTO articles (title,body,created)
    VALUES ('The title', 'This is the article body.', NOW());
INSERT INTO articles (title,body,created)
    VALUES ('A title once again', 'And the article body follows.', NOW());
INSERT INTO articles (title,body,created)
    VALUES ('Title strikes back', 'This is really exciting! Not.', NOW());
```

Les choix des noms pour les tables et les colonnes ne sont pas arbitraires. 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" permet de les avoir 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' => [
            '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. 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' => 'something long and containing lots of different values.',
],
```

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 que votre serveur web fonctionne :

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)
    {
        $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 la possibilité de générer du 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 toute interaction avec les articles va se faire. 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;

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.

La seule instruction que cette action utilise est `set()`, pour transmettre les données du controller à la vue (que nous créerons à la prochaine étape). La ligne définit la variable de vue appelée "articles" qui est égale à la valeur de retour de la méthode `find('all')` de l'objet table 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 la possibilité de générer du code pour créer le controller 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 et les flux définis par notre controller, nous allons créer une vue pour l'action « index » que nous avons créé ci-dessus.

Les vues de CakePHP sont juste des fragments de présentation « assaisonnée », qui s'intègrent au sein d'un layout applicatif. 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, encapsulé autour d'une vue. Ils peuvent être définis et interchangés, 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 devrait transmettre l'objet query à la vue pour être invoqué par une itération `foreach`.

Les fichiers de template de CakePHP sont stockés dans **src/Template** 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 :

```
<!-- File: src/Template/Articles/index.ctp -->

<h1>Tous les articles du Blog</h1>
<table>
    <tr>
        <th>Id</th>
        <th>Title</th>
        <th>Created</th>
    </tr>

    <!-- Ici se trouve l'itération sur l'objet query de nos $articles, l'affichage␣
↪des infos des articles -->
```

(suite sur la page suivante)

(suite de la page précédente)

```
    <?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 de « helpers » (des assistants) 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()` génèrera 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. Ceci est expliqué dans le chapitre des Routes. Utiliser les tableaux dans les URLs vous permet de tirer profit des capacités de CakePHP à ré-inverser les routes. Vous pouvez aussi utiliser les URLs relatives depuis la base de l'application sous la forme `/controller/action/param1/param2` ou en utilisant les *routes nommées*.

A 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 (le lien sur le titre d'un article mène à l'URL `/articles/view/un_id_quelconque`), vous avez sûrement été informé par CakePHP que l'action n'a pas encore été définie. Si vous n'avez pas été 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;

use App\Controller\AppController;

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'));
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
}
```

L'appel de `set()` devrait vous être familier. Notez que nous utilisons `get()` plutôt que `find('all')` parce que nous voulons seulement récupérer les informations d'un seul article.

Notez que notre action « view » prend un paramètre : l'ID de l'article que nous aimerions voir. Ce paramètre est transmis à l'action grâce à l'URL demandée. 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. Si un utilisateur requête `/articles/view`, nous lancerons un `NotFoundException` et laisserons le Gestionnaire d'Erreur de CakePHP ErrorHandler prendre le dessus. En utilisant la fonction `get()` dans la table Articles, nous faisons aussi une vérification similaire pour nous assurer que l'utilisateur a accès à l'enregistrement qui existe. Dans le cas où l'article requêté n'est pas présent dans la base de données, la fonction `get()` va lancer une `NotFoundException`.

Maintenant, créons la vue pour notre nouvelle action "view" et plaçons-la dans **src/Template/Articles/view.ctp**.

```
<!-- File: src/Template/Articles/view.ctp -->

<h1><?= h($article->title) ?></h1>
<p><?= h($article->body) ?></p>
<p><small>Created: <?= $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}`.

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` :

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

namespace App\Controller;

use App\Controller\AppController;

class ArticlesController extends AppController
{
    public function initialize()
    {
        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);
```

(suite sur la page suivante)

(suite de la page précédente)

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

    public function add()
    {
        $article = $this->Articles->newEntity();
        if ($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 d'inclure le component *FlashComponent* dans chaque controller où vous voulez les utiliser. Si nécessaire, incluez-les dans le controller principal (`AppController`).

Voici ce que fait l'action `add()` : si la requête HTTP est de type POST, essayez de sauvegarder les données en utilisant le model « Articles ». Si pour une raison quelconque, la sauvegarde a échouée, affichez simplement la vue. Cela nous donne une chance de voir les erreurs de validation de l'utilisateur et 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 les 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()` pour définir un message dans une variable de session. Ces méthodes sont fournies via la méthode magique _call() [51] 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 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 [52] 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 erreurs de validation et interrompra l'enregistrement si une erreur survient. Nous verrons la façon dont les erreurs sont traitées dans les sections suivantes.

51. http://php.net/manual/fr/language.oop5.overloading.php#object.call
52. https://api.cakephp.org

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. *Cake\View\Helper\FormHelper* est disponible par défaut dans toutes les vues avec la variables `$this->Form`.

Voici le code de notre vue « add » (ajout) :

```
<!-- File: src/Template/Articles/add.ctp -->

<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 quels champs ils correspondent et le second paramètre vous permet de spécifier un large éventail d'options - dans ce 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()` cloture le formulaire. Affiche les champs cachés si la protection de falsification de formulaire et/ou CRSF est activée.

A présent, revenons en arrière et modifions notre vue **src/Template/Articles/index.ctp** 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)
    {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        $this->addBehavior('Timestamp');
    }

    public function validationDefault(Validator $validator)
    {
        $validator
            ->notEmpty('title')
            ->requirePresence('title')
            ->notEmpty('body')
            ->requirePresence('body');

        return $validator;
    }
}
```

Le méthode `validationDefault()` indique à CakePHP comment valider vos données lorsque la méthode `save()` est appelée. Ici, j'ai 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 vide afin de voir comment 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.

Editer des Articles

L'édition de 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

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 back ou kick back et montrer les erreurs de validation de l'utilisateur.

La vue d'édition devrait ressembler à quelque chose comme cela :

```
<!-- File: src/Template/Articles/edit.ctp -->

<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 un article ou la mise à jour d'un article existant.

Vous pouvez maintenant mettre à jour votre vue index avec des liens pour éditer des articles :

```
<!-- File: src/Template/Articles/index.ctp  (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>
        <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

A présent, mettons en place un moyen de supprimer les articles pour les utilisateurs. Démarrons avec 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, une exception est levée. 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.

Etant donné que nous exécutons juste un peu de logique et de redirection, cette action n'a pas de vue. Vous voudrez peut-être mettre à jour votre vue index avec des liens pour permettre aux utilisateurs de supprimer des articles, ainsi :

```
<!-- File: src/Template/Articles/index.ctp -->

<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' => 'Etes-vous sûr?'])
            ?>
```

(suite sur la page suivante)

(suite de la page précédente)

```
            <?= $this->Html->link('Modifier', ['action' => 'edit', $article->id]) ?>
        </td>
    </tr>
    <?php endforeach; ?>

</table>
```

Utiliser `View\Helper\FormHelper::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 tous les supprimer.

Note : Ce code de vue utilise aussi le helper `FormHelper` pour demander à l'utilisateur une confirmation 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 facilité d'utilisation 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 rapide modification des routes dans ce tutoriel.

Pour plus d'informations sur les techniques de routages, 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`).

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 :

```
$routes->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 :

```
$routes->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 "reverse routing" (ou routage inversé). Par exemple, pour la route définie plus haut, en ajoutant `['controller' => 'Articles', 'action' => 'index']` à la fonction retournant un tableau, l'URL "/" sera utilisée. Il est d'ailleurs bien avisé de toujours utiliser un tableau pour les URLs afin que vos routes définissent où vont les URLs, mais aussi pour s'assurer qu'elles aillent dans la même direction.

Conclusion

Simple n'est ce pas ? 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 <https ://api.cakephp.org>* 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 les portions de vues.
3. /bake/usage Générer un code CRUD basique.
4. *Tutoriel d'un Blog - Authentification et Autorisations* : Tutoriel sur l'enregistrement et la connexion d'utilisateurs.

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 voulons utiliser le plugin migrations [53] 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');` dans le fichier bootstrap.php 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 :

53. https://github.com/cakephp/migrations

```
<?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,
            '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,
        ]);
```

(suite sur la page suivante)

(suite de la page précédente)

```
        $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,
        ]);
        $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 niveau supérieur 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 categories.

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;
```

(suite sur la page suivante)

(suite de la page précédente)

```
class ArticlesTable extends Table
{
    public function initialize(array $config)
    {
        $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 de bake suivantes :

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

De manière alternative, vous pouvez créer la totalité avec 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 **src/Template/Categories/add.ctp** et **src/Template/Categories/edit.ctp** :

```
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 MPTT logic [54] pour gérer les données. Les structures en arbre MPTT sont optimisées pour lire des données ce qui les rend souvent pratique 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 quelques fonctionnalités comme la réorganisation de l'ordre des categories. Nous verrons cela dans un moment.

54. http://www.sitepoint.com/hierarchical-data-database-2/

Mais pour l'instant, vous devez retirer les lignes suivantes dans vos fichiers de template add et edit :

```
echo $this->Form->control('lft');
echo $this->Form->control('rght');
```

De plus, vous devez désactiver ou retirer les requirePresence du validateur pour `lft` et `rght` dans votre model CategoriesTable :

```
public function validationDefault(Validator $validator)
{
    $validator
        ->add('id', 'valid', ['rule' => 'numeric'])
        ->allowEmpty('id', 'create');

    $validator
        ->add('lft', 'valid', ['rule' => 'numeric'])
    //    ->requirePresence('lft', 'create')
        ->notEmpty('lft');

    $validator
        ->add('rght', 'valid', ['rule' => 'numeric'])
    //    ->requirePresence('rght', 'create')
        ->notEmpty('rght');
}
```

Ces champs sont automatiquement gérés par le TreeBehavior quand une catégorie est sauvegardée.

En utilisant votre navigateur, ajoutez quelques nouvelles catégories en utilisant l'action du controller `/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.

Modifiez la méthode index dans votre **CategoriesController.php** et ajoutez 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']);
        $this->set(compact('categories'));
        $this->set('_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('The category has been moved Up.');
        } else {
            $this->Flash->error('The category could not be moved up. Please, try␣
↪again.');
        }
```

(suite sur la page suivante)

(suite de la page précédente)

```
        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('The category has been moved down.');
        } else {
            $this->Flash->error('The category could not be moved down. Please, try␣
↪again.');
        }
        return $this->redirect($this->referer(['action' => 'index']));
    }
}
```

Remplacez le contenu existant dans **src/Template/Categories/index.ctp** par ceci :

```
<div class="actions large-2 medium-3 columns">
    <h3><?= __('Actions') ?></h3>
    <ul class="side-nav">
        <li><?= $this->Html->link(__('Nouvelle Categorie'), ['action' => 'add']) ?></
↪li>
    </ul>
</div>
<div class="categories index large-10 medium-9 columns">
    <table cellpadding="0" cellspacing="0">
    <thead>
        <tr>
            <th>Id</th>
            <th>Parent Id</th>
            <th>Lft</th>
            <th>Rght</th>
            <th>Name</th>
            <th>Description</th>
            <th>Created</th>
            <th class="actions"><?= __('Actions') ?></th>
        </tr>
    </thead>
    <tbody>
    <?php foreach ($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)]) ?>
```

(suite sur la page suivante)

(suite de la page précédente)

```
                <?= $this->Form->postLink(__('Descendre'), ['action' => 'moveDown',
↪$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)]) ?>
            </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 lorsque l'on va le créer ou le modifier :

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

namespace App\Controller;

// Prior to 3.6 use Cake\Network\Exception\NotFoundException
use Cake\Http\Exception\NotFoundException;

class ArticlesController extends AppController
{

    // ...

    public function add()
    {
        $article = $this->Articles->newEntity();
        if ($this->request->is('post')) {
            $article = $this->Articles->patchEntity($article, $this->request->
↪getData());
            if ($this->Articles->save($article)) {
                $this->Flash->success(__('Your article has been saved.'));
                return $this->redirect(['action' => 'index']);
            }
            $this->Flash->error(__('Unable to add your 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 :

```
<!-- File: src/Template/Articles/add.ctp -->

<h1>Add Article</h1>
<?php
echo $this->Form->create($article);
// Ajout des input (via la méthode "control") liés aux catégories
echo $this->Form->control('category_id');
echo $this->Form->control('title');
echo $this->Form->control('body', ['rows' => '3']);
echo $this->Form->button(__('Save Article'));
echo $this->Form->end();
```

Quand vous allez à l'adresse `/yoursite/categories/add`, vous devriez voir une liste des catégories à choisir.

Tutoriel d'un Blog - Authentification et Autorisations

Suivez notre exemple *Tutoriel d'un Blog*, imaginons que nous souhaitions sécuriser l'accès de certaines URLs, basées sur la connexion de l'utilisateur. Nous avons aussi un autre impératif : permettre à notre blog d'avoir plusieurs auteurs, afin que chacun d'eux puisse créer ses propres articles, les modifier et les supprimer mais ne laisser la possibilité de ne modifier que ses propres messages.

Créer le code lié à tous les utilisateurs

Premièrement, créons une nouvelle table dans notre base de données blog pour enregistrer les données de notre utilisateur :

```
CREATE TABLE users (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50),
    password VARCHAR(255),
    role VARCHAR(20),
    created DATETIME DEFAULT NULL,
    modified DATETIME 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 du nom d'utilisateur et du mot de passe 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;

class UsersTable extends Table
{
```

(suite sur la page suivante)

(suite de la page précédente)

```
    public function validationDefault(Validator $validator)
    {
        return $validator
            ->notEmpty('username', "Un nom d'utilisateur est nécessaire")
            ->notEmpty('password', 'Un mot de passe est nécessaire')
            ->notEmpty('role', 'Un role est nécessaire')
            ->add('role', 'inList', [
                'rule' => ['inList', ['admin', 'author']],
                'message' => 'Merci de rentrer un role valide'
            ]);
    }

}
```

Créons aussi notre UsersController, le contenu suivant correspond à la classe obtenue grâce à l'utilitaire de génération de code fournis par CakePHP :

```
// src/Controller/UsersController.php

namespace App\Controller;

use App\Controller\AppController;
use Cake\Event\Event;

class UsersController extends AppController
{

    public function beforeFilter(Event $event)
    {
        parent::beforeFilter($event);
        $this->Auth->allow('add');
    }

     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 = $this->Users->newEntity();
        if ($this->request->is('post')) {
            // Avant 3.4.0 $this->request->data() etait utilisée.
            $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' => 'index']);
            }
            $this->Flash->error(__("Impossible d'ajouter l'utilisateur."));
```

(suite sur la page suivante)

(suite de la page précédente)

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

}
```

De la même façon, nous avons créé les vues pour nos articles de blog en utilisant l'outil de génération de code. Dans le cadre de ce tutoriel, nous allons juste montrer le add.ctp :

```
<!-- src/Template/Users/add.ctp -->

<div class="users form">
<?= $this->Form->create($user) ?>
    <fieldset>
        <legend><?= __('Ajouter un utilisateur') ?></legend>
        <?= $this->Form->control('username') ?>
        <?= $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 prêt à ajouter notre couche d'authentification. Dans CakePHP, c'est géré par `Cake\Controller\Component\AuthComponent`, une classe responsable d'exiger la connexion pour certaines actions, de gérer la connexion et la déconnexion, et aussi d'autoriser aux utilisateurs connectés les actions que l'on souhaite leur voir autorisées.

Pour ajouter ce component à votre application, ouvrez votre fichier **src/Controller/AppController.php** et ajoutez les lignes suivantes :

```
// src/Controller/AppController.php

namespace App\Controller;

use Cake\Controller\Controller;
use Cake\Event\Event;

class AppController extends Controller
{
    //...

    public function initialize()
    {
        $this->loadComponent('Flash');
        $this->loadComponent('Auth', [
            'loginRedirect' => [
                'controller' => 'Articles',
                'action' => 'index'
            ],
            'logoutRedirect' => [
```

(suite sur la page suivante)

(suite de la page précédente)

```
                'controller' => 'Pages',
                'action' => 'display',
                'home'
            ]
        ]);
    }

    public function beforeFilter(Event $event)
    {
        $this->Auth->allow(['index', 'view', 'display']);
    }
    //...
}
```

Il n'y a pas grand chose à configurer, puisque nous avons utilisé les conventions pour la table users. Nous avons juste configuré les URLs qui seront chargées après que la connexion et la déconnexion des actions sont effectuées, dans notre cas, respectivement à `/articles/` et `/`.

Ce que nous avons fait dans la fonction `beforeFilter()` a été de dire au AuthComponent de ne pas exiger un login pour toutes les actions `index()` et `view()`, dans chaque controller. Nous voulons que nos visiteurs soient capables de lire et lister les entrées sans s'inscrire sur le site.

Maintenant, nous avons besoin d'être capable d'inscrire des nouveaux utilisateurs, de sauvegarder leur nom d'utilisateur et mot de passe, et plus important de hasher leur mot de passe afin qu'il ne soit pas stocké en clair dans notre base de données. Disons à AuthComponent de laisser certains utilisateurs non-authentifiés accéder à la fonction add des utilisateurs et de réaliser l'action connexion et déconnexion :

```
// src/Controller/UsersController.php
namespace App\Controller;

use App\Controller\AppController;
use Cake\Event\Event;

class UsersController extends AppController
{
    // Other methods..

    public function beforeFilter(Event $event)
    {
        parent::beforeFilter($event);
        // Allow users to register and logout.
        // You should not add the "login" action to allow list. Doing so would
        // cause problems with normal functioning of AuthComponent.
        $this->Auth->allow(['add', 'logout']);
    }

    public function login()
    {
        if ($this->request->is('post')) {
            $user = $this->Auth->identify();
            if ($user) {
                $this->Auth->setUser($user);
                return $this->redirect($this->Auth->redirectUrl());
            }
            $this->Flash->error(__('Invalid username or password, try again'));
        }
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    public function logout()
    {
        return $this->redirect($this->Auth->logout());
    }
}
```

Le hash du mot de passe n'est pas encore fait, nous avons besoin d'une classe Entity pour notre User afin de gérer sa propre logique spécifique. Créons fichier entity dans **src/Model/Entity/User.php** et ajoutons ce qui suit :

```
// src/Model/Entity/User.php
namespace App\Model\Entity;

use Cake\Auth\DefaultPasswordHasher;
use Cake\ORM\Entity;

class User extends Entity
{

    // Rend les champs assignables en masse sauf pour le champ clé primaire "id".
    protected $_accessible = [
        '*' => true,
        'id' => false
    ];

    // ...

    protected function _setPassword($password)
    {
        if (strlen($password) > 0) {
            return (new DefaultPasswordHasher)->hash($password);
        }
    }

    // ...
}
```

Ainsi, maintenant à chaque fois qu'un utilisateur est sauvegardé, le mot de passe est hashé en utilisant la classe `DefaultPasswordHasher`. Il nous manque juste un fichier template de vue pour la fonction de connexion. Ouvrez votre fichier **src/Template/Users/login.ctp** et ajoutez les lignes suivantes :

```
<!-- src/Template/Users/login.ctp -->

<div class="users form">
<?= $this->Flash->render() ?>
<?= $this->Form->create() ?>
    <fieldset>
        <legend><?= __("Merci de rentrer vos nom d'utilisateur et mot de passe") ?></
↪legend>
        <?= $this->Form->control('username') ?>
        <?= $this->Form->control('password') ?>
    </fieldset>
<?= $this->Form->button(__('Se Connecter')); ?>
<?= $this->Form->end() ?>
</div>
```

Vous pouvez maintenant inscrire un nouvel utilisateur en rentrant l'URL `/users/add` et vous connecter avec ce pro-

fil nouvellement créé en allant sur l'URL `/users/login`. Essayez aussi d'aller sur n'importe quel URL qui n'a pas été explicitement autorisée telle que `/articles/add`, vous verrez que l'application vous redirige automatiquement vers la page de connexion.

Et c'est tout ! Cela semble trop simple pour être vrai. Retournons en arrière un peu pour expliquer ce qui s'est passé. La fonction `beforeFilter()` dit au component AuthComponent de ne pas exiger de connexion pour l'action `add()` en plus des actions `index()` et `view()` qui étaient déjà autorisées dans la fonction `beforeFilter()` de l'AppController.

L'action `login()` appelle la fonction `$this->Auth->identify()` dans AuthComponent, et cela fonctionne sans autre config car nous suivons les conventions comme mentionnées plus tôt. C'est-à-dire, avoir un model User avec les colonnes username et password, et utiliser un formulaire posté à un controller avec les données d'utilisateur. Cette fonction retourne si la connexion a réussi ou non, et en cas de succès, alors nous redirigeons l'utilisateur vers l'URL de redirection configurée que nous utilisions quand nous avons ajouté AuthComponent à notre application.

La déconnexion fonctionne juste en allant à l'URL `/users/logout` et redirigera l'utilisateur vers l'Url de Déconnexion configurée décrite précédemment. Cette URL est le résultat de la fonction `AuthComponent::logout()` en cas de succès.

Autorisation (Qui est autorisé à accéder à quoi)

Comme mentionné avant, nous convertissons ce blog en un outil multi-utilisateur à autorisation, et pour ce faire, nous avons besoin de modifier un peu la table articles pour ajouter la référence à la table Users :

```
ALTER TABLE articles ADD COLUMN user_id INT(11);
```

Aussi, un petit changement dans ArticlesController est nécessaire pour stocker l'utilisateur connecté courant en référence pour l'article créé :

```
// src/Controller/ArticlesController.php
public function add()
{
    $article = $this->Articles->newEntity();
    if ($this->request->is('post')) {
        // Avant 3.4.0 $this->request->data() etait utilisée.
        $article = $this->Articles->patchEntity($article, $this->request->getData());
        // Ajout de cette ligne
        $article->user_id = $this->Auth->user('id');
        // Vous pourriez aussi faire ce qui suit
        //$newData = ['user_id' => $this->Auth->user('id')];
        //$article = $this->Articles->patchEntity($article, $newData);
        if ($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);

    // Ajoute seulement la liste des catégories pour pouvoir choisir
    // une catégorie pour un article
    $categories = $this->Articles->Categories->find('treeList');
    $this->set(compact('categories'));
}
```

La fonction `user()` fournie par le component retourne toute colonne à partir de l'utilisateur connecté courant. Nous avons utilisé cette méthode pour ajouter les données dans les infos requêtées qui sont sauvegardées.

Sécurisons maintenant notre app pour empêcher certains auteurs de modifier ou supprimer les articles des autres. Des règles basiques pour notre app sont que les utilisateurs admin peuvent accéder à tout URL, alors que les utilisateurs normaux (le role auteur) peuvent seulement accéder aux actions permises. Ouvrez encore la classe AppController et ajoutez un peu plus d'options à la config de Auth :

```
// src/Controller/AppController.php

public function initialize()
{
    $this->loadComponent('Flash');
    $this->loadComponent('Auth', [
        'authorize' => ['Controller'], // Ajout de cette ligne
        'loginRedirect' => [
            'controller' => 'Articles',
            'action' => 'index'
        ],
        'logoutRedirect' => [
            'controller' => 'Pages',
            'action' => 'display',
            'home'
        ]
    ]);
}

public function isAuthorized($user)
{
    // Admin peuvent accéder à chaque action
    if (isset($user['role']) && $user['role'] === 'admin') {
        return true;
    }

    // Par défaut refuser
    return false;
}
```

Nous venons de créer un mécanisme très simple d'autorisation. Les utilisateurs avec le role `admin` pourront accéder à toutes les URL du site quand ils sont connectés. Tous les autres utilisateurs – ceux avec le role `author` – auront le même accès que les utilisateurs qui ne sont pas loggés.

Ce n'est pas exactement ce que nous souhaitons. Nous devons fournir plus de règles à notre méthode `isAuthorized()`. Cependant au lieu de le faire dans AppController, nous déléguons la gestion de ces règles supplémentaires à chaque controller individuellement. Les règles que nous allons ajouter à ArticlesController devraient permettre aux auteurs de créer des articles mais évitent aux auteurs de modifier les articles qui ne leur appartiennent pas. Ajoutez le contenu suivant à votre `ArticlesController.php` :

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

public function isAuthorized($user)
{
    // Tous les utilisateurs enregistrés peuvent ajouter des articles
    // Avant 3.4.0 $this->request->param('action') etait utilisée.
    if ($this->request->getParam('action') === 'add') {
        return true;
    }

    // Le propriétaire d'un article peut l'éditer et le supprimer
    // Avant 3.4.0 $this->request->param('action') etait utilisée.
    if (in_array($this->request->getParam('action'), ['edit', 'delete'])) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        // Avant 3.4.0 $this->request->params('pass.0')
        $articleId = (int)$this->request->getParam('pass.0');
        if ($this->Articles->isOwnedBy($articleId, $user['id'])) {
            return true;
        }
    }

    return parent::isAuthorized($user);
}
```

Nous surchargeons maintenant l'appel `isAuthorized()` de AppController's et vérifions à l'intérieur si la classe parente autorise déjà l'utilisateur. Si elle ne le fait pas, alors nous ajoutons juste l'autorisation d'accéder à l'action add, et éventuellement autorisons l'accès pour modifier et supprimer. Une dernière chose à que nous avons oubliée de faire est de dire si l'utilisateur à l'autorisation ou non de modifier l'article, nous appelons une fonction `isOwnedBy()` dans la table Articles. Intégrons la fonction suivante :

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

public function isOwnedBy($articleId, $userId)
{
    return $this->exists(['id' => $articleId, 'user_id' => $userId]);
}
```

Ceci conclut notre tutoriel simple sur l'authentification et les autorisations. Pour sécuriser le Controller UsersController, vous pouvez suivre la même technique que nous faisions pour ArticlesController, vous pouvez aussi être plus créatif et coder quelque chose de plus général dans AppController basé sur vos propres règles.

Si vous avez besoin de plus de contrôle, nous vous suggérons de lire le guide complet de Auth dans la section *Authentification* où vous en trouverez plus sur la configuration du component, la création de classes d'autorisation personnalisée, et bien plus encore.

Lectures suivantes suggérées

1. /bake/usage Génération basique CRUD de code
2. *Authentification* : Inscription d'utilisateur et connexion

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 [55], 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.

55. http://en.wikipedia.org/wiki/Continuous_integration

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
- 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 [56].
- 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 [57] pour les caractères accentués, le livre utilise UTF-8.
- 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 [58]. 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.

56. http://en.wikipedia.org/wiki/Register_(linguistics)
57. http://en.wikipedia.org/wiki/List_of_XML_and_HTML_character_entity_references
58. http://en.wikipedia.org/wiki/ReStructuredText

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`.

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 <http://php.net>`_
```

Le lien résultant ressemblerait à ceci : Lien externe vers php.net [59]

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 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>`.

59. http://php.net

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 [60] 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.

`.. **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`*

60. http://pypi.python.org/pypi/sphinxcontrib-phpdomain

.. 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.

Nouveau dans la version 2.6.3 : Cette super fonctionnalité a été ajoutée à partir de la version 2.6.3.

Obsolète depuis la version 2.6.3 : Cette vieille fonctionnalité a été dépréciée à partir de la version 2.6.3.

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 [61].

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 [62] 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 [63] a plusieurs développeurs prêts à répondre à vos questions. Les autres possibilités sont le Groupe Google de CakePHP [64] et le déjà populaire Stack Overflow [65].

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.

61. https://github.com/cakephp/cakephp/issues
62. https://github.com/cakephp/cakephp/search?q=it+is+broken&ref=cmdform&type=Issues
63. https://webchat.freenode.net
64. http://groups.google.com/group/cake-php
65. https://stackoverflow.com/questions/tagged/cakephp

— 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 5.6 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 :

```
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 [66].

Récupérez un clone du code source de CakePHP sous GitHub :
— Si vous n'avez pas de compte github [67] , créez-en un.
— Forkez le dépôt de CakePHP [68] 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*.

66. http://git-scm.com/book/
67. http://github.com
68. http://github.com/cakephp/cakephp

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 :

- 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 `3.2.10`, la branche acceptant les nouvelles fonctionnalités sera `3.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 `3.2.2` alors la prochaine fois qu'un comportement peut être cassé sera dans `4.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 [69] sera la propriétaire de toutes les contributions de code. Les contributeurs doivent suivre les Guidelines de la Communauté CakePHP [70].

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-2 [71] 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 [72] 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.

69. http://cakefoundation.org/pages/about
70. http://community.cakephp.org/guidelines
71. http://www.php-fig.org/psr/psr-2/fr/
72. https://github.com/cakephp/cakephp-codesniffer

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 :

```
$matches = array_intersect_key($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) {
    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 .ctp) 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.
```

(suite sur la page suivante)

(suite de la page précédente)

```
 */
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-2 [73], 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();
```

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 [74] suivants :

- @author [75]
- @copyright [76]
- @deprecated [77] Using the `@version <vector> <description>` format, where `version` and `description` are mandatory.

73. http://www.php-fig.org/psr/psr-2/
74. http://phpdoc.org
75. http://phpdoc.org/docs/latest/references/phpdoc/tags/author.html
76. http://phpdoc.org/docs/latest/references/phpdoc/tags/copyright.html
77. http://phpdoc.org/docs/latest/references/phpdoc/tags/deprecated.html

— @example [78]
— @ignore [79]
— @internal [80]
— @link [81]
— @see [82]
— @since [83]
— @version [84]

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.

78. http://phpdoc.org/docs/latest/references/phpdoc/tags/example.html
79. http://phpdoc.org/docs/latest/references/phpdoc/tags/ignore.html
80. http://phpdoc.org/docs/latest/references/phpdoc/tags/internal.html
81. http://phpdoc.org/docs/latest/references/phpdoc/tags/link.html
82. http://phpdoc.org/docs/latest/references/phpdoc/tags/see.html
83. http://phpdoc.org/docs/latest/references/phpdoc/tags/since.html
84. http://phpdoc.org/docs/latest/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);

// 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 [85].

85. http://php.net/require_once

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 (**.ctp**) 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 :

```
$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** [86]) 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 :

```
define('NOM_LONG_DE_CONSTANTE', 2);
```

86. https://tools.ietf.org/html/rfc2606.html

Attention quand vous utilisez empty()/isset()

While `empty()` is an easy to use function, it can mask errors and cause unintended effects when `'0'` and `0` are given. When variables or properties 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,
    ];

    // Not recommended, the key is already set
    if (isset($array['key'])) {
        // ...
    }

    // Recommended
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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 [87] 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.

87. http://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 est rapide et facile à installer. Les conditions minimales requises sont un serveur web et une copie de CakePHP, c'est tout ! Bien que ce manuel se focalise principalement sur la configuration avec Apache (parceque c'est la plus simple à installer et configurer), CakePHP fonctionnera sur une diversité de serveurs web tels que nginx, LightHTTPD ou Microsoft IIS.

Exigences

- Un serveur HTTP. Par exemple : Apache. mod_rewrite est préférable, mais en aucun cas nécessaire.
- PHP 5.6 ou plus (y compris PHP 7.4)
- L'extension PHP mbstring
- L'extension PHP intl
- L'extension PHP simplexml

Note : Avec XAMPP et WAMP, l'extension mbstring fonctionne par défaut.

Dans XAMPP, l'extension intl est incluse mais vous devez décommenter `extension=php_intl.dll` 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 (5.5.3 ou supérieur)
- MariaDB (5.5 ou supérieur)
- PostgreSQL

— Microsoft SQL Server (2008 ou supérieur)
— SQLite 3

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 5.6 (CLI) ou supérieur. Le serveur web doit utiliser la même version de PHP que votre interface en ligne de commande.

Installer Composer

CakePHP utilise Composer [88], 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 [89] 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 [90]. D'autres instructions pour l'installeur Windows de Composer se trouvent dans le README ici [91].

Créer un Projet CakePHP

Maintenant que vous avez téléchargé et installé Composer, imaginons que vous souhaitiez créer une nouvelle application CakePHP dans le dossier my_app_name. Pour ceci vous pouvez lancer la commande suivante :

```
php composer.phar create-project --prefer-dist cakephp/app:^3.8 my_app_name
```

Ou si Composer est installé globalement :

```
composer self-update && composer create-project --prefer-dist cakephp/app:^3.8 my_app_
↪name
```

Une fois que Composer finit 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.

88. http://getcomposer.org
89. https://getcomposer.org/download/
90. https://github.com/composer/windows-setup/releases/
91. https://github.com/composer/windows-setup

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 : **src/Template/Pages/home.ctp**.

Bien que Composer soit la méthode d'installation recommandée, il existe des versions pré-installées disponibles sur Github [92]. 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.

Rester à jour avec les derniers changements de CakePHP

Par défaut le **composer.json** de l'application ressemble à cela :

```
"require": {
    "cakephp/cakephp": "3.8.*"
}
```

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 `^3.8` pour recevoir également les dernières versions mineures stables de la branche 3.x.

Si vous voulez rester à jour avec les derniers changements non stables de CakePHP, vous pouvez changer la version en `dev-master` le **composer.json** de votre application :

```
"require": {
    "cakephp/cakephp": "dev-master"
}
```

Notez que ce n'est pas recommandé, puisque votre application peut cesser de fonctionner quand la prochaine version majeure sera déployée. De plus, Composer ne met pas en cache les branches de développement, ce qui ralentit les Composer installs/updates consécutifs.

Installation en utilisant Oven

Une autre manière rapide d'installer CakePHP est d'utiliser Oven [93]. Il s'agit d'un simple script PHP qui vérifie si vous respectez les recommandations systèmes, installe le squelette d'application CakePHP et met en place l'environnement de développement.

Après l'installation, votre application CakePHP est prête !

Note : IMPORTANT : Ceci n'est pas un script de déploiement. Il est destiné à aider les développeur à installer CakePHP pour la première fois et à rapidement mettre en place un environnement de développement. Les environnements de production devraient prendre en compte d'autres facteurs comme les permissions de fichiers, les configurations de vhost, etc.

92. https://github.com/cakephp/cakephp/tags
93. https://github.com/CakeDC/oven

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`.

De même, 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` (ou `bin/cake.php`) 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'exécutions (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/
    src/
    tests/
    tmp/
    vendor/
    webroot/ (ce répertoire est défini comme DocumentRoot)
    .gitignore
    .htaccess
    .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 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 http://wiki.apache.org/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 [QSA,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 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
```

(suite sur la page suivante)

94. http://clickontyler.com/virtualhostx/

(suite de la page précédente)

```
    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;
    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;
```

(suite sur la page suivante)

(suite de la page précédente)

```
        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.

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[95] pour installer l'URL Rewrite Module 2.0[96] ou téléchargez-le directement (32-bit[97] / 64-bit[98]).
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 :

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <system.webServer>
        <rewrite>
            <rules>
                <rule name="Exclude direct access to webroot/*"
                  stopProcessing="true">
                    <match url="^webroot/(.*)$" ignoreCase="false" />
                    <action type="None" />
                </rule>
                <rule name="Rewrite routed access to assets(img, css, files, js,␣
↪favicon)"
                  stopProcessing="true">
                    <match url="^(img|css|files|js|favicon.ico)(.*)$" />
                    <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>
```

95. http://www.microsoft.com/web/downloads/platform.aspx
96. http://www.iis.net/downloads/microsoft/url-rewrite
97. http://www.microsoft.com/en-us/download/details.aspx?id=5747
98. http://www.microsoft.com/en-us/download/details.aspx?id=7435

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.

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 options de configurations 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 `config()` pour faciliter la configuration et la rendre plus transparente.

Charger les Fichiers de Configuration Supplémentaires

Si votre application a plusieurs 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());
Configure::load('app', 'default', false);
Configure::load('other_config', 'default');
```

Vous pouvez aussi utiliser des fichiers de configuration supplémentaires pour surcharger un environnement spécifique. Chaque fichier chargé après **app.php** peut redéfinir les valeurs déclarées précédemment ce qui vous permet de personnaliser la configuration pour les environnements de développement ou intermédiaires.

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, créer un nouvel autoloader en lançant `php composer.phar dumpautoload`.

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.

App.encoding Définit l'encodage que votre application utilise. Cet encodage est utilisé pour générer le charset dans le layout, et les entities encodés. Cela 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.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.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.

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.

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 de la classe 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* 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 *Connecter les 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' => [
                APP . 'Template' . DS,
                APP . 'Template2' . DS
            ],
            'locales' => [
                APP . 'Locale' . DS
            ]
        ]
    ]
];
```

Les chemins doivent finir par un séparateur de répertoire, ou ils ne fonctionneront pas correctement.

Configuration de Inflection

Regardez *Configuration d'Inflexion* pour plus d'informations.

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 [99]. 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*

99. http://12factor.net/

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 [100] pour faciliter l'utilisation des variables d'environnement. 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.

Modifié dans la version 3.5.0 : Support de la librairie dotenv ajouté au squelette d'application.

Classe Configure

class `Cake\Core\`**`Configure`**

La nouvelle 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 tentative 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 nous avons mis en place.

Vous pouvez accéder à `Configure` partout dans votre application :

```
Configure::read('debug');
```

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 dans des groupes logiques.

L'exemple ci-dessus pourrait aussi être écrit en un appel unique :

100. https://github.com/josegonzalez/php-dotenv

```
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. C'est particulièrement pratique pour les interactions JSON quand les informations de debug peuvent entraîner des problèmes de parsing.

Lire les Données de Configuration

static `Cake\Core\Configure::read`(*$key = null*)

Utilisée pour lire les données de configuration à partir de l'application. Par défaut, la valeur de debug de CakePHP est au plus important. Si une clé est fournie, la donnée est retournée. En utilisant nos exemples du write() ci-dessus, nous pouvons lire cette donnée :

```
Configure::read('Company.name');     // Renvoie: 'Pizza, Inc.'
Configure::read('Company.slogan');  // Renvoie: 'Pizza for your body and soul'

Configure::read('Company');

//yields:
['name' => 'Pizza, Inc.', 'slogan' => 'Pizza for your body and soul'];
```

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'];
```

Nouveau dans la version 3.1.7 : `Configure::readOrFail()` a été ajoutée dans 3.1.7

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-null :

```
$exists = Configure::check('Company.name');
```

Supprimer une Donnée de Configuration

static `Cake\Core\Configure::delete`(*$key*)

Utilisée pour supprimer l'information à partir 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.

Lire et Ecrire les Fichiers de Configuration

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 [101] pour plus d'informations sur les fichiers ini spécifiés. 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');
```

101. http://php.net/parse_ini_file

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 :

```
// 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 ne vont pas toujours écraser 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 PhpReader. 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 [102]. Les moteurs inclus dans CakePHP sont :

- JsonConfig [103]
- IniConfig [104]
- PhpConfig [105]

Par défaut, votre application utilisera `PhpConfig`.

Bootstrapping CakePHP

Si vous avez des besoins de configuration supplémentaires, utilisez le fichier bootstrap de CakePHP dans **config/bootstrap.php**. Ce fichier est inclus juste avant chaque requête et commande CLI.

Ce fichier est idéal pour un certain nombre de tâches de bootstrapping courantes :

- Définir des fonctions commodes.
- Déclarer des constantes.
- Créer des configurations de cache.
- Définir la configuration des logs.
- Configurer les inflections personnalisées.
- Charger les fichiers de configuration.

Il pourrait être tentant de placer des fonctions de formatage ici pour les utiliser dans vos controllers. Comme vous le verrez dans les documentations sur les *Controllers (Contrôleurs)* et les *Views (Vues)*, il y a de meilleurs moyens pour vous d'ajouter de la logique personnalisée dans votre application.

102. https://api.cakephp.org/3.x/class-Cake.Core.Configure.ConfigEngineInterface.html
103. https://api.cakephp.org/3.x/class-Cake.Core.Configure.Engine.JsonConfig.html
104. https://api.cakephp.org/3.x/class-Cake.Core.Configure.Engine.IniConfig.html
105. https://api.cakephp.org/3.x/class-Cake.Core.Configure.Engine.PhpConfig.html

Application : :bootstrap()

En plus du fichier **config/bootstrap.php** qui doit être utilisé pour faire de la configuration « bas niveau » de votre application, vous pouvez également utiliser la méthode « hook » `Application::bootstrap()` pour charger / initialiser des plugins et attacher des écouteurs d'événements globaux :

```
// in src/Application.php
namespace App;

use Cake\Core\Plugin;
use Cake\Http\BaseApplication;

class Application extends BaseApplication
{
    public function bootstrap()
    {
        // Appeler la méthode parente permet de faire le `require_once`
        // pour charger le fichier config/bootstrap.php
        parent::bootstrap();

        Plugin::load('MyPlugin', ['bootstrap' => true, 'routes' => true]);
    }
}
```

Charger les plugins et les événements dans `Application::bootstrap()` rend les *Test d'Intégrations des Controllers* plus faciles car les événements et les routes seront ainsi à nouveau traités pour chaque méthode de test.

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 panel 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;
// Prior to 3.6 use Cake\Network\Exception\NotFoundException
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);
        }
    });
}
```

CHAPITRE 8

Routing

class Cake\Routing**Router**

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 **routes.php** :

```
use Cake\Routing\Router;

// EN utilisant le route builder scopé.
Router::scope('/', function ($routes) {
    $routes->connect('/', ['controller' => 'Articles', 'action' => 'index']);
});

// En utilisant la méthode statique.
Router::connect('/', ['controller' => 'Articles', 'action' => 'index']);
```

`Router` fournit deux interfaces pour connecter les routes. La méthode statique est une interface retro-compatible, alors que le builder scopé (lié la portée) offre une syntaxe plus laconique pour construire des routes multiples, et de meilleures performances.

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 :

```
$routes->connect(
    '/articles/:id',
    ['controller' => 'Articles', 'action' => 'view'],
)
->setPatterns(['id' => '\d+'])
->setPass(['id']);

// Avant 3.5, utilisez les tableaux 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(
    '/login',
    ['controller' => 'Users', 'action' => 'login'],
    ['_name' => 'login']
);

use Cake\Routing\Router;

echo Router::url(['_name' => 'login']);
// Va afficher
/login
```

Pour aider à garder votre code de router « DRY », le router apporte le concept de "scopes". Un scope (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 :

```
Router::scope('/blog', ['plugin' => 'Blog'], function ($routes) {
    $routes->connect('/', ['controller' => 'Articles']);
```

(suite sur la page suivante)

(suite de la page précédente)

```
});
```

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

`Cake\Routing\Router::connect`(*$route*, *$defaults = []*, *$options = []*)

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 opération. 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\Route\DashedRoute;

Router::scope('/', function ($routes) {
    // Connecte la route de 'fallback' générique
    $routes->fallbacks(DashedRoute::class);
});
```

La méthode `connect()` prend trois paramètres : l'URL que vous souhaitez faire correspondre, les valeurs par défaut pour les éléments de votre route, et les règles d'expression régulière 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',
    ['default' => 'defaultValue'],
    ['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 pattern template exact fourni.

Une fois que vous spécifiez une URL, vous utilisez les deux derniers paramètres de `connect()` pour dire à CakePHP quoi faire avec une requête une fois qu'elle a été matchée. Le deuxième paramètre est un tableau associatif. Les clés du tableau devraient être appelées après les éléments de route dans l'URL, ou les éléments par défaut : `:controller`, `:action`, et `:plugin`. Les valeurs dans le tableau sont les valeurs par défaut pour ces clés. Regardons quelques exemples simples avant que nous commencions à voir l'utilisation du troisième paramètre de `connect()` :

```
$routes->connect(
    '/pages/*',
    ['controller' => 'Pages', 'action' => 'display']
);
```

Cette route est trouvée dans le fichier routes.php distribué avec CakePHP. Cette route matche toute URL commençant par `/pages/` et il tend vers l'action `display()` de `PagesController` La requête `/pages/products` serait mappé vers `PagesController->display('products')`.

En plus de l'étoile greedy `/*` 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 de `the-example-/-and-proof`.

Vous pouvez utiliser le deuxième paramètre de `connect()` pour fournir tout paramètre de routing qui est composé 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 peut impliquer la création de segments d'URL qui ne correspondent pas aux noms de vos controllers ou de vos models. 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 les entrées utilisateur ou les 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',
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ['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". Ces méthodes 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.

Nouveau dans la version 3.5.0 : Les méthodes pour les verbes HTTP ont été ajoutées dans 3.5.0

Les Eléments de Route

Vous pouvez spécifier vos propres éléments de route et ce faisant cela vous donne le pouvoir de définir des places 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, toute expression non `/` sera traitée comme une partie du paramètre :

```
$routes->connect(
    '/:controller/:id',
    ['action' => 'view']
)->setPatterns(['id' => '[0-9]+']);

// Avant 3.5, utilisez les options de tableau
$routes->connect(
    '/:controller/:id',
    ['action' => 'view'],
    ['id' => '[0-9]+']
);
```

Cet exemple simple montre comment créer une manière rapide de voir les models à partir de tout controller 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 controller 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.
$routes->scope('/', function ($routes) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
    $routes->setRouteClass(DashedRoute::class);
    $routes->connect('/:controller/:id', ['action' => 'view'])
        ->setPatterns(['id' => '[0-9]+']);

    // Avant 3.5, utilisez le tableau d'options
    $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. Si vous avez besoin d'URLs en minuscule avec des underscores en migrant d'une application CakePHP 2.x, vous pouvez utiliser à la place la classe `InflectedRoute`.

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 :

```
// Avant 3.5, utilisez le tableau d'options à la place de setPatterns()
$routes->connect(
    '/:userShortcut',
    ['controller' => 'Teachers', 'action' => 'profile', 1],
)->setPatterns(['userShortcut' => '(?i:principal)']);
```

Un exemple de plus, et vous serez un pro du routing :

```
// Avant 3.5, utilisez le tableau d'options à la place de setPatterns()
$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 groupement) ne sont pas supportées dans les expressions régulières. Vous pouvez toujours spécifier des alternatives, comme dessus, mais ne pas grouper avec les parenthèses.

Une fois définie, cette route va matcher `/articles/2007/02/01`, `/posts/2004/11/16`, gérant les requêtes pour les actions `index()` de ses controllers respectifs, avec les paramètres de date dans `$this->request->getParam()`.

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 *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.
- `_full` Si à `true`, la constante *FULL_BASE_URL* va être ajoutée devant les URLS générées.
- `#` Vous permet de définir les fragments de hash d'URL.
- `_ssl` Défini à `true` pour convertir l'URL générée à https, ou `false` pour forcer http.
- `_method` Defini 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'],
)
// 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',
])
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Autorise également l'extension JSON
->setExtensions(['json'])

// Définit 'lang' pour être un paramètre persistant
->setPersist(['lang']);
```

Nouveau dans la version 3.5.0 : Les méthodes fluides ont été ajoutées dans 3.5.0

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 aux arguments à la place. L'option `pass` défini une liste des éléments de route doit également être rendu disponible en tant qu'arguments passé aux fonctions du controller :

```
// 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" avec
    // articleId dans votre action.
    ->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.ctp
// 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,
```

(suite sur la page suivante)

(suite de la page précédente)

```
    'CakePHP_Rocks'
]);
```

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 pour 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 (3.5.0+)
$routes->post(
    '/logout',
    ['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 permtttant de définir des préfixes de nom dans chaque scope :

```
Router::scope('/api', ['_namePrefix' => 'api:'], function ($routes) {
    // le nom de cette route sera `api:ping`
    $routes->get('/ping', ['controller' => 'Pings'], 'ping');
});

Router::plugin('Contacts', ['_namePrefix' => 'contacts:'], function ($routes) {
    // Connecte les routes.
});

Router::prefix('Admin', ['_namePrefix' => 'admin:'], function ($routes) {
    // Connecte les routes.
});
```

Vous pouvez aussi utiliser l'option `_namePrefix` dans les scopes imbriqués et elle fonctionne comme vous pouvez vous y attendre :

```
Router::plugin('Contacts', ['_namePrefix' => 'contacts:'], function ($routes) {
    $routes->scope('/api', ['_namePrefix' => 'api:'], function ($routes) {
        // Le nom de cette route sera `contacts:api:ping`
        $routes->get('/ping', ['controller' => 'Pings'], '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`.

Nouveau dans la version 3.1 : L'option `_namePrefix` a été ajoutée dans 3.1

Prefix de Routage

static `Cake\Routing\Router::`**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 depuis le fichier de configuration du cœur en configurant les préfixes avec Routing.prefixes. Les préfixes peuvent être soit activés en utilisant la valeur de configuration `Routing.prefixes`, soit en définissant la clé `prefix` avec un appel de `Router::connect()` :

```
use Cake\Routing\Route\DashedRoute;

Router::prefix('admin', function ($routes) {
    // Toutes les routes ici seront préfixées avec `/admin` et auront
    // l'élément de route prefix => admin ajouté.
    $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 de plus petits et/ou de plus simples controllers. 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 **src/Template/Admin/Users/edit.ctp**.

Vous pouvez faire correspondre l'URL /admin à votre action `index()` du controller Pages en utilisant la route suivante :

```
Router::prefix('admin', function ($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` :

```
Router::prefix('admin', ['param' => 'value'], function ($routes) {
    // Routes connectées ici sont préfixées par '/admin' et
    // ont la clé 'param' de routing définie.
    $routes->connect('/:controller');
});
```

Vous pouvez aussi définir les préfixes dans les scopes de plugin :

```
Router::plugin('DebugKit', function ($routes) {
    $routes->prefix('admin', function ($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 :

```
Router::prefix('manager', function ($routes) {
    $routes->prefix('admin', function ($routes) {
        $routes->connect('/:controller');
    });
});
```

Ce qui est au-dessus va créer un template de route de type `/manager/admin/:controller`. 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. Voici comment construire ce lien en utilisant le helper HTML :

```
// Aller vers une route préfixée.
echo $this->Html->link(
    'Manage articles',
    ['prefix' => 'manager', '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.

Routing des Plugins

static `Cake\Routing\Router::`**plugin**(*$name*, *$options = []*, *$callback*)

Les routes des *Plugins* sont plus faciles à créer en utilisant la méthode `plugin()`. Cette méthode crée un nouveau scope pour les routes de plugin :

```
Router::plugin('DebugKit', function ($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` :

```
Router::plugin('DebugKit', ['path' => '/debugger'], function ($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 :

```
Router::prefix('admin', function ($routes) {
    $routes->plugin('DebugKit', function ($routes) {
        $routes->connect('/:controller');
    });
});
```

Le code ci-dessus devrait créer une route similaire à `/admin/debug_kit/:controller`. Elle devrait avoir 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, controllers, 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;

Router::plugin('ToDo', ['path' => 'to-do'], function ($routes) {
    $routes->fallbacks(DashedRoute::class);
});
```

Matching des Méthodes HTTP Spécifiques

Les routes peuvent « matcher » des méthodes HTTP spécifiques en utilisant les méthodes spécifiques :

```
Router::scope('/', function($routes) {
    // Cette route matchera seulement les requêtes POST.
    $routes->post(
        '/reviews/start',
        ['controller' => 'Reviews', 'action' => 'start']
    );

    // Matcher plusieurs verbes
    // Avant 3.5, utilisez $options['_method'] pour définir les méthodes
    $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 :

```
Router::scope('/', function($routes) {
    // Cette route ne va "matcher" que sur le domaine http://images.example.com
    // Avant 3.5, utilisez l'option _host
    $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-log.png',
        ['controller' => 'Images', 'action' => 'oldLogo']
    )->setHost('*.example.com');
});
```

L'option `_host` n'affecte que le parsing des URL depuis les requêtes et n'intervient jamais dans la génération d'URL.

Nouveau dans la version 3.4.0 : L'option `_host` a été ajoutée dans la version 3.4.0

Routing des Extensions de Fichier

static `Cake\Routing\Router::extensions` (*string|array|null $extensions*, *$merge = true*)

Pour manipuler différentes extensions de fichier avec vos routes, vous pouvez définir vos extensions de manière globale ou dans un *scope*. Définir des extensions globales peut se faire via la méthode static `Router::extensions()`

Router : :extensions(["json", "xml"]) ; // . . .

Ceci affectera **toutes** les routes qui seront connectées **après** cet appel, quelque soit leur *scope*.

Pour restreindre les extensions à un *scope* spécifique, vous pouvez les définir en utilisant la méthode `Cake\Routing\RouteBuilder::extensions()` :

```
Router::scope('/', function ($routes) {
    // Avant 3.5.0 utilisez `extensions()`
    $routes->setExtensions(['json', 'xml']);
    // ...
});
```

Cela activera les extensions pour toutes les routes qui seront définies dans ce scope **après** l'appel à `extensions()`, tout en incluant les routes inclus dans les scopes imbriqués. De la même manière que la méthode `Router::extensions()`, toutes les routes connectées avant cet appel n'hériteront pas de ces 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 :

```
Router::scope('/page', function ($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.

Connecter des Middlewares à un scope

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.

Avant qu'un middleware ne puisse être appliqué à un scope, il a besoin d'être enregistré dans la collection de routes :

```
// in config/routes.php
use Cake\Http\Middleware\CsrfProtectionMiddleware;
use Cake\Http\Middleware\EncryptedCookieMiddleware;

Router::scope('/', function ($routes) {
    $routes->registerMiddleware('csrf', new CsrfProtectionMiddleware());
    $routes->registerMiddleware('cookies', new EncryptedCookieMiddleware());
});
```

Une fois enregistré dans le builder de routes, le middleware peut être appliqué à des scopes spécifiques :

```
$routes->scope('/cms', function ($routes) {
    // Active les middlewares enregistrés pour ce scope.
    $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 :

```
$routes->scope('/api', function ($routes) {
    $routes->applyMiddleware('ratelimit', 'auth.api');
    $routes->scope('/v1', function ($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 :

```
$routes->scope('/blog', function ($routes) {
    $routes->applyMiddleware('auth');
    // Connecter les actions qui nécessitent l'authentification aux 'blog' ici
});
$routes->scope('/blog', function ($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']);

// Application du groupe
$routes->applyMiddleware('web');
```

Nouveau dans la version 3.5.0 : Le support des middlewares par scope et des groupes de middlewares a été ajouté dans 3.5.0

Créer des Routes RESTful

Le router rend facile la génération des routes RESTful pour vos controllers. Les routes RESTful sont utiles lorsque vous créez des points de terminaison d'API pour vos applications. Si nous voulions permettre l'accès à une base de données REST, nous ferions quelque chose comme ceci :

```
//Dans config/routes.php

Router::scope('/', function ($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	Controller action invoked
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)

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 POST _method
2. Le X_HTTP_METHOD_OVERRIDE
3. Le header REQUEST_METHOD

La variable POST _method est utile dans l'utilisation d'un navigateur comme un 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 :

```
Router::scope('/api', function ($routes) {
    $routes->resources('Articles', function ($routes) {
        $routes->resources('Comments');
    });
});
```

Le code ci-dessus va générer une ressource de routes 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');
```

By default resource routes map to the same prefix as the containing scope. If you have both nested and non-nested resource controllers you can use a different controller in each context by using prefixes :

```
Router::scope('/api', function ($routes) {
    $routes->resources('Articles', function ($routes) {
        $routes->resources('Comments', ['prefix' => 'articles']);
    });
});
```

L'exemple ci-dessus mapperait le champs "Comments" vers `App\Controller\Articles\CommentsController`. Une séparation des controllers 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.

Nouveau dans la version 3.3 : L'option `prefix` a été ajoutée à `resources()` dans la version 3.3.

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 `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' => 'DELETE',
            '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".

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()` :

```
Router::scope('/', function ($routes) {
    $routes->resources('books', [
        'connectOptions' => [
            'routeClass' => 'ApiRoute',
        ]
    ];
});
```

Inflection de l'URL pour les Routes Ressource

Par défaut le fragment d'URL pour les controllers à 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` :

```
Router::scope('/', function ($routes) {
    $routes->resources('BlogPosts', [
        'inflect' => 'dasherize' // Utilisera ``Inflector::dasherize()``
    ]);
})
```

Ce qui est au-dessus va générer des URLs de style **/blog-posts***.

Note : Depuis CakePHP 3.1, le squelette de l'app officiel utilise `DashedRoute` comme classe de route par défaut. Donc il est recommandé d'utiliser l'option `'inflect' => 'dasherize'` pour connecter les routes resssource afin de garder la cohérence de l'URL.

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` :

```
Router::scope('/', function ($routes) {
    $routes->resources('BlogPosts', ['path' => 'posts']);
});
```

Nouveau dans la version 3.5.0 : L'option `path` a été ajoutée dans 3.5.0.

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 possible de forcer des paramètres particuliers comme étant des paramètres passés également.

Si vous alliez visiter l'URL mentionné précédemment, et que vous aviez une action de controller qui ressemblait à cela :

```
class CalendarsController extends AppController
{
    public function view($arg1, $arg2)
    {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        debug(func_get_args());
    }
}
```

Vous auriez le résultat suivant :

```
Array
(
    [0] => recent
    [1] => mark
)
```

La même donnée est aussi disponible dans `$this->request->getParam('pass')` dans vos controllers, vues, et helpers. Les valeurs dans le tableau pass sont indicées numériquement basé sur l'ordre dans lequel elles apparaissent dans l'URL appelé :

```
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` a une clé numérique, il est traité comme un argument passé.

Générer des URLs

static `Cake\Routing\Router::url`(*$url = null*, *$full = false*)

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. En utilisant des *tableaux de routing* pour définir vos URLs, vous pouvez configurer les routes plus tard et les URLs générés vont automatiquement être mises à jour.

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 :

```
$this->Html->link(
    'View',
    ['controller' => 'Articles', 'action' => 'view', $id]
);
```

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.

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 :

```
Router::url([
    'controller' => 'Articles',
    'action' => 'index',
    '?' => ['page' => 1],
    '#' => 'top'
]);

// Cela générera une URL comme:
/articles/index?page=1#top
```

Le Router convertira également tout paramètre inconnu du tableau de routing en paramètre d'URL. Le `?` est disponible pour la rétrocompatibilité avec les anciennes versions de CakePHP.

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 constante *FULL_BASE_URL* va être ajoutée devant les URLS générées.
- `_ssl` Défini à `true` pour convertir l'URL générée à https, ou `false` pour forcer http.
- `_name` Nom de route. Si vous avez configuré les routes nommées, vous pouvez utiliser cette clé pour les spécifier.

Routing de Redirection

Le routing de redirection permet de créer des statuts HTTP de redirection 30x pour les routes entrantes et les 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 :

```
Router::scope('/', function ($routes) {
    $routes->redirect(
        '/home/*',
        ['controller' => 'Articles', 'action' => 'view'],
        ['persist' => true]
        // Or ['persist'=>['id']] for default routing where the
        // view action expects $id as an 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 :

```
Router::scope('/', function ($routes) {
    $routes->redirect('/articles/*', 'http://google.com', ['status' => 302]);
});
```

Cela redirigerait `/articles/*` vers `http://google.com` avec un statut HTTP 302.

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`.
- 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 controller & action. Renvoyez `false` 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.
$routes->scope('/', function ($routes) {
    // Avant 3.5.0, utilisez `routeClass()`
    $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\Router::`**`defaultRouteClass`**(*$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 `Router::defaultRouteClass()` avant de définir la moindre route et éviter de spécifier l'option `routeClass` pour chaque route. Par exemple en utilisant :

```
use Cake\Routing\Route\InflectedRoute;

Router::defaultRouteClass(InflectedRoute::class);
```

Cela provoquera l'utilisation de la classe `InflectedRoute` pour toutes les routes suivantes. Appeler la méthode sans argument va retourner la classe de route courante par défaut.

Méthode Fallbacks

`Cake\Routing\Router::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 `Router::defaultRouteClass()` 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 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.

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 ($params, $request) {
    if ($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 ($params, $request) {
    if (empty($params['plugin']) || $params['plugin'] !== 'MyPlugin' || empty($params[
↪'controller'])) {
        return $params;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }
    if ($params['controller'] === 'Languages' && $params['action'] === 'view') {
        $params['controller'] = 'Locations';
        $params['action'] = 'index';
        $params['language'] = $params[0];
        unset($params[0]);
    }
    return $params;
});
```

Gérer les Paramètres Nommés dans les URLs

Bien que les paramètres nommés aient été retirés dans CakePHP 3.0, les applications peuvent publier des URLs les contenant. Vous pouvez continuer à accepter les URLs contenant les paramètres nommés.

Dans la méthode de votre `beforeFilter()`, vous pouvez appeler `parseNamedParams()` pour extraire tout paramètre nommé à partir des arguments passés :

```
public function beforeFilter(Event $event)
{
    parent::beforeFilter($event);
    Router::parseNamedParams($this->request);
}
```

Ceci va remplir `$this->request->getParam('named')` avec tout paramètre nommé trouvé dans les arguments passés. Tout argument passé qui a été interprété comme un paramètre nommé, sera retiré de la liste des arguments passés.

Filtres du Dispatcher

Obsolète depuis la version 3.3.0 : Depuis la version 3.3.0 , les Filtres de Dispatcher sont dépréciés. Vous devriez maintenant utiliser le /controllers/middleware à la place.

Il y a plusieurs raisons de vouloir exécuter un bout de code avant que tout code de controller ne soit lancé ou juste avant que la réponse ne soit envoyée au client, comme la mise en cache de la réponse, le tunning de header, une authentification spéciale ou juste pour fournir l'accès à une réponse de l'API critique plus rapidement qu'avec un cycle complet de dispatch de requêtes.

CakePHP fournit une interface propre et extensible pour de tels cas pour attacher les filtres au cycle de dispatch, de la même façon qu'une couche middleware pour fournir des services empilables ou des routines pour chaque requête. Nous les appelons *Dispatcher Filters*.

Filtres Intégrés

CakePHP fournit plusieurs filtres de dispatcher intégrés. Ils gèrent des fonctionnalités habituelles dont toutes les applications vont avoir besoin. Les filtres intégrés sont :

— `AssetFilter` vérifie si la requête fait référence au fichier d'asset de plugin ou du theme, comme un fichier CSS, un fichier JavaScript ou une image stockée soit dans le dossier webroot d'un plugin ou celui qui correspond pour un Theme. Il va servir le fichier correspondant s'il est trouvé, stoppant le reste du cycle de dispatch :

```
// Utilisez options pour définir le cacheTime de vos assets statiques
// S'il n'est pas défini, il est de 1 heure (+1 hour) par défaut.
DispatcherFactory::add('Asset', ['cacheTime' => '+24 hours']);
```

— `RoutingFilter` applique les règles de routing de l'application pour l'URL de la requête. Remplit `$request->getParam()` avec les résultats de routing.
— `ControllerFactory` utilise `$request->getParam()` pour localiser le controller qui gère la requête courante.
— `LocaleSelector` active le langage automatiquement en changeant le header `Accept-Language` envoyé par le navigateur.

Utiliser les Filtres

Les filtres sont habituellement activés dans le fichier **bootstrap.php** de votre application, mais vous pouvez les charger à n'importe quel moment avant que la requête ne soit dispatchée. Ajouter et retirer les filtres se fait avec `Cake\Routing\DispatcherFactory`. Par défaut, le template d'une application CakePHP est fourni avec un couple de classes filter déjà activées pour toutes les requêtes ; Regardons la façon dont elles sont ajoutées :

```
DispatcherFactory::add('Routing');
DispatcherFactory::add('ControllerFactory');

// La syntaxe de plugin est aussi possible
DispatcherFactory::add('PluginName.DispatcherName');

// Utilisez les options pour définir la priorité
DispatcherFactory::add('Asset', ['priority' => 1]);
```

Les filtres Dispatcher avec une priorité `priority` supérieure (nombres les plus faibles) - seront exécutés les premiers. La priorité est par défaut à `10`.

Alors qu'utiliser le nom de la chaîne est pratique, vous pouvez aussi passer les instances dans `add()` :

```
use Cake\Routing\Filter\RoutingFilter;

DispatcherFactory::add(new RoutingFilter());
```

Configurer l'Ordre de Filter

Lors de l'ajout de filtres, vous pouvez contrôler l'ordre dans lequel ils sont appelés en utilisant les priorités du gestionnaire d'event. Alors que les filtres peuvent définir une priorité par défaut en utilisant la propriété `$_priority`, vous pouvez définir une priorité spécifique quand vous attachez le filtre :

```
DispatcherFactory::add('Asset', ['priority' => 1]);
DispatcherFactory::add(new AssetFilter(['priority' => 1]));
```

Plus la priorité est haute, plus le filtre sera appelé tardivement.

Appliquer les Filtres de Façon Conditionnelle

Si vous ne voulez pas exécuter un filtre sur chaque requête, vous pouvez utiliser des conditions pour les appliquer seulement certaines fois. Vous pouvez appliquer les conditions en utilisant les options `for` et `when`. L'option `for` vous laisse faire la correspondance sur des sous-chaines d'URL, alors que l'option `when` vous permet de lancer un callable :

```
// Only runs on requests starting with `/blog`
DispatcherFactory::add('BlogHeader', ['for' => '/blog']);

// Only run on GET requests.
DispatcherFactory::add('Cache', [
    'when' => function ($request, $response) {
        return $request->is('get');
    }
]);
```

The callable provided to `when` should return `true` when the filter should run. The callable can expect to get the current request and response as arguments.

Construire un Filtre

Pour créer un filtre, définissez une classe dans **src/Routing/Filter**. Dans cet exemple, nous allons créer un filtre qui ajoute un cookie de tracking pour la page d'accueil. Premièrement, créez le fichier. Son contenu doit ressembler à ceci :

```
namespace App\Routing\Filter;

use Cake\Event\Event;
use Cake\Routing\DispatcherFilter;

class TrackingCookieFilter extends DispatcherFilter
{

    public function beforeDispatch(Event $event)
    {
        $request = $event->getData('request');
        $response = $event->getData('response');
        if (!$request->getCookie('landing_page')) {
            $response->cookie([
                'name' => 'landing_page',
                'value' => $request->here(),
                'expire' => '+ 1 year',
            ]);
        }
    }
}
```

Enregistrez ce fichier sous **src/Routing/Filter/TrackingCookieFilter.php**. Comme vous pouvez le voir, à l'image des autres classes dans CakePHP, les filtres de dispatcher suivent quelques conventions :

- Les noms de classes finissent par `Filter`.
- Les classes sont dans le namespace `Routing\\Filter`. Par exemple, `App\\Routing\\Filter`.
- Généralement, les filtres étendent `Cake\\Routing\\DispatcherFilter`.

`DispatcherFilter` expose deux méthodes qui peuvent être surchargées dans les sous-classes qui sont `beforeDispatch()` et `afterDispatch()`. Ces méthodes sont exécutées respectivement avant et après l'exécution de tout controller. les deux méthodes reçoivent un objet `Cake\Event\Event` contenant les objets

`ServerRequest` et `Response` (instances de *Cake\Http\ServerRequest* et *Cake\Http\Response*) dans la propriété `data`.

Alors que notre filtre était relativement simple, il y a quelques autres choses intéressantes que nous pouvons réaliser dans les méthodes de filtre. En renvoyant un objet `Response`, vous pouvez court-circuiter le process de dispatch et empêcher le controller d'être appelé. Lorsque vous renvoyez une response, n'oubliez pas d'appeler `$event->stopPropagation()` pour que les autres filtres ne soient pas appelés.

Note : Lorsque la méthode beforeDispatch renvoie une response, le controller, et l'event afterDispatch ne seront pas appelés.

Créons maintenant un autre filtre pour modifier l'en-tête de response de n'importe quelle page publique, dans notre cas ce serait tout ce qui est servi depuis le `PagesController` :

```
namespace App\Routing\Filter;

use Cake\Event\Event;
use Cake\Routing\DispatcherFilter;

class HttpCacheFilter extends DispatcherFilter
{

    public function afterDispatch(Event $event)
    {
        $request = $event->getData('request');
        $response = $event->getData('response');

        if ($response->statusCode() === 200) {
            $response->sharable(true);
            $response->expires(strtotime('+1 day'));
        }
    }
}

// Dans notre bootstrap.php
DispatcherFactory::add('HttpCache', ['for' => '/pages'])
```

Ce filtre enverra un en-tête d'expiration pour 1 jour dans le futur pour toutes responses produites pour le controller pages. Vous pourriez bien entendu faire la même chose dans un controller, ce n'est qu'un exemple de ce qui peut être réalisé avec les filtres. Par exemple, au lieu d'altérer la response, vous pourriez la mettre en cache en utilisant *Cake\Cache\Cache* en servant la response depuis le callback `beforeDispatch()`.

Bien que très puissants, les filtres du dispatcher peuvent également compliquer la maintenance de votre application. Les filtres sont des outils extrêmement puissants lorsqu'ils sont utilisés sagement et ajouter des gestionnaires de responses pour chaque URL dans votre application n'est pas une bonne utilisation. Gardez à l'esprit que tout n'a pas besoin d'être un filtre ; Les *Controllers* et les *Components* sont souvent un choix plus précis pour ajouter tout code de gestionnaire de requête à votre application.

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 incluses 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 envois d'en-têtes (headers), 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.

Depuis la version 3.4.0, l'objet ServerRequest de CakePHP implémente l'interface PSR-7 ServerServerRequestInterface [106] facilitant l'utilisation des librairies en-dehors de CakePHP.

106. http://www.php-fig.org/psr/psr-7/

Paramètres de la Requête

`ServerRequest` propose les paramètres de routing avec la méthode `getParam()` :

```
$controllerName = $this->request->getParam('controller');

// Avant 3.4.0
$controllerName = $this->request->param('controller');
```

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*)

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');

// Avant 3.4.0
$page = $this->request->query('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();
```

Nouveau dans la version 3.4.0 : `getQueryParams()` et `getQuery()` ont été ajoutées dans la version 3.4.0

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()`. Toute donnée de formulaire qui contient un préfix `data` aura ce préfixe supprimé. Par exemple :

```
// Un input avec un attribut de nom égal à 'MyModel[title]' est accessible via
$title = $this->request->getData('MyModel.title');
```

Toute clé qui n'existe pas va retourner `null` :

```
$foo = $this->request->getData('Valeur.qui.n.existe.pas');
// $foo == null
```

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();
```

Nouveau dans la version 3.4.0 : `getServerParams()` a été ajoutée dans la version 3.4.0

Données XML ou JSON

Les applications employant *REST* échangent souvent des données dans des organes post non encodées en URL. Vous pouvez lire les données entrantes dans n'importe quel format en utilisant `Http\ServerRequest::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');
```

Quelques méthodes de desérialization requièrent des paramètres supplémentaires quand elles sont appelées, comme le paramètre de type tableau de `json_decode`. Si vous voulez convertir du XML en objet DOMDocument, `Http\ServerRequest::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');

// Avant la version 3.4.0
$webroot = $request->webroot;
$base = $request->base;
$here = $request->here();
```

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 utilisée. 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 quatre 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 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 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. Depuis la version␣
↪3.3.0
$this->request->addDetector(
    'controller',
    function ($request, $name) {
        return $request->getParam('controller') === $name;
    }
);
```

`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.

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('requested')` Vérifie si la requête a un paramètre de requête "requested" avec la valeur 1.
— `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".

Nouveau dans la version 3.3.0 : Les détecteurs peuvent prendre des paramètres supplémentaires depuis la version 3.3.0.

Données de Session

Pour accéder à la session pour une requête donnée, utilisez la méthode `session()` :

```
$userName = $this->request->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 la méthode HTTP où la requête a été faite :

```
// Affiche POST
echo $request->getMethod();

// Avant la version 3.4.0
echo $request->method();
```

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()
{
    // Only accept POST and DELETE requests
    $this->request->allowMethod(['post', 'delete']);
    ...
}
```

Lire les en-têtes HTTP

Ces méthodes vous permettent d'accéder à n'importe quel en-tête `HTTP_*` qui ont été utilisés dans la requête. Par exemple :

```
// Récupère le header dans une chaîne
$userAgent = $this->request->getHeaderLine('User-Agent');

// Récupère un tableau de toutes les valeurs
$acceptHeader = $this->request->getHeaders();

// Vérifie l'existence d'un header
$hasAcceptHeader = $this->request->hasHeader('Accept');

// Avant to 3.4.0
$userAgent = $this->request->header('User-Agent');
```

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 = false)`

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 forwarded 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();
```

(suite sur la page suivante)

(suite de la page précédente)

```
$scheme = $this->request->scheme();
$clientIp = $this->request->clientIp();
```

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');
```

Cookies

Les cookies de la Request 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');

// 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 (à partir de 3.5.0)
$cookies = $this->request->getCookieCollection()
```

Référez-vous à la documentation de *`Cake\Http\Cookie\CookieCollection`* pour savoir comment travailler avec les collections de cookies.

Nouveau dans la version 3.5.0 : `ServerRequest::getCookieCollection()` a été ajouté dans 3.5.0

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`, `ServerRequestHandlerComponent` 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 tout 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 `withType()` comme ceci :

```
// Ajouter un type vCard
$this->response->withType(['vcf' => 'text/v-card']);

// Configurer la réponse de Type de Contenu pour vcard.
$this->response->withType('vcf');

// Avant 3.4.0
$this->response->type('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 vue de commutation automatique de *RequestHandlerComponent*, si vous l'utilisez.

Envoyer des fichiers

Cake\Http\Response::**withFile**(*$path, $options = []*)

Il y a des fois où vous voulez envoyer des fichiers en réponses de vos requêtes. Vous pouvez le faire en utilisant *Cake\Http\Response::withFile()* :

```
public function sendFile($id)
{
    $file = $this->Attachments->getFile($id);
    $response = $this->response->withFile($file['path']);
    // Retourne la réponse pour éviter que le controller n'essaie de
    // rendre la vue
    return $response;
}

// Avant 3.4.0
```

(suite sur la page suivante)

(suite de la page précédente)

```
$file = $this->Attachments->getFile($id);
$this->response->file($file['path']);
// Retourne la réponse pour éviter que le controller n'essaie de
// rendre la vue
return $this->response;
```

Comme montré dans l'exemple ci-dessus, vous devez passer le chemin du fichier à la méthode. CakePHP va envoyer le bon en-tête de type de contenu si c'est un type de fichier connu listé dans *Cake\Http\Reponse : :$_mimeTypes*. Vous pouvez ajouter des nouveaux types avant d'appeler `Cake\Http\Response::withFile()` en utilisant la méthode `Cake\Http\Response::withType()`.

Si vous voulez, vous pouvez aussi forcer un fichier à être téléchargé au lieu d'être affiché dans le navigateur en spécifiant les options :

```
$response = $this->response->withFile(
    $file['path'],
    ['download' => true, 'name' => 'foo']
);

// Avant 3.4.0
$this->response->file(
    $file['path'],
    ['download' => true, 'name' => 'foo']
);
```

les options possibles sont :

name Le nom vous permet de spécifier un nom de fichier alternatif à envoyer à l'utilisateur.

download Une valeur booléenne indiquant si les en-têtes doivent être définis pour forcer le téléchargement.

Envoyer une Chaîne de Caractères en 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();
    $reponse = $this->response;
    $response->body($icsString);

    $this->response->withType('ics');

    // Force le téléchargement de fichier en option
    $response = $this->response->withDownload('filename_for_download.ics');

    // Retourne l'object pour éviter au controller d'essayer de rendre
    // une vue
    return $response;
}
```

Streaming Resources

Vous pouvez utiliser une fonction de rappel avec `body()` pour convertir des flux de ressources en réponses :

```
$file = fopen('/some/file.png', 'r');
$this->response->body(function () use ($file) {
    rewind($file);
    fpassthru($file);
    fclose($file);
});
```

Les fonctions de rappel 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*)

Les définitions de headers se font 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');

// Avant to 3.4.0 - Définit a header
$this->response->header('Location', 'http://example.com');
```

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']));
```

Nouveau dans la version 3.4.3 : `withStringBody()` was added in 3.4.3

`Cake\Http\Response::withBody($body)`

Pour définir le corps de la réponse, utilisez la méthode `withBody()` qui est fournie par le `Zend\Diactoros\MessageTrait` :

```
$response = $response->withBody($stream);

// Avant 3.4.0, pour définir le corps de la réponse
$this->response->body('My Body');
```

Assurez-vous que `$stream` est un objet de type `Psr\Http\Message\StreamInterface`. Ci-dessous, la manière de créer un nouveau stream.

Vous pouvez également « *streamer* » les réponses depuis des fichiers en utilisant des streams `Zend\Diactoros\Stream` :

```
// Pour "streamer" depuis un fichier
use Zend\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);

// Avant 3.4.0, vous pouvez utiliser la méthode ci-dessous pour créer des
// réponses sous forme de stream
$file = fopen('/some/file.png', 'r');
$this->response->body(function () use ($file) {
    rewind($file);
    fpassthru($file);
    fclose($file);
});
```

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');

// Avant to 3.4.0
$this->response->charset('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()
{
    $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, en espérant 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 affinée 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 modèles, 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 affiner les en-têtes de cache HTTP pour tirer profit du cache du navigateur ou du proxy inverse.

L'En-tête de Contrôle du Cache

`Cake\Http\Response::withSharable`(*$public = null*, *$time = null*)

Utilisé sous le modèle 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 = null*)

Vous pouvez définir l'en-tête `Expires` avec une date et un temps après lesquels la réponse n'est plus considérée comme récente. 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 récent. 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 choses.

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 `checkNotModified()` ou inclure le *Request Handling (Gestion des requêtes)* in your controller :

```
public function index()
{
    $articles = $this->Articles->find('all');
    $response = $this->response->withEtag($this->Articles->generateHash($articles));
    if ($response->checkNotModified($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 = null)`

De même, avec le modèle de validation 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 `checkNotModified()` ou inclure le *Request Handling (Gestion des requêtes)* in your controller :

```
public function view()
{
    $article = $this->Articles->find()->first();
    $response = this->response->withModified($article->modified);
    if ($this->response->checkNotModified($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` :

```
$this->response = $this->response->withVary('User-Agent');
$this->response = $this->response->withVary('Accept-Encoding', 'User-Agent');
$this->response = $this->response->withVary('Accept-Language');
```

Envoyer des Réponses Non-Modifiées

`Cake\Http\Response::checkNotModified(ServerRequest $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 récent. Si oui, il supprime le contenu de la réponse et envoie l'en-tête *304 Not Modified* :

```
// Dans une action de controller.
if ($this->response->checkNotModified($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`*:

```
// Ajoute un cookie avec un tableau en utilisant l'API immutable (3.4.0+)
$this->response = $this->response->withCookie('remember_me', [
    'value' => 'yes',
    'path' => '/',
    'httpOnly' => true,
    'secure' => false,
    'expire' => strtotime('+1 year')
]);

// Avant 3.4.0
$this->response->cookie('remember', [
    'value' => 'yes',
    'path' => '/',
    'httpOnly' => true,
    'secure' => false,
    'expire' => strtotime('+1 year')
]);
```

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 :

```
// À partir de 3.5.0
$this->response = $this->response->withExpiredCookie('remember_me');
```

Définir les En-têtes de Requête d'Origine Croisée (Cross Origin Request Headers = CORS)

Depuis 3.2, vous pouvez utiliser la méthode `cors()` pour définir le Contrôle d'Accès HTTP [107] et ses en-têtes liés avec une interface simple :

```
$this->response->cors($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.

Nouveau dans la version 3.2 : `CorsBuilder` a été ajouté dans 3.2

Erreurs Communes avec les Responses Immutables

Depuis CakePHP 3.4.0, 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.

107. https://developer.mozilla.org/fr/docs/HTTP/Access_control_CORS

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
    '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` :

```
$response = $this->response->withCookie($cookie);
```

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);

// 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();
```

Nouveau dans la version 3.5.0 : `CookieCollection` et `Cookie` ont été ajoutés dans 3.5.0.

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 models sont appelés et que la bonne réponse ou vue est rendue. Les controllers peuvent être imaginés comme une couche au milieu entre le Model et la Vue. Le mieux est de garder des controllers peu chargés, et des models 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 model. 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 model. Dans CakePHP, un controller est nommé d'après le model 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;

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()
    {
        // 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 spécialement. 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 seront toujours surchargées par 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 **src/Template/Recipes/view.ctp**, **src/Template/Recipes/share.ctp**, et **src/Template/Recipes/search.ctp**. 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 couvrons 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:
?>

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);
```

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.

Nouveau dans la version 3.1 : ViewBuilder a été ajouté dans 3.1

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 **src/Template/Recipes/search.ctp** sera utilisé :

```
namespace App\Controller;

class RecipesController extends AppController
{
// ...
    public function search()
    {
        // Render the view in src/Template/Recipes/search.ctp
        $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 **src/Template**. Cela permet un affichage direct des éléments, ce qui est très pratique lors d'appels AJAX :

```
// Rend un élément dans src/Template/Element/ajaxreturn.ctp
$this->render('/Element/ajaxreturn');
```

Le 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 my_action()
    {
        $this->render('custom_file');
    }
}
```

Cela rendrait **src/Template/Posts/custom_file.ctp** au lieu de **src/Template/Posts/my_action.ctp**.

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/src/Template/UserDetails/custom_file.ctp**

Rediriger vers d'Autres Pages

`Cake\Controller\Controller::redirect` (*string|array $url*, *integer $status*)

La méthode de contrôle de flux que vous utiliserez le plus souvent est `Controller::redirect()`. Cette méthode prend son premier paramètre sous la forme d'une URL relative à votre application CakePHP. Quand un utilisateur a réalisé un paiement avec succès, vous aimeriez le rediriger vers un écran affichant le reçu :

```
public function place_order()
{
    // Logique pour finaliser la commande
    if ($success) {
        return $this->redirect(
            ['controller' => 'Orders', 'action' => 'thanks']
        );
    }
    return $this->redirect(
        ['controller' => 'Orders', 'action' => 'confirm']
    );
}
```

La méthode va retourner l'instance de réponse avec les bons headers définis. Vous devrez retourner l'instance de réponse à partir de votre action pour éviter les rendus de view et laisser le dispatcher gérer la bonne redirection.

Vous pouvez aussi utiliser une URL relative ou absolue avec $url :

```
return $this->redirect('/orders/thanks');
return $this->redirect('http://www.example.com');
```

Vous pouvez aussi passer des données à l'action :

```
return $this->redirect(['action' => 'edit', $id]);
```

Le second paramètre de la fonction `Controller::redirect()` vous permet de définir un code de statut HTTP accompagnant la redirection. Vous aurez peut-être besoin d'utiliser le code 301 (document déplacé de façon permanente) ou 303 (voir ailleurs), en fonction de la nature de la redirection.

Si vous avez besoin de rediriger à la page appelante, vous pouvez utiliser :

```
return $this->redirect($this->referer());
```

Un exemple d'utilisation des requêtes en chaînes et hashés ressemblerait à ceci :

```
return $this->redirect([
    'controller' => 'Orders',
    'action' => 'confirm',
    '?' => [
        'product' => 'pizza',
        'quantity' => 5
    ],
    '#' => 'top'
]);
```

L'URL générée serait :

```
http://www.example.com/orders/confirm?product=pizza&quantity=5#top
```

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 Models Supplémentaires

`Cake\Controller\Controller::loadModel`(*string $modelClass*, *string $type*)

La fonction `loadModel()` devient pratique quand vous avez besoin d'utiliser une table de model/collection qui n'est pas le model du controller par défaut ou un de ses models associés :

```
// Dans une méthode de controller.
$this->loadModel('Articles');
$recentArticles = $this->Articles->find('all', [
    'limit' => 5,
    'order' => 'Articles.created DESC'
]);
```

Si vous utilisez un provider de table différent de l'ORM intégré, vous pouvez lier ce système de table dans les controllers de CakePHP en connectant sa méthode factory :

```
// Dans une méthode de controller.
$this->modelFactory(
    'ElasticIndex',
    ['ElasticIndexes', 'factory']
);
```

Après avoir enregistré la table factory, vous pouvez utiliser `loadModel()` pour charger les instances :

```
// Dans une méthode de controller.
$this->loadModel('Locations', 'ElasticIndex');
```

Note : La TableRegistry intégrée dans l'ORM est connectée par défaut comme provider de "Table".

Paginer un Model

`Cake\Controller\Controller::paginate()`

Cette méthode est utilisée pour paginer les résultats retournés par vos models. 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 donne une façon facile de personnaliser la façon dont `paginate()` se comporte :

```
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()
{
    parent::initialize();
    $this->loadComponent('Csrf');
    $this->loadComponent('Comments', Configure::read('Comments'));
}
```

property `Cake\Controller\Controller::$components`

La propriété `$components` de vos controllers vous permet de configurer les components. Les components configurés et leurs dépendances vont être créés par CakePHP pour vous. Lisez la section *Configuration des Components* pour plus d'informations. Comme mentionné plus tôt, la propriété `$components` sera fusionnée avec la propriété définie dans chacune des classes parentes de votre controller.

Configurer les Helpers à Charger

property `Cake\Controller\Controller::$helpers`

Voyons comment dire à un controller de CakePHP que vous avez prévu d'utiliser les classes MVC supplémentaires :

```
class RecipesController extends AppController
{
    public $helpers = ['Form'];
}
```

Chacune de ces variables sont fusionnées avec leurs valeurs héritées, ainsi il n'est pas nécessaire (par exemple) de redéclarer `FormHelper`, ou bien tout ce qui est déclaré dans votre `AppController`.

Obsolète depuis la version 3.0 : Le chargement des helpers depuis le controller est fourni pour des raisons de rétro-compatibilité. Référez-vous à la section suivante pour apprendre à *Configurer les Helpers*.

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`**(*Event $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`**(*Event $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`**(*Event $event*)

Cette méthode est appelée pendant l'event `Controller.shutdown` qui se produit après chaque action du controller, et après que l'affichage est 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\Event;
public function beforeFilter(Event $event)
{
    parent::beforeFilter($event);
}
```

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 **src/Template/Pages/home.ctp**. Ex : Si vous écrivez un fichier de vue **src/Template/Pages/a_propos.ctp**, 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.

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 [108] 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 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 :

108. https://github.com/FriendsOfCake/awesome-cakephp/blob/master/README.md#authentication-and-authorization

```
// Configuration simple
$this->Auth->config('authenticate', ['Form']);

// Passer la configuration
$this->Auth->config('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->config('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`.
- `storage` Classe de stockage. Par défaut à `Session`.
- Les options `scope` et `contain` sont dépréciées dans 3.1. Utilisez un finder personnalisé à la place pour modifier la requête qui récupère l'utilisateur.
- L'option `userFields` a été dépréciée depuis la version 3.1. Utilisez `select()` dans vos finders personnalisés.

Pour configurer les différents champs de l'utilisateur dans la méthode `initialize()` :

```
public function initialize()
{
    parent::initialize();
    $this->loadComponent('Auth', [
        '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()
{
    parent::initialize();
    $this->loadComponent('Auth', [
        'loginAction' => [
```

(suite sur la page suivante)

(suite de la page précédente)

```
            '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()
{
    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;
}
```

Note : L'option `finder` est disponible depuis 3.1. Pour les versions antérieures, vous pouvez utiliser les options `scope` et `contain` pour modifier la requête.

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 les certificats 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. Avant 3.4.0, la valeur de la clé `Auth.redirect` stockée en session était utilisée.
- 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 tel qu'espéré, 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 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 certificats d'identification de l'authentification Basic seront revérifiés à chaque requête.

Avertissement : L'authentification Basic transmet les certificats 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 certificats 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;
use Cake\Event\Event;
use Cake\ORM\Table;

class UsersTable extends Table
{
    public function beforeSave(Event $event)
    {
        $entity = $event->getData('entity');

        if ($entity->isNew()) {
            $hasher = new DefaultPasswordHasher();

            // Generate an API 'token'
            $entity->api_key_plain = Security::hash(Security::randomBytes(32), 'sha256
↪', false);
```

(suite sur la page suivante)

(suite de la page précédente)

```
            // 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
{
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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;

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 **src/Template/Layouts/default.ctp** dans la section body :

```
// Seule cette ligne est nécessaire à partir de 3.4.0.
echo $this->Flash->render();

// Avant 3.4.0, cette ligne sera également nécessaire.
echo $this->Flash->render('auth');
```

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. Avant 3.4.0, la clé par défaut était "auth".
— `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 les configurations 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($password)
    {
        if (strlen($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()
{
    parent::initialize();
    $this->loadComponent('Auth', [
        'authenticate' => [
            'Form' => [
                'passwordHasher' => [
                    'className' => 'Legacy',
                ]
            ]
        ]
    ]);
}
```

Supporter des systèmes légaux 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()
{
    parent::initialize();
    $this->loadComponent('Auth', [
        'authenticate' => [
```

(suite sur la page suivante)

(suite de la page précédente)

```
            '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());
        }
        ...
    }
}
```

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.

Hachage des Mots de Passe pour l'Authentification Digest

Puisque l'authentification Digest nécessite un mot de passe haché dans un format défini par la RFC, afin d'hacher correctement un mot de passe pour l'utilisation de l'authentification Digest, vous devriez utilisez la fonction spéciale `DigestAuthenticate`. Si vous vous apprêtez à combiner l'authentification Digest avec d'autres stratégies d'authentifications, il est aussi recommandé de stocker le mot de passe Digest dans une colonne séparée, pour le hachage normal de mot de passe :

```
namespace App\Model\Table;

use Cake\Auth\DigestAuthenticate;
use Cake\Event\Event;
use Cake\ORM\Table;
```

(suite sur la page suivante)

(suite de la page précédente)

```
class UsersTable extends Table
{

    public function beforeSave(Event $event)
    {
        $entity = $event->data['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'information que pour d'autres mots de passe hachés, basé sur le RFC pour l'authentification Digest.

Note : Le troisième paramètre de DigestAuthenticate : :password() doit correspondre à la valeur de la configuration "realm" définie quand DigestAuthentication était configuré dans AuthComponent : :$authenticate. Par défaut à `env('SCRIPT_NAME')`. Vous devez utiliser une chaîne statique si vous voulez un hachage permanent dans des environnements multiples.

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 en utilisant `AuthComponent::user()` :

```
// 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 "Déconnecte-moi" à l'intérieur de la zone membres de votre application :

```
public function logout()
{
    $this->redirect($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(Event $event)`. 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, `ÀuthComponent` 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 [109].

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 de traiter toutes les exceptions levées et de les manipuler.

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 :

```
// 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.

109. https://github.com/cakephp/acl

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', // app authorize object.
    'AuthBag.Combo', // plugin authorize object.
]);
```

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->config('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 (AuthComponnent) 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 publique. 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();

// 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()
    {
        parent::initialize();
        $this->loadComponent('Auth', [
            'authorize' => 'Controller',
        ]);
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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->config()` 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->config()` vous permet d'obtenir une valeur de configuration en appelant seulement l'option de configuration :

```
$this->Auth->config('loginAction');

$this->redirect($this->Auth->config('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 Protégées par AuthComponent* pour avoir des astuces sur la façon de tester les actions de controller qui sont protégées par `AuthComponent`.

CookieComponent

class `Cake\Controller\Component\`**`CookieComponent`**(*ComponentRegistry $collection*, *array $config = []*)

Le component Cookie est un conteneur de la méthode native de PHP `setcookie()`. Il simplifie la manipulation des cookies et chiffre automatiquement les données du cookie. Les cookies ajoutés via le CookieComponent seront seulement envoyés si l'action du controller se termine.

Obsolète depuis la version 3.5.0 : Vous devriez utiliser le encrypted-cookie-middleware à la place du `CookieComponent`.

Paramétrage des Cookies

Les cookies peuvent être configurés soit globalement, soit au niveau supérieur. Les données de configuration globale seront fusionnées avec la configuration de niveau supérieur. Donc vous devez simplement surcharger les parties qui sont différentes. Pour configurer les paramètres globaux, utilisez la méthode `config()` :

```
$this->Cookie->config('path', '/');
$this->Cookie->config([
    'expires' => '+10 days',
    'httpOnly' => true
]);
```

pour configurer une clé spécifique, utilisez la méthode `configKey()` :

```
$this->Cookie->config('User', 'path', '/');
$this->Cookie->configKey('User', [
    'expires' => '+10 days',
    'httpOnly' => true
]);
```

Il y a plusieurs valeurs de configuration pour les cookies :

expires Combien de temps les cookies doivent durer. Par défaut 1 mois.

path Le chemin sur le serveur web dans lequel le cookie sera disponible. Si le chemin est défini à "/foo/", le cookie sera seulement disponible dans le répertoire /foo/ et tous ses sous-répertoires comme /foo/bar/ du domaine. La valeur par défaut est le chemin de base de votre application.

domain Le domaine pour lequel le cookie est disponible. Pour rendre le cookie disponible sur tous les sous-domaines de example.com, définissez le domaine à ".example.com".

secure Indique que le cookie soit être transmis avec une connection sécurisée HTTPS. Quand il est défini à `true`, le cookie ne sera défini que si une connection sécurisée existe.

key La clé de chiffrement utilisé quand les cookies chiffrés sont activés. Par défaut à Security.salt.

httpOnly Défini à `true` pour ne faire que des cookies HTTP. Les Cookies qui sont HTTPOnly ne sont pas accessible en JavaScript. Par défaut à `false`.

encryption Le type de chiffrement à utiliser. Par défaut à "aes". Peut aussi être "rijndael" pour une compatibilité rétroactive.

Utiliser le Component

Le Component Cookie offre plusieurs méthodes pour travailler avec les Cookies.

`Cake\Controller\Component\CookieComponent::write`(*mixed $key*, *mixed $value = null*)

La méthode write() est le cœur du composant Cookie. $key est le nom de la variable désirée, et $value est l'information à stocker :

```
$this->Cookie->write('name', 'Larry');
```

Vous pouvez également grouper vos variables en utilisant la notation point "." dans les paramètres de clé :

```
$this->Cookie->write('User.name', 'Larry');
$this->Cookie->write('User.role', 'Lead');
```

Si vous voulez écrire plus d'une valeur dans le cookie en une fois, vous pouvez passer un tableau :

```
$this->Cookie->write('User',
    ['name' => 'Larry', 'role' => 'Lead']
);
```

Toutes les valeurs dans le cookie sont chiffrées avec AES par défaut. Si vous voulez stocker les valeurs en texte, assurez-vous de configurer l'espace de la clé :

```
$this->Cookie->configKey('User', 'encryption', false);
```

`Cake\Controller\Component\CookieComponent::read`(*mixed $key = null*)

Cette méthode est utilisée pour lire la valeur d'une variable de cookie avec le nom spécifié dans $key :

```
// Sortie "Larry"
echo $this->Cookie->read('name');

// Vous pouvez aussi utiliser la notation par point pour lire
```

(suite sur la page suivante)

(suite de la page précédente)

```
echo $this->Cookie->read('User.name');

// Pour récupérer les variables que vous aviez groupées en utilisant
// la notation par point comme tableau, faites quelque chose comme
$this->Cookie->read('User');

// ceci retourne quelque chose comme ['name' => 'Larry', 'role' => 'Lead']
```

Avertissement : CookieComponent ne peut pas intéragir avec les valeurs de chaînes vides qui contiennent `,`. Le component va tenter d'interpreter ces valeurs en tableaux, ce qui conduit à des résultats incorrects. A la place, vous devez utiliser `$request->getCookie()`.

`Cake\Controller\Component\CookieComponent::check`(*$key*)

Paramètres

— **$key** (*string*) – La clé à vérifier.

Utilisé pour vérifier si une clé/chemin existe et a une valeur non null.

`Cake\Controller\Component\CookieComponent::delete`(*mixed $key*)

Efface une variable de cookie dont le nom est défini dans $key. Fonctionne avec la notation par point :

```
// Efface une variable
$this->Cookie->delete('bar');

// Efface la variable bar du cookie, mais rien d'autre sous foo.
$this->Cookie->delete('foo.bar');
```

Cross Site Request Forgery

En activant le component CSRFComponent vous bénéficiez d'une protection contre les attaques CSRF [110] ou « Cross Site Request Forgery » qui est une vulnérabilité habituelle dans les applications web. Cela permet à un attaquant de capturer et rejouer une requête précédente, et parfois soumettre des données en utilisant des balises images ou des ressources sur d'autres domaines.

Le CsrfComponent fonctionne en installant un cookie sur le navigateur de l'utilisateur. Quand des formulaires sont créés à l'aide du *`Cake\View\Helper\FormHelper`*, un champ caché contenant un jeton CSRF est ajouté. Au cours de l'évènement `Controller.startup`, si la requête est de type POST, PUT, DELETE, PATCH, le component va comparer les données de la requête et la valeur du cookie. Si l'une des deux est manquantes ou que les deux valeurs ne correspondent pas, le component lancera une `Cake\Network\Exception\InvalidCsrfTokenException`.

Note : 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.

Nouveau dans la version 3.1 : Le type d'exception a changé de `Cake\Network\Exception\ForbiddenException` en `Cake\Network\Exception\InvalidCsrfTokenException`.

Obsolète depuis la version 3.5.0 : Vous devriez utiliser le csrf-middleware à la place du `CsrfComponent`.

110. http://fr.wikipedia.org/wiki/Cross-Site_Request_Forgery

Utiliser le CsrfComponent

En ajoutant simplement le `CsrfComponent` à votre tableau de components, vous pouvez profiter de la protection CSRF fournie :

```
public function initialize()
{
    parent::initialize();
    $this->loadComponent('Csrf');
}
```

Des réglages peuvent être transmis au composant par l'intermédiaire des paramètres de votre composant. Les options de configuration disponibles sont les suivants :

- `cookieName` Le nom du cookie à envoyer. Par défaut `csrfToken`.
- `expiry` Durée avant l'expiration du jeton CSRF. Session du navigateur par défaut.
- `secure` Si le cookie doit être créé avec Secure flag ou pas. Le cookie ne sera défini que sur une connection HTTPS et chaque tentative vers du HTTP standard échouera. Par défaut à `false`.
- `field` Le champ de formulaire à vérifier. Par défaut `_csrfToken`. Changer cette valeur nécessite également de configurer le FormHelper.

Lorsqu'il est activé, vous pouvez accéder au jeton CSRF actuel sur l'objet request :

```
$token = $this->request->getParam('_csrfToken');
```

Intégration avec le FormHelper

Le CsrfComponent s'intègre de façon transparente avec `FormHelper`. Chaque fois que vous créez un formulaire avec FormHelper, il va insérer un champ caché contenant le jeton CSRF.

Note : Lorsque vous utilisez le CsrfComponent vous devez toujours commencer vos formulaires avec le FormHelper. Si vous ne le faites pas, vous devrez créer manuellement les champs cachés dans chacun de vos formulaires.

Protection CSRF et Requêtes AJAX

En plus des paramètres de données de requête, les jetons CSRF peuvent être soumis par le biais d'un en-tête spécial `X-CSRF-Token`. Utiliser un en-tête facilite souvent l'intégration des jetons CSRF pour les applications Javascript lourdes ou pour les API basées sur XML/JSON.

Désactiver le Component CSRF pour des Actions Spécifiques

Bien que non recommandé, vous pouvez désactiver le CsrfComponent pour certaines requêtes. Vous pouvez réalisez ceci en utilisant le dispatcheur d'event du controller, au cours de la méthode `beforeFilter()` :

```
public function beforeFilter(Event $event)
{
    $this->eventManager()->off($this->Csrf);
}
```

FlashComponent

class Cake\Controller\Component**FlashComponent**(*ComponentCollection $collection*, *array $config = []*)

FlashComponent est un moyen de définir des messages de notifications à afficher après avoir envoyé un formulaire ou des données connus. 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 **src/Template/Element/Flash**. Par convention, les méthodes en camelcase vont être liées à un nom d'element en minuscule et avec des underscores (_) :

```
// Utilise src/Template/Element/Flash/success.ctp
$this->Flash->success('C\'était un succès');

// Utilise src/Template/Element/Flash/great_success.ctp
$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');
```

Nouveau dans la version 3.1 : Les messages Flash peuvent maintenant s'empiler. Des appels successifs à `set()` et `__call()` avec la même clé ajouteront les messages à `$_SESSION`. Si vous souhaitez conserver l'ancien comportement (un message malgré plusieurs appels successifs), définissez le paramètre `clear` à `true` quand vous configurez le Component.

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'element à utiliser pour le rendu.
- `params` Un tableau en option de clés/valeurs pour rendre disponible des variables dans un element.

Nouveau dans la version 3.1 : Une nouvelle option `clear` a été ajoutée. Elle doit être un `bool` et vous permet de supprimer tous les messages de la pile pour en commencer une nouvelle.

Un exemple de l'utilisation de ces options :

```
// Dans votre Controller
$this->Flash->success("L'utilisateur a été sauvegardé", [
    'key' => 'positive',
    'params' => [
        'name' => $user->name,
        'email' => $user->email
    ]
]);

// Dans votre Vue
<?= $this->Flash->render('positive') ?>

<!-- Dans src/Template/Element/Flash/success.ctp -->
```

(suite sur la page suivante)

(suite de la page précédente)

```
<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.ctp** dans **plugins/PluginName/src/Template/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

Nouveau dans la version 3.3.3.

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 offre une manière simple d'inclure 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 `View\Helper\FormHelper::create()`) et `View\Helper\FormHelper::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.

Nouveau dans la version cakephp/cakephp : 3.2.6

Depuis la version 3.2.6, un paramètre supplémentaire est inclus dans le callback blackHole, une instance de `Cake\Controller\Exception\SecurityException` est incluse dans le deuxième paramètre.

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\Event;

class WidgetsController extends AppController
{

    public function initialize()
    {
        parent::initialize();
        $this->loadComponent('Security');
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    public function beforeFilter(Event $event)
    {
        if ($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\Event;

class WidgetsController extends AppController
{

    public function initialize()
    {
        parent::initialize();
        $this->loadComponent('Security', ['blackHoleCallback' => 'forceSSL']);
    }

    public function beforeFilter(Event $event)
    {
        if ($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 *Cross Site Request Forgery*.

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\Event;

class WidgetController extends AppController
{

    public function initialize()
    {
        parent::initialize();
        $this->loadComponent('Security');
    }

    public function beforeFilter(Event $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`**

Les principaux défis lors de la création d'une application flexible et ergonomique sont le design et d'avoir une interface utilisateur intuitive. De nombreuses applications ont tendance à augmenter rapidement en taille et en complexité, et les designers ainsi que les programmeurs trouvent même qu'ils sont incapables de faire face a l'affichage de centaines ou de milliers d'enregistrements. Réécrire prend du temps, et les performances et la satisfaction des utilisateurs peut 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 rapide et facile pour paginer les données.

La pagination dans CakePHP se fait par un Component dans le controller, pour faciliter la création des requêtes de pagination. Dans la Vue, *View\Helper\PaginatorHelper* est utilisé pour faciliter la génération de la pagination, des liens et des boutons.

Utiliser Controller : :paginate()

Dans le controller, nous commençons par définir les conditions de la requête de pagination qui seront utilisées par défaut dans la variable `$paginate` du controller. Ces conditions, vont servir de base à vos requêtes de pagination. Elles sont complétées par les paramètres `sort`, `direction`, `limit` et `page` passés dans l'URL. Ici, il est important de noter que la clé `order` doit être définie dans une structure en tableau comme ci-dessous :

```
class ArticlesController extends AppController
{

    public $paginate = [
        'limit' => 25,
        'order' => [
            'Articles.title' => 'asc'
        ]
    ];

    public function initialize()
    {
        parent::initialize();
        $this->loadComponent('Paginator');
    }
}
```

Vous pouvez aussi inclure d'autres options *ORM\Table::find()*, comme `fields` :

```
class ArticlesController extends AppController
{

    public $paginate = [
        'fields' => ['Articles.id', 'Articles.created'],
        'limit' => 25,
        'order' => [
            'Articles.title' => 'asc'
        ]
    ];

    public function initialize()
    {
        parent::initialize();
        $this->loadComponent('Paginator');
    }
}
```

Alors que vous pouvez passer la plupart des options de query à partir de la propriété paginate, 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 = [
```

(suite sur la page suivante)

(suite de la page précédente)

```
        'finder' => 'published',
    ];
}
```

Comme les méthodes finder personnalisées peuvent aussi être prises en options, voici comment vous pouvez passer des options dans une méthode de finder personnalisée dans la propriété paginate :

```
class ArticlesController extends AppController
{

    // trouve les articles selon les tags
    public function tags()
    {
        $tags = $this->request->getParam('pass');

        $customFinderOptions = [
            'tags' => $tags
        ];
        // la méthode de finder personnalisée est appelée findTagged dans
        // ArticlesTable.php
        // elle devrait ressembler à ceci:
        // public function findTagged(Query $query, array $options) {
        // ainsi vous utilisez tagged en clé
        $this->paginate = [
            'finder' => [
                'tagged' => $customFinderOptions
            ]
        ];

        $articles = $this->paginate($this->Articles);

        $this->set(compact('articles', 'tags'));
    }
}
```

En plus de définir les valeurs de pagination générales, vous pouvez définir plus d'un jeu de pagination par défaut dans votre controller, vous avez juste à nommer les clés du tableau d'après le model que vous souhaitez configurer :

```
class ArticlesController extends AppController
{

    public $paginate = [
        'Articles' => [],
        'Authors' => [],
    ];
}
```

Les valeurs des clés `Articles` et `Authors` peuvent contenir toutes les propriétés qu'un model/clé sans `$paginate` peut contenir.

Une fois que la variable `$paginate` à été définie, nous pouvons utiliser la méthode *`Controller\Controller::paginate()`* pour créer les données paginées et ajouter le `PaginatorHelper` s'il n'a pas déjà été ajouté. La méthode paginate du controller va retourner l'ensemble des résultats de la requête paginée, et définir les meta-données de pagination de la requête. Vous pouvez accéder aux meta-données de pagination avec `$this->request->getParam('paging')`. un exemple plus complet de l'utilisation de `paginate()` serait :

```
class ArticlesController extends AppController
{

    public function index()
    {
        $this->set('articles', $this->paginate());
    }
}
```

Par défaut la méthode `paginate()` va utiliser le model par défaut pour un controller. Vous pouvez aussi passer la requête résultante d'une méthode find :

```
public function index()
{
   $query = $this->Articles->find('popular')->where(['author_id' => 1]);
   $this->set('articles', $this->paginate($query));
}
```

Si vous voulez paginer un model différent, vous pouvez lui fournir une requête l'objet table lui-même, ou son nom :

```
//Utiliser une query
$comments = $this->paginate($commentsTable->find());

// Utiliser le nom du model.
$comments = $this->paginate('Comments');

// Utiliser un objet table.
$comments = $this->paginate($commentTable);
```

Utiliser Directement Paginator

Si vous devez paginer des données d'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 duquel vous souhaitez paginer les résultats. En option, vous pouvez passer l'objet table et laisser la query être construite pour vous. Le second paramètre doit être le tableau des configurations 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. Il faut que vous passiez la query que vous souhaitez voir paginée.

Requêtes de Paginating Multiple

Vous pouvez paginer plusieurs models dans une unique action de controller en utilisant l'option `scope`, à la fois via la propriété `$paginate` d'un 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 elements HTML scopés et les URLS pour la pagination.

Nouveau dans la version 3.3.0 : Pagination multiple a été ajoutée dans la version 3.3.0

Contrôler les Champs Utilisés pour le Tri

Par défaut le tri peut être fait sur n'importe quelle colonne d'une table. Ceci n'est parfois pas souhaité puisque cela permet aux utilisateurs de trier sur des colonnes non indexées qui peuvent être compliquées à trier. Vous pouvez définir la liste blanche des champs qui peut être triée en utilisant l'option `sortWhitelist`. Cette option est nécessaire quand vous voulez trier sur des données associées, ou des champs calculés qui peuvent faire parti de la requête de pagination :

```
public $paginate = [
    'sortWhitelist' => [
        'id', 'title', 'Users.username', 'created'
    ]
];
```

Toute requête qui tente de trier les champs qui ne sont pas dans la liste blanche sera ignorée.

Limiter le Nombre Maximum de Lignes par Page

Le nombre de résultat qui sont récupérés et montrés à l'utilisateur est configuré par le paramètre `limit`. En général on ne souhaite pas permettre aux utilisateurs de récupérer toutes les lignes d'un ensemble paginé. L'option `maxLimit` permet à ce que personne ne puisse définir cette limite trop haute depuis l'extérieur. Par défaut, CakePHP limite le nombre maximum de lignes qui peuvent être récupérées à 100. Si cette valeur par défaut n'est pas approprié 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

Des associations supplémentaires peuvent être chargées à 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 Out of Range

PaginatorComponent va lancer une `NotFoundException` quand on essaie d'accéder à une page non existante, par exemple le nombre de page demandé est supérieur au total du nombre de pages.

Ainsi vous pouvez soit laisser s'afficher la page d'erreur normale, soit utiliser un bloc try catch et faire des actions appropriées quand une `NotFoundException` est attrapée :

```
// Prior to 3.6 use Cake\Network\Exception\NotFoundException
use Cake\Http\Exception\NotFoundException;

public function index()
{
    try {
        $this->paginate();
    } catch (NotFoundException $e) {
        // Faire quelque chose ici comme rediriger vers la première ou dernière page.
        // $this->request->getParam('paging') vous donnera les infos demandées.
    }
}
```

Pagination dans la Vue

Consultez la documentation *View\Helper\PaginatorHelper* pour savoir comment créer des liens de navigation paginés.

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 `Router::parseExtensions()`, 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()
    {
        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()
    {
        parent::initialize();
        $this->loadComponent('RequestHandler');
    }

    public function beforeFilter(Event $event)
    {
        if ($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()
    {
        parent::initialize();
        $parser = function ($data) {
            $rows = str_getcsv($data, "\n");
            foreach ($rows as &$row) {
                $row = str_getcsv($row, ',');
            }
            return $rows;
        };
        $this->loadComponent('RequestHandler', [
            'inputTypeMap' => [
                'csv' => [$parser]
            ]
        ]);
    }
}
```

Vous pouvez utiliser n'importe quel callback [111] 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->addInputType('json', ['json_decode', true]);

// Depuis 3.1.0, vous devez utiliser
$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`.

Obsolète depuis la version 3.1.0 : Depuis 3.1.0 la méthode `addInputType()` est dépréciée. Vous devez utiliser `config()` pour ajouter des types d'input à la volée.

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');
```

111. http://php.net/callback

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.

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()
{
    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()
{
    parent::initialize();
    $this->loadComponent('RequestHandler', [
        'viewClassMap' => [
            'json' => 'ApiKit.MyJson',
            'xml' => 'ApiKit.MyXml',
            'csv' => 'ApiKit.Csv'
        ]
    ]);
}
```

Obsolète depuis la version 3.1.0 : Depuis 3.1.0, la méthode `viewClassMap()` est dépréciée. Vous devez utiliser `config()` pour changer viewClassMap à la volée.

Configuration des Components

De nombreux components du cœur nécessitent une configuration. Quelques exemples : *Authentification* et *CookieComponent*. 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()
    {
        parent::initialize();
        $this->loadComponent('Auth', [
            'authorize' => ['controller'],
            'loginAction' => ['controller' => 'Users', 'action' => 'login']
        ]);
        $this->loadComponent('Cookie', ['expires' => '1 day']);
    }

}
```

Vous pouvez configurer les components à la volée en utilisant la méthode `config()`. Souvent, ceci est fait dans la méthode `beforeFilter()` de votre controller. Ceci peut aussi être exprimé comme ceci :

```
public function beforeFilter(Event $event)
{
    $this->Auth->config('authorize', ['controller']);
    $this->Auth->config('loginAction', ['controller' => 'Users', 'action' => 'login
↪']);

    $this->Cookie->config('name', 'CookieMonster');
}
```

Comme les helpers, les components ont une méthode `config()` qui est utilisée pour récupérer et définir toutes les configurations pour un component :

```
// Lire des données de config.
$this->Auth->config('loginAction');

// Définir la config
$this->Csrf->config('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 `config()`.

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()
    {
        parent::initialize('Auth', [
            'className' => 'MyAuth'
        ]);
    }
}

// src/Controller/Component/MyAuthComponent.php
use Cake\Controller\Component\AuthComponent;

class MyAuthComponent extends AuthComponent
{
    // 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 les actions 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 `initialize()` 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()
    {
        parent::initialize();
        $this->loadComponent('Flash');
    }

    public function delete()
    {
        if ($this->Post->delete($this->request->getData('Post.id')) {
            $this->Flash->success('Post deleted.');
            return $this->redirect(['action' => 'index']);
        }
    }
```

Note : Puisque les Models 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()
{
    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()
{
    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)
    {
        $this->Existing->foo();
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    public function bar()
    {
        // ...
   }
}

// src/Controller/Component/ExistingComponent.php
namespace App\Controller\Component;

use Cake\Controller\Component;

class ExistingComponent extends Component
{

    public function foo()
    {
        // ...
    }
}
```

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->_registry->getController();
```

Vous pouvez également accéder facilement au controller dans n'importe quel callback via l'objet event :

```
$controller = $event->getSubject());
```

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(*Event $event*)
Est appelée avant la méthode du controller beforeFilter, mais *après* la méthode initialize() du controller.

startup(*Event $event*)
Est appelée après la méthode du controller beforeFilter mais avant que le controller n'exécute l'action prévue.

beforeRender(*Event $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(*Event $event*)
Est appelée avant que la sortie soit envoyée au navigateur.

beforeRedirect(*Event $event*, *$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.

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
namespace App\View;

use Cake\View\View;

class AppView extends View
{

    public function initialize()
    {
        // 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 **.ctp** (CakePHP Template) et utilisent la syntaxe alternative de PHP [112] 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 echo $variable; ?>
```

Avec le support de « Short Tag » :

```
<?= $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` :

```
<ul>
<?php foreach ($todo as $item): ?>
<li><?= $item ?></li>
<?php endforeach; ?>
</ul>
```

112. http://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 [113], une sous-classe de View va faire le pont entre le langage du template et CakePHP.

Un fichier de template est stocké dans **src/Template/**, 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 **src/Template/Products/view.ctp**.

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 :

113. http://twig.sensiolabs.org

```
$this->set('activeMenuButton', 'posts');
```

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 :

```
<!-- src/Template/Common/view.ctp -->
<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 :

```
<!-- src/Template/Posts/view.ctp -->
<?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.ctp** 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');
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Assigner une chaine vide aura le même effet.
$this->assign('sidebar', '');
```

Nouveau dans la version 3.2 : View : :reset() a été ajoutée dans 3.2

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 src/Template/Layout/default.ctp
<?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]);
?>

// 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 **src/Template/Layout/default.ctp**. 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 **src/Template/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>
```

(suite sur la page suivante)

(suite de la page précédente)

```
<!-- Si vous voulez qu'un menu soit rendu pour toutes vos vues,
incluez le ici -->
<div id="header">
    <div id="menu">...</div>
</div>

<!-- 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 **src/Template/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 :

```
// A partir d'un controller
public function view()
{
    // Défini le layout
    $this->viewBuilder()->setLayout('admin');

    // Avant 3.4
    $this->viewBuilder()->layout('admin');

    // Avant 3.1
    $this->layout = 'admin';
}

// A 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;
```

(suite sur la page suivante)

(suite de la page précédente)

```
class UsersController extends AppController
{
    public function viewActive()
    {
        $this->set('title', 'View Active Users');
        $this->viewBuilder()->setLayout('default_small_ad');

        // ou ce qui suit avant 3.4
        $this->viewBuilder()->layout('default_small_ad');

        // ou ce qui suit avant 3.1
        $this->layout = 'default_small_ad';
    }

    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');
        // ou ce qui suit avant 3.1
        $this->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 **src/Template/Element/**, et ont une extension .ctp. 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, this text is very helpful."
]);
```

Dans le fichier element, toutes les variables passés 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 **src/Template/Element/helpbox.ctp** peut utiliser la variable `$helptext` :

```
// A l'intérieur de src/Template/Element/helpbox.ctp
echo $helptext; //outputs "Oh, this text is very helpful."
```

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']
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ]
);
```

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.ctp**), utilisez ce qui suit :

```
echo $this->element('Contacts.sidebar/helpbox');
```

Préfix de Routing et Elements

Nouveau dans la version 3.0.1.

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 **src/Template/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)
    {
        // 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 [114], 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 **src/Template/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;
```

(suite sur la page suivante)

114. https://github.com/apotonick/cells

(suite de la page précédente)

```
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 **src/Template/Cell/Inbox/display.ctp**. 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.

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()
    {
        $this->loadModel('Messages');
        $unread = $this->Messages->find('unread');
        $this->set('unread_count', $unread->count());
    }

}
```

Puisque les cells utilisent `ModelAwareTrait` et `ViewVarsTrait`, elles se comportent un peu comme un controller. Nous pouvons utiliser les méthodes `loadModel()` et `set()` un peu comme nous le ferions dans un controller. Dans notre fichier de template, ajoutons ce qui suit :

```
<!-- src/Template/Cell/Inbox/display.ctp -->
<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.ctp**.

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->template = '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)
    {
        $this->loadModel('Messages');

        // Création du paginator
        $paginator = new Paginator();

        // Pagination du model
        $results = $paginator->paginate(
            $this->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"*.

Nouveau dans la version 3.5.0 : `Cake\Datasource\Paginator` a été ajoutée dans 3.5.0.

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 facilite le changement du visuel et du ressenti de votre page rapidement. 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';
```

(suite sur la page suivante)

(suite de la page précédente)

```
    public function beforeRender(\Cake\Event\Event $event)
    {
        $this->viewBuilder()->setTheme('Modern');

        // 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 **src/Template/**.

Par exemple, le fichier de vue pour une action edit d'un controller Posts se trouvera dans **plugins/Modern/src/Template/Posts/edit.ctp**. Les fichiers de layout se trouveront dans **plugins/Modern/src/Template/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/src/Template/Plugin/Cms/Tags/edit.ctp** 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 **src/Template/**. 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 la clé `_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()
{
    ...
    $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 la Clé Serialize

La clé `_serialize` est une variable de vue spéciale qui 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()
    {
        parent::initialize();
        $this->loadComponent('RequestHandler');
    }

    public function index()
```

(suite sur la page suivante)

(suite de la page précédente)

```
    {
        // Défini les variables de vues qui doivent être sérialisées.
        $this->set('articles', $this->paginate());

        // Spécifie quelles variables de vues JsonView doit sérialiser.
        $this->set('_serialize', ['articles']);
    }
}
```

Vous pouvez aussi définir `_serialize` en tableau de variables de vue à combiner :

```
class ArticlesController extends AppController
{
    public function initialize()
    {
        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->set('_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.

Nouveau dans la version 3.1.0 : Dans cette version, la variable `_serialize` est maintenant automatiquement définie à `true` pour sérialiser toutes les variables de vue au lieu de devoir les spécifier explicitement.

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'));
    }
}
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Code de la vue - src/Template/Articles/json/index.ctp
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.

Note : Depuis 3.1.0, le AppController du squelette d'application ajoute automatiquement `'_serialize' => true` à toutes les requêtes XML/JSON. Vous devrez retirer ce code à partir du callback beforeRender ou définir `'_serialize' => false` dans l'action de votre controller si vous souhaitez utiliser les fichiers de vue.

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 la variable de vue `_rootNode`.

La classe XmlView intègre la variable `_xmlOptions` qui vous permet de personnaliser les options utilisées pour générer le XML, par exemple `tags` au lieu d'`attributes`.

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 [115] 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 :

```
// Dans l'action de votre controller, quand une sauvegarde échoue
$this->set('errors', $articles->errors());
$this->set('_jsonOptions', JSON_FORCE_OBJECT);
$this->set('_serialize', ['errors']);
```

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`.

115. http://php.net/json_encode

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;
// Prior to 3.6 use Cake\Network\Exception\NotFoundException
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->set('_serialize', ['videos']);

        // Définit le téléchargement forcé
        // Avant 3.4.0
        // $this->response->download('report-' . date('YmdHis') . '.' . $format);
        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 = []*)

Nouveau dans la version 3.3.6.

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([
    ['title' => 'Produits', 'url' => ['controller' => 'products', 'action' => 'index
↪']],
    ['title' => 'Nom du produit', 'url' => ['controller' => 'products', 'action' =>
↪'view', 1234]]
```

(suite sur la page suivante)

(suite de la page précédente)

```
]);

// 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(
    ['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 `templates()` du `StringTemplateTrait` :

```
$this->Breadcrumbs->templates([
    '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->templates([
    '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' => [
```

(suite sur la page suivante)

(suite de la page précédente)

```
            '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);
```

Nouveau dans la version 3.4.0 : La méthode `reset()` a été ajoutée dans la version 3.4.0

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 **src/Template/Element/Flash**. Vous remarquerez que le template de l'App de CakePHP est livré avec trois éléments flash : **success.ctp**, **default.ctp** et **error.ctp**.

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'
```

(suite sur la page suivante)

(suite de la page précédente)

```
]);

// Dans votre View: Va utiliser great_success.ctp au lieu de succcess.ctp
<?= $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.

Nouveau dans la version 3.1 : Le *FlashComponent* empile maintenant les messages. Si vous définissez plusieurs messages, lors d'un appel à `render()`, chaque message sera rendu dans son élément, dans l'ordre dans lequel les messages ont été définis.

Pour plus d'informations sur le tableau d'options disponibles, consultez la section *FlashComponent*.

Préfixe de Routage et Messages Flash

Nouveau dans la version 3.0.1.

Si vous avez configuré un préfixe de Routage, vous pouvez maintenant stocker vos elements de messages Flash dans **src/Template/{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 **src/Template/Element/Flash/error.ctp**, 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 [116] ; ce contexte permet au FormHelper de fonctionner avec les retours de l'ORM.
- Un tableau contenant la clé `schema`, qui mappe vers ArrayContext [117] ce qui vous permet de créer des structures simples de données pour construire des formulaires.
- `null` et `false` mappe vers NullContext [118] ; 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.

Toutes les classes de contexte ont aussi un accès aux données requêtées, facilitant la construction des formulaires.

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 rendant la construction de formulaires simples. Vous pouvez fermer le contexte actif en utilisant la méthode `end()`, ou en appelant `create()` à nouveau. Pour créer un formulaire pour une entity, faîtes 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 :

116. https://api.cakephp.org/3.x/class-Cake.View.Form.EntityContext.html
117. https://api.cakephp.org/3.x/class-Cake.View.Form.ArrayContext.html
118. https://api.cakephp.org/3.x/class-Cake.View.Form.NullContext.html

```
// src/Controller/ArticlesController.php:
public function edit($id = null)
{
    if (empty($id)) {
        throw new NotFoundException;
    }
    $article = $this->Articles->get($id);
    // Logique d'enregistrement
    $this->set('article', $article);
}

// View/Articles/edit.ctp:
// Si $article->isNew() est false, nous aurons un formulaire d'édition
<?= $this->Form->create($article) ?>
```

Affichera :

```
<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.

Options for Form Creation

Le tableau `$options` est là où la configuration du formulaire se passe. Ce tableau spécial peut contenir un certain nombre de paires de clé-valeur différentes qui affectent la façon dont la balise form est générée. Voici les valeurs autorisés :

- `'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'` - Définira le verbe HTTP à, respectivement, PUT, DELETE ou PATCH 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`.

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 la query string

Nouveau dans la version 3.4.0.

Les sources de valeurs du FormHelper définissent d'où les éléments du formulaire reçoivent leurs valeurs.

Par défaut, Formhelper récupère ses valeurs depuis le « context ». Les contextes par défaut, comme le `EntityContext`, récupèrera ses valeurs depuis l'entité qui lui est attribuée ou dans `$request->data`.

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, [
    'valueSources' => ['query', 'context']
]);

// Même effet :
echo $this->Form
    ->setValueSources(['query', 'context'])
    ->create($articles);

// Lecture des valeurs seulement dans la query string
echo $this->Form->create($article);
$this->Form->setValueSources('query');

// Même effet :
echo $this->Form->create($article, ['valueSources' => 'query']);
```

Les sources supportées sont `context`, `data` et `query`. Vous pouvez utiliser une ou plusieurs sources. Tous les widgets générés par le `FormHelper` iront récupérer leurs valeurs dans les sources spécifiées, dans l'ordre dans lequel vous les avez définies.

Les sources définies seront réinitialisées à leur valeur par défaut (`['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 `login()` du controller courant, vous pouvez fournir le tableau `$options` comme ce qui suit :

```
echo $this->Form->create($article, ['url' => ['action' => 'login']]);
```

Affichera :

```
<form method="post" action="/users/login">
```

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' => 'http://www.google.com/search',
    'type' => 'get'
]);
```

Affichera :

```
<form method="get" action="http://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.

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 intégrer Cake\View\Form\ContextInterface [119] . Une fois que vous avez intégré cette interface, vous pouvez connecter votre nouveau contexte dans le FormHelper. Il est souvent mieux de faire ceci 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($request, $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, retourne null.

Création d'éléments de Formulaire

`Cake\View\Helper\FormHelper::control`(*string $fieldName*, *array $options = []*)

- `$fieldName` - Nom du champ (attribut `name`) de l'élément dans le formulaire (exemple : `'Modelname.fieldname'`).
- `$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 laisse générer des inputs de formulaire. Ces inputs incluent 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.

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 n'importe quel type de balise `input` ainsi que tous les autres types d'éléments HTML de formulaire (`select`, `button`, `textarea`)

119. https://api.cakephp.org/3.x/class-Cake.View.Form.ContextInterface.html

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 day, month, et year selects

datetime, timestamp day, month, year, hour, minute, et meridian selects

time hour, minute, et meridian selects

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 *`View\Helper\FormHelper::submit()`* ou définir `'novalidate' => true` dans les options pour *`View\Helper\FormHelper::create()`*.

Par exemple, supposons que votre model User intègre les champs pour un *username* (varchar), *password* (varchar), *approved* (datetime) and *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 des champs Jour, mois, année, heure, minute
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_dt', [
    'label' => 'Date de naissance',
    'minYear' => date('Y') - 70,
    'maxYear' => date('Y') - 18,
]);
```

Outre les *Options pour la méthode control()* pour `control()` vu ci-dessus, vous pouvez spécifier n'importe quelle option des méthodes spécifiques pour le type d'input 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'));
```

Ensuite, ajouter les lignes suivantes à votre template de vue de 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'));
```

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 (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 *`View\Helper\FormHelper::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 les 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 dans des 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 facilite la sauvegarde des données avec 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 sont triées.

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 (et les autres type d'éléments comme les `checkbox` ou les `textarea`), comme les attributs HTML. Ce qui suit va couvrir les options spécifiques de `FormHelper::control()`.

— `$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 utilisé comme valeur pour l'élément HTML `<label>` ou bien un tableau *d'options pour le label*. 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>
```

(suite sur la page suivante)

(suite de la page précédente)

```
    <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 vous passez un tableau, vous pourrez fournir des options supplémentaires pour l'element `label`. Si vous le faîtes, 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'
    ]
]);
```

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 passez à cette option un tableau contenant les choix pour les éléments comme les `radio` et les `select`. Reportez vous à *Créer des Boutons Radio* et *Créer des Select* pour plus de détails. Les défaut est `null`.

— `$options['error']` Utiliser cette clé vous permettra de transformer les messages de model par défaut 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 dans vos models. De plus, vous pouvez fournir des messages i18n pour vos formulaires.

— `$options['nestedInput']` - Utilisez avec les inputs `checkbox` et `radio`. Cette option permet de contrôler si les éléments `input` doivent être générés dans 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`.
Cependant, 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"*. En fonction du `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 ou bien 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` à 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 *`View\Helper\FormHelper::label()`* et *`View\Helper\FormHelper::error()`* pour générer des layouts (mise en page) complètements personnalisés.

Options Communes

Beaucoup des différentes méthodes d'input supportent un jeu d'options communes. Toutes ses 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). Une valeur par défaut explicite va surcharger toute valeur définie par défaut dans le schéma.
Exemple d'utilisation :

```
echo $this->Form->text('ingredient', ['default' => 'Sugar']);
```

Exemple avec un champ sélectionné (Taille « Medium » sera sélectionné 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 chekbox - 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. A 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 la source de valeurs, vous devrez définir `value` en `''` (au lieu de le définir avec `null`).

En plus des options ci-dessus, vous pouvez mixer n'importe quel attribut HTML que vous souhaitez utiliser. Tout nom d'option non-special sera traité comme un attribut HTML, et appliqué à l'élément HTML input généré. NdT. celui qui capte cette phrase gagne un giroTermoOnduleur à double convection.

Modifié dans la version 3.3.0 : Depuis la version 3.3.0, FormHelper va automatiquement utiliser les valeurs par défaut définies dans le schéma de votre base de données. Vous pouvez désactiver ce comportement en définissant l'option `schemaDefault` à `false`.

Créer des Elements 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. Toutefois, dans ce cas, $options est utilisé avant tout pour spécifier les attributs des balises HTML (comme la valeur ou l'id DOM d'un élément du formulaire).

Créer des Inputs Text

`Cake\View\Helper\FormHelper::text`(*string $name*, *array $options*)

- `$name` - Le `name` du champ sous la forme `'Modelname.fieldname'`.
- `$options` - Un tableau optionnel d'options avec n'importe laquelle *des options générales* ainsi que n'importe quel attribut HTML valide.

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*)

- `$name` - Le `name` du champ sous la forme `'Modelname.fieldname'`.
- `$options` - Un tableau optionnel d'options avec n'importe laquelle *des options générales* ainsi que n'importe quel attribut HTML valide.

Création d'un input `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*)

- `$name` - Le `name` du champ sous la forme `'Modelname.fieldname'`.
- `$options` - Un tableau optionnel d'options avec n'importe laquelle *des options générales* ainsi que n'importe quel attribut HTML valide.

Créera un input `hidden` de form. Exemple :

```
echo $this->Form->hidden('id');
```

Affichera :

```
<input name="id" value="10" type="hidden" />
```

Créer des Textareas

`Cake\View\Helper\FormHelper::textarea` (*string $fieldName*, *array $options*)

- `$name` - Le `name` du champ sous la forme `'Modelname.fieldname'`.
- `$options` - Un tableau optionnel d'options avec n'importe laquelle *des options générales* ainsi que n'importe quel attribut HTML valide.

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 form est édité (ainsi, le tableau `$this->request->getData()` va contenir les informations sauvegardées pour le model `User`), la valeur correspondant au champs `notes` sera automatiquement ajoutée au HTML généré. Exemple :

```
<textarea name="notes" id="notes">
    Ce Texte va être édité.
</textarea>
```

Options for Textarea

En plus *des options générales*, `textarea()` supporte 2 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 sont, respectivement, le nombre de lignes et de colonnes :

  ```
  echo $this->Form->textarea('textarea', ['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 options et des points en communs, c'est pourquoi ils sont regroupés dans section.

Les Options pour Select, Checkbox et Inputs Radio

Vous trouverez ci-dessous les options partagées entre `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 la valeur qui sera définit quand le formulaire sera rendu (dans ces cas là, `'value'` doit avoir une valeur qui existe dans l'élément). Elle peut 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 vous passe cette option `radio()`, cela créera un élément `input` supplémentaire qui sera affiché avant le premier bouton radio, avec une valeur de `''` et un `label` qui vaudra la chaîne passée dans l'option.
  - Si vous la passez à la méthode `select`, cela créer un élément `option` vide avec une valeur vide dans la liste de 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="">(choose one)</option>
    <option value="0">1</option>
    <option value="1">2</option>
    <option value="2">3</option>
    <option value="3">4</option>
```

(suite sur la page suivante)

(suite de la page précédente)

```
    <option value="4">5</option>
</select>
```

— `hiddenField` Pour certain types d'input (checkboxes, radios) un input caché est créé. 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 en place à différents endroits c'est seulement le dernier groupe de valeur d'input qui sera sauvegardé.
Dans cet exemple , seules les couleurs tertiaires seront passées, et les couleurs primaires seront réécrites :

```
<h2>Primary Colors</h2>
<input type="hidden" name="color" value="0" />
<label for="color-red">
    <input type="checkbox" name="color[]" value="5" id="color-red" />
    Red
</label>

<label for="color-blue">
    <input type="checkbox" name="color[]" value="5" id="color-blue" />
    Blue
</label>

<label for="color-yellow">
    <input type="checkbox" name="color[]" value="5" id="color-yellow" />
    Green
</label>

<h2>Tertiary Colors</h2>
<input type="hidden" name="color" value="0" />
<label for="color-green">
    <input type="checkbox" name="color[]" value="5" id="color-green" />
    Yellow
</label>
<label for="color-purple">
    <input type="checkbox" name="color[]" value="5" id="color-purple" />
    Purple
</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 ce comportement.
Vous pouvez définir une valeur différente pour le champ caché autre que 0 comme "N" :

```
echo $this->Form->checkbox('published', [
    'value' => 'Y',
    'hiddenField' => 'N',
]);
```

Créer des Checkboxes

`Cake\View\Helper\FormHelper::checkbox`(*string $fieldName*, *array $options*)

- `$name` - Le `name` du champ sous la forme `'Modelname.fieldname'`.
- `$options` - Un tableau optionnel d'options avec n'importe laquelle *des options générales*, des options de la section *Les Options pour Select, Checkbox et Inputs Radio*, des options spécifiques aux checkbox (ci-dessous) ainsi que n'importe quel attribut HTML valide.

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'` - Booléen. Si passé à `true`, la checkbox aura l'attribut `disabled`.

Cette méthode génère également un input de type `hidden` pour forcer l'existence de la donnée dans le tableau de POST.

E.g.

```
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*)

- `$name` - Le `name` du champ sous la forme `'Modelname.fieldname'`.
- `$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 le input `hidden` (si `'hiddenField'` vaut `true`) ou pas d'élément du tout (si `'hiddenField'` vaut `false`).
- `$attributes` - Un tableau optionnel d'options avec n'importe laquelle *des options générales*, des options de la section *Les Options pour Select, Checkbox et Inputs Radio*, des options spécifiques aux radios (ci-dessous) ainsi que n'importe quel attribut HTML valide.

Crée un jeu d'inputs 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. 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 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 les label pour les boutons radio via l'argument `$options`.

Par exemple :

```
$this->Form->radio('gender', ['Masculine','Feminine','Neuter']);
```

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` est une simple paire 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;'],
        ['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>
```

Créer des Select

`Cake\View\Helper\FormHelper::select` (*string $fieldName*, *array $options*, *array $attributes*)

- `$name` - Le `name` du champ sous la forme `'Modelname.fieldname'`.
- `$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 d'options avec n'importe laquelle *des options générales*, des options de la section *Les Options pour Select, Checkbox et Inputs Radio*, des options spécifiques aux select (ci-dessous) ainsi que n'importe quel attribut HTML valide.

Crée un élément `select`, rempli des éléments compris dans `$options`. Si l'option `$attributes['value']` est fournie, alors les éléments `option` ayant cette valeur 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 utilisez les templates suivant :

```
'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 entités HTML seront convertis). 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 de 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 élément `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>
```

(suite sur la page suivante)

(suite de la page précédente)

```
    </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 Attributs

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' => 'Male', 'F' => 'Female'];
echo $this->Form->select('gender', $options, ['empty' => true]);
```

Affichera :

```
<select name="gender">
    <option value=""></option>
    <option value="M">Male</option>
    <option value="F">Female</option>
</select>
```

— `'escape'` - Booleén. Si passée à `true`, le contenu des `option` sera échappé (les entités HTML seront encodées).
Par exemple

```
// This will prevent HTML-encoding the contents of each option element
$options = ['M' => 'Male', 'F' => 'Female'];
echo $this->Form->select('gender', $options, ['escape' => false]);
```

— `'multiple'` - Si définie à `true`, cette option rendra le `select` multiple.
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'
```

(suite sur la page suivante)

(suite de la page précédente)

```
];
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>
</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' => 'Masculine',
    'F' => 'Feminine',
    'N' => 'Neuter'
];
echo $this->Form->select('gender', $options, [
    'disabled' => ['M', 'N']
]);
```

Affichera :

```
<select name="gender">
    <option value="M" disabled="disabled">Masculine</option>
    <option value="F">Feminine</option>
    <option value="N" disabled="disabled">Neuter</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">
```

(suite sur la page suivante)

(suite de la page précédente)

```
        <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*)

- `$name` - Le `name` du champ sous la forme `'Modelname.fieldname'`.
- `$options` - Un tableau optionnel d'options avec n'importe laquelle *des options générales* ainsi que n'importe quel attribut HTML valide.

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 placer des valeurs par défauts dans les champs inputs de type "file". A chaque fois que le formulaire sera affiché, la valeur sera vide.

Lors de la soumission, le champ file fournit un tableau étendu de données au script recevant les données de formulaire.

Pour l'exemple ci-dessus, les valeurs dans le tableau de données soumis devraient être organisées comme ci-dessous, si CakePHP à été installé sur un server Windows (la clé `tmp_name` aura un chemin différent dans un environnement Unix) :

```
$this->request->data['submittedfile'] = [
    'name' => 'conference_schedule.pdf',
    'type' => 'application/pdf',
```

(suite sur la page suivante)

(suite de la page précédente)

```
    'tmp_name' => 'C:/WINDOWS/TEMP/php1EE.tmp',
    'error' => 0, // Peut être une chaine sur Windows.
    'size' => 41737,
];
```

Ce tableau est généré par PHP lui-même, pour plus de détails sur la façon dont PHP gère les données passées a travers les champs `files`, lire la section file uploads du manuel de PHP [120].

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()`.

Creating Date & Time Related Controls

Les Options communes pour les éléments Date et Time

These options are common for the date and time related controls :

- `'empty'` - Si à `true`, un élément `option` vide sera ajouté dans le `select` en début de liste. Si vous fournissez une chaîne, elle sera utilisé comme texte de l'`option`. Défaut à `true`.
- `'default'` | `value` - Utilisez l'une ou l'autre de ces options pour définir la valeur qui sera affiché pour le champ. Une valeur présente dans `$this->request->getData()` avec en clé le nom du champ écrasera cette valeur. Si aucune valeur par défaut n'est fournie, `time()` sera utilisé.
- `'year', 'month', 'day', 'hour', 'minute', 'second', 'meridian'` - Ces options vous permettent de contrôler quels éléments sont générés ou non. En définissant une de ces options à `false`, vous pouvez désactiver la génération du `select` correspondant (s'il est normalement rendu par la méthode appelée). En plus de ce comportement, vous pouvez également passer des attributs HTML pour les éléments en question.

Options pour les éléments Date

Ces options sont liées aux méthodes liées aux dates, c'est-à-dire `year()`, `month()`, `day()`, `dateTime()` et `date()` :

- `'monthNames'` - Si à `false`, un nombre à deux chiffres sera utilisé pour afficher les mois plutôt que le nom des mois. Si vous passez un tableau (`['01' => 'Jan', '02' => 'Feb', ...]`), le tableau passé sera utilisé à la place.
- `'minYear'` - Année minimum à utiliser pour le `select` qui correspond à l'année
- `'maxYear'` - Année maximum à utiliser pour le `select` qui correspond à l'année
- `'orderYear'` - L'ordre d'affichage des années dans `select` qui correspond à l'année. Les valeurs possibles sont `'asc'` et `'desc'`. Défaut à `'desc'`.

120. http://php.net/features.file-upload

Options pour les éléments Time

Ces options sont liées aux méthodes liées à l'heure - `hour()`, `minute()`, `second()`, `dateTime()` et `time()` :

- `'interval'` - L'intervale en minute entre les valeurs affichées dans le `select` qui correspond aux minutes. Défaut à 1.
- `'round'` - Définir à `up` ou `down` si vous voulez forcer les minutes à être arrondie dans l'une ou l'autre des direction (au supérieur ou à l'inférieur) si la valeur ne correspond pas à l'intervale défini. Défaut à `null`.
- `timeFormat` - Applicable à `dateTime()` et `time()`. Le format d'heure à utiliser pour le `select` des heures : `12` ou `24`. Si vous passez autre chose que `24`, le format `12` sera utilisé par défaut et le `select` correspondant au `meridian` sera affiché automatiquement à côté du `select` des secondes. Défaut à 24.
- `format` - S'applique à la méthode `hour()` : soit `12` soit `24`. Si vous la définissez à `12`, le `select` correspondant au `meridian` ne sera pas automatiquement affiché. Défaut à `24`.
- `second` - Applicable à `dateTime()` and `time()`. Définir à `true` pour activer le `select` correspondant aux secondes. Défaut à `false`.

Créer des champs DateTime

`Cake\View\Helper\FormHelper::`**`dateTime`**(*$fieldName*, *$options = []*)

- `$fieldName` - Une chaîne qui sera utilisé comme préfixe pour l'attribut `name` des `select`.
- `$options` - Un tableau optionnel d'options avec n'importe laquelle *des options générales*, les options spécifiques (vu ci-dessus) ainsi que n'importe quel attribut HTML valide.

Crée un ensemble d'élément `select` qui permettent de définir une date et une heure.

Pour contrôler l'ordre des éléments ainsi qu'ajouter des éléments ou du contenu entre les différents éléments, vous pouvez surcharger le template `dateWidget`. Par défaut, le template a cette forme :

```
{{year}}{{month}}{{day}}{{hour}}{{minute}}{{second}}{{meridian}}
```

Appeler cette méthode sans options générera, par défaut, 5 `select` : année (4 chiffres), mois (textuelle complète), jour (numérique), heure (numérique), minutes (numérique).

Par exemple :

```
<?= $this->form->dateTime('registered') ?>
```

Affichera :

```
<select name="registered[year]">
    <option value="" selected="selected"></option>
    <option value="2022">2022</option>
    ...
    <option value="2012">2012</option>
</select>
<select name="registered[month]">
    <option value="" selected="selected"></option>
    <option value="01">January</option>
    ...
    <option value="12">December</option>
</select>
<select name="registered[day]">
    <option value="" selected="selected"></option>
    <option value="01">1</option>
    ...
    <option value="31">31</option>
```

(suite sur la page suivante)

(suite de la page précédente)

```
</select>
<select name="registered[hour]">
    <option value="" selected="selected"></option>
    <option value="00">0</option>
    ...
    <option value="23">23</option>
</select>
<select name="registered[minute]">
    <option value="" selected="selected"></option>
    <option value="00">00</option>
    ...
    <option value="59">59</option>
</select>
```

Pour créer des éléments avec des classes et des attributs spécifiques sur un élément donné, vous pouvez passer un tableau de paramètres pour chaque élément, via l'argument `$options`.

Par exemple :

```
echo $this->Form->dateTime('released', [
    'year' => [
        'class' => 'year-classname',
    ],
    'month' => [
        'class' => 'month-class',
        'data-type' => 'month',
    ],
]);
```

Ce qui créera les 2 éléments suivant :

```
<select name="released[year]" class="year-class">
    <option value="" selected="selected"></option>
    <option value="00">0</option>
    <option value="01">1</option>
    <!-- .. snipped for brevity .. -->
</select>
<select name="released[month]" class="month-class" data-type="month">
    <option value="" selected="selected"></option>
    <option value="01">January</option>
    <!-- .. snipped for brevity .. -->
</select>
```

Créer des éléments Date

`Cake\View\Helper\FormHelper::date`(*$fieldName*, *$options = []*)

- `$fieldName` - Une chaîne qui sera utilisé comme préfixe pour l'attribut `name` des `select`.
- `$options` - Un tableau optionnel d'options avec n'importe laquelle *des options générales*, les options spécifiques (vu ci-dessus) ainsi que n'importe quel attribut HTML valide.

Va créer, par défaut, 3 `select` : un pour l'année (4 chiffres), un pour le mois (forme textuelle complète) et un pour le jour (numérique).

Vous pouvez contrôler les éléments `select` en passant des tableaux de d'options.

Par exemple :

```
// En partant du principe que l'année en cours est 2017, cet appel va
// désactiver le select pour le jour, retirer l'option vide pour le select
// de l'année, limiter l'année minimum à 2018, ajouter des attributs HTML
// à l'année, ajouter une chaîne pour l'option vide des mois et changer les
// mois pour qu'il soit afficher sous forme de chiffres
<?php
    echo $this->Form->date('registered', [
        'minYear' => 2018,
        'monthNames' => false,
        'empty' => [
            'year' => false,
            'month' => 'Choisissez un mois...'
        ],
        'day' => false,
        'year' => [
            'class' => 'cool-years',
            'title' => 'Année d'inscription'
        ]
    ]);
?>
```

Affichera :

```
<select class= "cool-years" name="registered[year]" title="Année d'inscription">
    <option value="2022">2022</option>
    <option value="2021">2021</option>
    ...
    <option value="2018">2018</option>
</select>
<select name="registered[month]">
    <option value="" selected="selected">Choisissez un mois...</option>
    <option value="01">1</option>
    ...
    <option value="12">12</option>
</select>
```

Créer des éléments Time

`Cake\View\Helper\FormHelper::time` (*$fieldName*, *$options = []*)

- `$fieldName` - Une chaîne qui sera utilisé comme préfixe pour l'attribut `name` des `select`.
- `$options` - Un tableau optionnel d'options avec n'importe laquelle *des options générales*, les options spécifiques (vu ci-dessus) ainsi que n'importe quel attribut HTML valide.

Va créer, par défaut, 2 `select` : un pour l'heure (sous forme 24 heures) et un pour le temps (sous forme 60 minutes).

Par exemple, pour créer un `select` qui propose les minutes par tranche de 15 minutes et appliquer une classe aux `select`, vous pouvez utiliser l'appel suivant :

```
echo $this->Form->time('released', [
    'interval' => 15,
    'hour' => [
        'class' => 'foo-class',
    ],
    'minute' => [
        'class' => 'bar-class',
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ],
]);
```

Ce qui générera le code suivant :

```
<select name="released[hour]" class="foo-class">
    <option value="" selected="selected"></option>
    <option value="00">0</option>
    <option value="01">1</option>
    <!-- .. snipped for brevity .. -->
<option value="22">22</option>
<option value="23">23</option>
</select>
<select name="released[minute]" class="bar-class">
    <option value="" selected="selected"></option>
    <option value="00">00</option>
    <option value="15">15</option>
    <option value="30">30</option>
    <option value="45">45</option>
</select>
```

Créer des éléments Année (Year)

`Cake\View\Helper\FormHelper::year` (*string $fieldName*, *array $options = []*)

- `$fieldName` - Une chaîne qui sera utilisé comme préfixe pour l'attribut `name` des `select`.
- `$options` - Un tableau optionnel d'options avec n'importe laquelle *des options générales*, les options spécifiques (vu ci-dessus) ainsi que n'importe quel attribut HTML valide.

Crée un élément `select` qui contiendra une option par année pour les années situées entre `minYear` et `maxYear` si ces options sont fournies ou pour les années entre -5 et +5 par rapport à l'année du jour. 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('purchased', [
    'minYear' => 2000,
    'maxYear' => date('Y')
]);
```

Si nous sommes en 2009, nous obtiendrons :

```
<select name="purchased[year]">
    <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 des éléments Mois (Month)

`Cake\View\Helper\FormHelper::month(string $fieldName, array $attributes)`

- `$fieldName` - Une chaîne qui sera utilisé comme préfixe pour l'attribut `name` des `select`.
- `$options` - Un tableau optionnel d'options avec n'importe laquelle *des options générales*, les options spécifiques (vu ci-dessus) ainsi que n'importe quel attribut HTML valide.

Crée un `select` avec les mois sous forme textuelle.

Par exemple :

```
echo $this->Form->month('mob');
```

Affichera :

```
<select name="mob[month]">
    <option value=""></option>
    <option value="01">January</option>
    <option value="02">February</option>
    <option value="03">March</option>
    <option value="04">April</option>
    <option value="05">May</option>
    <option value="06">June</option>
    <option value="07">July</option>
    <option value="08">August</option>
    <option value="09">September</option>
    <option value="10">October</option>
    <option value="11">November</option>
    <option value="12">December</option>
</select>
```

Vous pouvez passer votre propre tableau de valeurs en utilisant l'option `'monthNames'` ou bien avoir les mois afficher sous leur forme chiffrée en passant `false`.

Par exemple

```
echo $this->Form->month('mob', ['monthNames' => false]);
```

Note : Les mois par défaut peuvent être traduits via les fonctionnalités d'internationalisation de CakePHP. Vous reportez à la section *Internationalisation & Localisation* pour plus d'informations.

Créer des éléments Jour (Day)

`Cake\View\Helper\FormHelper::day(string $fieldName, array $attributes)`

- `$fieldName` - Une chaîne qui sera utilisé comme préfixe pour l'attribut `name` des `select`.
- `$options` - Un tableau optionnel d'options avec n'importe laquelle *des options générales*, les options spécifiques (vu ci-dessus) ainsi que n'importe quel attribut HTML valide.

Crée un `select` avec les jours du mois sous forme numérique.

Pour créer un `option` vide affichant le texte de votre choix (pour qu'à l'initialisation la première option soit « Jour » par exemple), vous pouvez définir le texte souhaité dans le paramètre `'empty'`.

Par exemple :

```
echo $this->Form->day('created', ['empty' => 'Jour']);
```

Affichera :

```
<select name="created[day]">
    <option value="" selected="selected">Jour</option>
    <option value="01">1</option>
    <option value="02">2</option>
    <option value="03">3</option>
    ...
    <option value="31">31</option>
</select>
```

Créer des éléments Heure (Hour)

`Cake\View\Helper\FormHelper::hour`(*string $fieldName*, *array $attributes*)

- `$fieldName` - Une chaîne qui sera utilisé comme préfixe pour l'attribut `name` des `select`.
- `$options` - Un tableau optionnel d'options avec n'importe laquelle *des options générales*, les options spécifiques (vu ci-dessus) ainsi que n'importe quel attribut HTML valide.

Créer un `select` avec les heures du jour.

Vous pouvez avoir un `select` au format 12 ou 24 heures en utilisant l'option `'format'` :

```
echo $this->Form->hour('created', [
    'format' => 12
]);
echo $this->Form->hour('created', [
    'format' => 24
]);
```

Créer des éléments Minute

`Cake\View\Helper\FormHelper::minute`(*string $fieldName*, *array $attributes*)

- `$fieldName` - Une chaîne qui sera utilisé comme préfixe pour l'attribut `name` des `select`.
- `$options` - Un tableau optionnel d'options avec n'importe laquelle *des options générales*, les options spécifiques (vu ci-dessus) ainsi que n'importe quel attribut HTML valide.

Crée un `select` avec les valeurs des minutes pour l'heure. VOus pouvez créer un `select` qui contient seulement des valeurs spécifiques en utilisant l'option `'interval'`.

Par exemple si vous voulez des paliers de 10 minutes, vous utiliseriez le code suivant :

```
echo $this->Form->minute('arrival', [
    'interval' => 10
]);
```

Affichera :

```
<select name="arrival[minute]">
    <option value="" selected="selected"></option>
    <option value="00">00</option>
    <option value="10">10</option>
```

(suite sur la page suivante)

(suite de la page précédente)

```
    <option value="20">20</option>
    <option value="30">30</option>
    <option value="40">40</option>
    <option value="50">50</option>
</select>
```

Creating Meridian Controls

`Cake\View\Helper\FormHelper::meridian`(*string $fieldName*, *array $attributes*)

- `$fieldName` - Une chaîne qui sera utilisé comme préfixe pour l'attribut `name` des `select`.
- `$options` - Un tableau optionnel d'options avec n'importe laquelle *des options générales*, les options spécifiques (vu ci-dessus) ainsi que n'importe quel attribut HTML valide.

Crée un `select` avec les valeurs "am" et "pm". Utile si vous utilisez le format d'heure `12` car il vous permettra de préciser la période de la journée à laquelle cette heure appartient

Créer les Labels

`Cake\View\Helper\FormHelper::label`(*string $fieldName*, *string $text*, *array $options*)

- `$fieldName` - Le `name` du champ sous la forme `'Modelname.fieldname'`.
- `$text` - Chaîne optionnelle pour définir le texte du label.
- `$options` - Optionel. Chaîne ou tableau qui peut contenir n'importe laquelle *des options générales* ainsi que n'importe quel attribut HTML valide.

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('User.name');
echo $this->Form->label('User.name', 'Your username');
```

Affichera :

```
<label for="user-name">Name</label>
<label for="user-name">Your username</label>
```

Avec le troisième paramètre `$options`, vous pouvez définir un id ou une classe :

```
echo $this->Form->label('User.name', null, ['id' => 'user-label']);
echo $this->Form->label('User.name', 'Your username', ['class' => 'highlight']);
```

Affichera :

```
<label for="user-name" id="user-label">Name</label>
<label for="user-name" class="highlight">Your username</label>
```

Afficher et vérifier les erreurs

FormHelper expose quelques méthodes qui vous permette de facilement vérifier si vos champs contiennent des erreurs ou pour afficher des messages d'erreur personnalisés.

Afficher les Erreurs

`Cake\View\Helper\FormHelper::error`(*string $fieldName*, *mixed $text*, *array $options*)

- `$fieldName` - Le `name` du champ sous la forme `'Modelname.fieldname'`.
- `$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 paire 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, spécifiée par `$text`, pour le champ donné, dans le cas où une erreur de validation a eu lieu. Si `$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 :

```
// Dans TicketsTable vous avez une règle de validation 'notEmpty' :
public function validationDefault(Validator $validator)
{
    $validator
        ->requirePresence('ticket', 'create')
        ->notEmpty('ticket');
}

// Dans Templates/Tickets/add.ctp 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 *`View\Helper\FormHelper::control()`*, les erreurs sont rendues par défaut, donc vous n'aurez pas besoin d'utiliser `isFieldError()` ou d'appeler `error()` manuellement.

Vérifier la Présence d'Erreurs

`Cake\View\Helper\FormHelper::isFieldError(string $fieldName)`

— `$fieldName` - Un nom de champ sous la forme `'Modelname.fieldname'`.

Retourne `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');
}
```

Note : En utilisant *`View\Helper\FormHelper::control()`*, les erreurs sont retournées par défaut.

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. Chaîne ou tableau qui peut contenir n'importe laquelle *des options générales* ainsi que n'importe quel attribut HTML valide.

Crée un input `submit` avec le texte `$caption`. Si la `$caption` fournie est l'URL d'une image (sous-entendu que la valeur fournie contient " ://" ou que son extension soit ".jpg, .jpe, .jpeg, .gif"), un bouton submit de l'image sera généré (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) pour le formulaire. Défaut à `'submit'`.
— `'templateVars'` - Utilisez ce tableau pour fournir des templates de variables supplémentaire pour l'élément et / ou ses conteneurs.
— Tout autre paramètre sera considéré comme un attribut à l'élément HTML `input`.

Le code suivant :

```
echo $this->Form->submit('Click me');
```

Affichera :

```
<div class="submit"><input value="Click me" type="submit"></div>
```

Vous pouvez aussi passer une URL relative ou absolue vers une image au paramètre caption au lieu d'un caption text :

```
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 Elements Button

`Cake\View\Helper\FormHelper::button`(*string $title*, *array $options = []*)

- `$title` - Chaîne obligatoire qui permet de fournir le texte du bouton.
- `$options` - Optionel. Chaîne ou tableau qui peut contenir n'importe laquelle *des options générales* ainsi que n'importe quel attribut HTML valide.

Crée un bouton HTML avec le titre spécifié et un type par défaut `button`. Définir

Options pour les Button

- `$options['type']` - Définissez cette variable à l'une des trois valeurs suivantes :
  1. `'submit'` - Comme pour la méthode `$this->Form->submit()`, cela créera un bouton de type `submit`. Notez cependant que ça 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 formulairere.
  3. `'button'` - Crée un bouton standard.
- `$options['escape']` - Booléen. Si cette option est définie à `true`, le contenu HTML de la valeur fournie pour `$title` sera échappé. Défaut à `false`.

Par exemple :

```
echo $this->Form->button('A Button');
echo $this->Form->button('Another Button', ['type' => 'button']);
echo $this->Form->button('Reset the Form', ['type' => 'reset']);
echo $this->Form->button('Submit Form', ['type' => 'submit']);
```

Affichera :

```
<button type="submit">A Button</button>
<button type="button">Another Button</button>
<button type="reset">Reset the Form</button>
<button type="submit">Submit Form</button>
```

Exemple en utilisant l'option `escape` :

```
// Va afficher le HTML echappé.
echo $this->Form->button('<em>Submit Form</em>', [
    'type' => 'submit',
    'escape' => true
]);
```

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 `SecurityComponent`.

La méthode `end()` ferme et complète le marquage du formulaire. Souvent, `end()` affiche juste la base fermante du formulaire, mais l'utilisation de `end()` permet également au FormHelper d'ajouter les champs cachées dont le component Security `Cake\Controller\Component\SecurityComponent` a besoin :

```
<?= $this->Form->create(); ?>

<!-- Elements de formulaire -->

<?= $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\SecurityComponent` 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 = []*)

— `$title` - Chaîne obligatoire qui sera utilisé comme texte du bouton. Notez que, par défaut, cette valeur ne sera pas échappée.
— `$url` - URL cible du formulaire, sous forme de chaîne ou de tableau.
— `$options` - Optionel. Chaîne ou tableau qui peut contenir n'importe laquelle *des options générales*, les options spécifiques (ci-dessous) ainsi que n'importe quel attribut HTML valide.

Crée une balise `<button>` avec un `<form>` l'entourant qui soumet une requête POST. De plus, par défaut, cela générera des inputs `hidden` pour le `SecurityComponent`.

Options for POST Button

— `'data'` - Tableau clé / valeur à passer aux inputs `hidden`.
— `'method'` - La méthode de la requête à utiliser. Par exemple si vous voulez que la requête émise simule 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.ctp
<?= $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>
```

Cette méthode crée un élément `<form>`. Donc n'utilisez pas cette méthode dans 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 = []*)

— `$title` - Chaîne obligatoire qui sera utilisé comme texte du bouton. Notez que, par défaut, cette valeur ne sera pas échappée.
— `$url` - URL cible du formulaire, sous forme de chaîne ou de tableau.
— `$options` - Optionel. Chaîne ou tableau qui peut contenir n'importe laquelle *des options générales*, les options spécifiques (ci-dessous) ainsi que n'importe quel attribut HTML valide.

Crée un lien HTML, mais accède à l'Url en utilisant la méthode POST (par défaut). Requiert que JavaScript soit autorisé dans votre navigateur.

Options pour les liens POST

— `'data'` - Tableau clé / valeur à passer aux inputs `hidden`.
— `'method'` - La méthode de la requête à utiliser. Par exemple si vous voulez que la requête émise simule 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 en un *view block* qui peut être affiché en dehors du formulaire principal.

Si vous souhaitez plutôt 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 des *Utiliser les Blocks de Vues*

Personnaliser les Templates que FormHelper Utilise

Comme beaucoup de helpers dans CakePHP, FormHelper utilise les string templates pour mettre en forme le HTML qu'il crée. Alors que les templates par défaut sont destinés à être un ensemble raisonnable de valeurs par défaut, vous aurez peut-être besoin de personnaliser les templates pour correspondre à votre application.

Pour changer les templates quand le helper est chargé, 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 dans **config/app_form.php**. Le 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 définissez vont remplacer ceux 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);
// Avant 3.4
$this->Form->templates($myTemplates);
```

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>`

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 [121].

Utiliser des conteneurs personnalisés distincts pour les éléments

En plus de ces templates, la méthode `control()` va essayer d'utiliser les templates 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('User.email_notifications', [
    'options' => ['y', 'n'],
    'type' => 'radio'
]);
```

Utiliser des groupe de formulaire (formGroup) 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 (formGroup). Un group de formulaire est un ensemble un label & input. Par exemple, lorsque vous créez des inputs de type radio, le template `radioFormGroup` sera utilisé s'il est présent. Si ce template est manquant, par défaut chaque ensemble `label` & `input` sera généré en utilisant le template `formGroup` :

```
// Ajoute un groupe de formulaire pour radio personnalisé
$this->Form->setTemplates([
    'radioFormGroup' => '<div class="radio">{{label}}{{input}}</div>'
]);
```

Ajouter des Variables de Template Supplémentaires aux Templates

Vous pouvez aussi ajouter des placeholders de template supplémentaires dans des templates personnalisés et remplir ces placeholders lors de la génération des inputs.

Par exemple :

```
// Ajoute un template avec le placeholder help.
$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', [
```

(suite sur la page suivante)

121. https://api.cakephp.org/3.x/class-Cake.View.Helper.FormHelper.html#%24_defaultConfig

(suite de la page précédente)

```
    '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>
```

Nouveau dans la version 3.1 : L'option templateVars a été ajoutée dans 3.1.0

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 framework 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`(*mixed $fields = [], $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 utilisé pour personnalisé 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 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 additionnelles 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']
    ],
    null,
    ['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 utilisé pour personnalisé 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 `$fields` est égal par défaut à *tous* les champs de l'entity de niveau supérieur actuelle. Pour exclure des champs spécifiques de la liste d'inputs générées, définissez les à `false` dans le paramètre `$fields` :

```
echo $this->Form->allControls(['password' => false]);
// Avant 3.4.0:
echo $this->Form->allInputs(['password' => false]);
```

Si vous désactiver le `fieldset`, la `legend` ne s'affichera pas.

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');

// Tags inputs (belongsToMany)
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');

// Select multiple pour belongsToMany
echo $this->Form->control('tags._ids', [
    'type' => 'select',
    'multiple' => true,
    'options' => $tagList,
]);

// 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'
    ]
```

(suite sur la page suivante)

(suite de la page précédente)

```
]);
```

Ajouter des Widgets Personnalisés

CakePHP permet d'ajouter des widgets personnalisés dans votre application, afin de 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

Les classes Widget ont une interface requise vraiment simple. Elles doivent implémenter la `Cake\View\Widget\WidgetInterface`. Cette interface nécessite que les méthodes `render(array $data)` et `secureFields(array $data)` soient implémentées. 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 voulez construire un widget Autocomplete, vous pouvez le faire comme ceci :

```
namespace App\View\Widget;

use Cake\View\Form\ContextInterface;
use Cake\View\Widget\WidgetInterface;

class AutocompleteWidget implements WidgetInterface
{

    protected $_templates;

    public function __construct($templates)
    {
        $this->_templates = $templates;
    }

    public function render(array $data, ContextInterface $context)
    {
        $data += [
            'name' => '',
        ];
        return $this->_templates->format('autocomplete', [
            'name' => $data['name'],
            'attrs' => $this->_templates->formatAttributes($data, ['name'])
        ]);
    }

    public function secureFields(array $data)
    {
        return [$data['name']];
    }
}
```

Évidemment, c'est un exemple très simple, mais il montre comment développer un widget personnalisé. Ce widget ferait un rendu du template « autocomplete », si défini 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 que FormHelper Utilise*.

Utiliser les Widgets

Vous pouvez charger des widgets personnalisés lors du chargement du FormHelper ou en utilisant la méthode `addWidget()`. Lors du changement 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' => [
            'App\View\Widget\AutocompleteWidget',
            'text',
            'label'
        ]
    ]
]);
```

Dans l'exemple ci-dessus, le widget `autocomplete` widget 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 au noms `text` et `label` lui sont passés. Ajouter des widgets en utilisant la méthode `addWidget` resemble à ceci :

```
// Utilise une classname.
$this->Form->addWidget(
    'autocomplete',
    ['Autocomplete', 'text', 'label']
);

// Utilise une instance - nécessite de 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 `control()` le fait toujours. Sinon vous pouvez juste créer un widget input en utilisant la méthode magique :

```
echo $this->Form->autocomplete('search', $options);
```

Travailler avec SecurityComponent

`Cake\Controller\Component\SecurityComponent` offre plusieurs fonctionnalités qui rendent vos formulaires plus sûrs et plus sécurisés. En incluant simplement le `SecurityComponent` dans votre controller, vous bénéficierez automatiquement des fonctionnalités de prévention contre la falsification de formulaires.

Tel que mentionné précédemment, lorsque vous utilisez le SecurityComponent, vous devez toujours fermer vos formulaires en utilisant *View\Helper\FormHelper::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 hashage de `SecurityComponent`. Cela autorise également 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 du formulaire ou une chaîne si la sécurisation des formulaire n'est pas utilisée. Si l'option `$secureAttributes` est définie, ces attributs HTML seront utilisés avec ceux générés par le SecurityComponent. Cela vous permet de définir des attributs HTML spécifique en fonction de vos besoins.

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"
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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'
);
?>
// 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
    <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('http://code.jquery.com/jquery.min.js');
```

Affichera :

```
<script src="http://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` :

```
echo $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`(*$code*, *$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`(*$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>
```

(suite sur la page suivante)

(suite de la page précédente)

```
                </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'],
    ['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
```

(suite sur la page suivante)

(suite de la page précédente)

```
    </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 *`View\Helper\HtmlHelper::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 *`View\Helper\HtmlHelper::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 *`View\Helper\HtmlHelper::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 `View\Helper\HtmlHelper::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' => '¥ ',
```

(suite sur la page suivante)

(suite de la page précédente)

```
    '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,
    '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()
{
    ...
    $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()
{
    ...
    $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]);
```

Nouveau dans la version 3.4.0 : Les options `first` et `last` ont été ajoutées dans la version 3.4.0

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.

Nouveau dans la version 3.4.0.

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`.

Nouveau dans la version 3.5.0 : La méthode `limitControl()` a été ajoutée dans 3.5.0

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 [122] 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 :

```
<!-- src/Template/Posts/index.ctp -->
<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 :

122. https://api.cakephp.org/3.x/class-Cake.View.Helper.PaginatorHelper.html

```
<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() ?>
```

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.

Nouveau dans la version 3.3.0 : La pagination multiple a été ajoutée dans la version 3.3.0

Rss

class Cake\View\Helper**RssHelper**(*View $view*, *array $config = []*)

Le RssHelper permet de générer facilement le XML pour les flux RSS [123].

Obsolète depuis la version 3.5.0 : Le RssHelper est déprécié à partir de 3.5.0 et sera supprimé dans 4.0.0

Créer un flux RSS avec RssHelper

Cet exemple suppose que vous ayez un Controller Articles, une Table Articles et une Entity Article déjà créés et que vous voulez créer une vue alternative pour les flux RSS.

Créer une version XML/RSS de `articles/index` est vraiment simple avec CakePHP. Après quelques étapes faciles, vous pouvez tout simplement ajouter l'extension .rss demandée à `articles/index` pour en faire votre URL `articles/index.rss`. Avant d'aller plus loin en essayant d'initialiser et de lancer notre service Web, nous avons besoin de faire un certain nombre de choses. Premièrement, le parsing d'extensions doit être activé dans **config/routes.php** :

```
Router::extensions('rss');
```

Dans l'appel ci-dessus, nous avons activé l'extension .rss. Quand vous utilisez *Cake\Routing\Router::extensions()*, vous pouvez passer une chaine de caractères ou un tableau d'extensions en tant que premier argument. Cela activera chaque extension/content-type utilisée dans votre application. Maintenant, quand l'adresse `posts/index.rss` est demandée, vous obtiendrez une version XML de votre `posts/index`. Cependant, nous devons éditer le controller pour y ajouter le code dédié au rss.

Code du Controller

C'est une bonne idée d'ajouter RequestHandler dans la méthode `initialize()` de votre controller Posts. Cela permettra beaucoup d'automagie :

```
public function initialize()
{
    parent::initialize();
    $this->loadComponent('RequestHandler');
}
```

Avant que nous puissions faire une version RSS de notre `articles/index`, nous avons besoin de mettre certaines choses en ordre. Il pourrait être tentant de mettre les métadonnées du canal dans l'action du controller et de le passer à votre vue en utilisant la méthode *Cake\Controller\Controller::set()*, mais ceci est inapproprié. Cette information pourrait également aller dans la vue. Cela arrivera sans doute plus tard, mais pour l'instant, si vous avez un ensemble de logique différent entre les données utilisées pour créer le flux RSS et les données pour la page HTML, vous pouvez utiliser la méthode `Cake\Controller\Component\RequestHandler::isRss()`, sinon votre controller pourrait rester le même :

```
// Modifie l'action du Controller Posts correspondant à
// l'action qui délivre le flux rss, laquelle est
// l'action index dans notre exemple

public function index()
{
    if ($this->RequestHandler->isRss() ) {
```

(suite sur la page suivante)

123. https://en.wikipedia.org/wiki/RSS

(suite de la page précédente)

```
        $articles = $this->Articles
            ->find()
            ->limit(20)
            ->order(['created' => 'desc']);
        $this->set(compact('articles'));
    } else {
        // ceci n'est pas une requête RSS
        // donc on retourne les données utilisées par l'interface du site web
        $this->paginate = [
            'order' => ['created' => 'desc'],
            'limit' => 10
        ];
        $this->set('articles', $this->paginate($this->Articles));
        $this->set('_serialize', ['articles']);
    }
}
```

Maintenant que toutes ces variables de Vue sont définies, nous avons besoin de créer un layout rss.

Layout

Un layout Rss est très simple, mettez le contenu suivant dans **src/Template/Layout/rss/default.ctp** :

```
if (!isset($documentData)) {
    $documentData = [];
}
if (!isset($channelData)) {
    $channelData = [];
}
if (!isset($channelData['title'])) {
    $channelData['title'] = $this->fetch('title');
}
$channel = $this->Rss->channel([], $channelData, $this->fetch('content'));
echo $this->Rss->document($documentData, $channel);
```

Il ne ressemble pas à plus mais grâce à la puissance du `RssHelper` il fait beaucoup pour améliorer le visuel pour nous. Nous n'avons pas défini `$documentData` ou `$channelData` dans le controller, cependant dans CakePHP vos vues peuvent passer des variables au layout. C'est l'endroit où notre tableau `$channelData` va venir définir toutes les données meta de notre flux.

Ensuite il y a le fichier de vue pour mes articles/index. Un peu comme le fichier de layout que nous avons créé, nous devons créer un répertoire **src/Template/Posts/rss/** et un nouveau fichier **index.ctp** à l'intérieur de celui-ci. Les contenus du fichier sont ci-dessous.

View

Notre vue, localisée dans **src/Template/Posts/rss/index.ctp**, commence par définir les variables `$documentData` et `$channelData` pour le layout, celles-ci contiennent toutes les metadonnées pour notre flux RSS. Cela est réalisé via la méthode *`Cake\View\View::set()`* qui est analogue à la méthode *`Cake\Controller\Controller::set()`*. Ici nous passons les données des canaux en retour au layout :

```
$this->set('channelData', [
    'title' => __("Most Recent Posts"),
```

(suite sur la page suivante)

(suite de la page précédente)

```
    'link' => $this->Url->build('/', true),
    'description' => __("Most recent posts."),
    'language' => 'en-us'
]);
```

La seconde partie de la vue génère les éléments pour les enregistrements actuels du flux. Ceci est accompli en bouclant sur les données qui ont été passées à la vue ($items) et en utilisant la méthode `RssHelper::item()`. L'autre méthode que vous pouvez utiliser `RssHelper::items()` qui prend un callback et un tableau des items pour le flux. La méthode de callback est généralement appelée `transformRss()`. Il y a un problème avec cette méthode, qui est qu'elle n'utilise aucune des classes de helper pour préparer vos données à l'intérieur de la méthode de callback parce que la portée à l'intérieur de la méthode n'inclut pas tout ce qui n'est pas passé à l'intérieur, ainsi ne donne pas accès au TimeHelper ou à tout autre helper dont vous auriez besoin. `RssHelper::item()` transforme le tableau associatif en un élément pour chaque paire de valeur de clé.

Note : Vous devrez modifier la variable $link comme il se doit pour votre application. Vous pourriez également utiliser une *propriété virtuelle* dans votre Entity.

```
foreach ($articles as $article) {
    $created = strtotime($article->created);

    $link = [
        'controller' => 'Articles',
        'action' => 'view',
        'year' => date('Y', $created),
        'month' => date('m', $created),
        'day' => date('d', $created),
        'slug' => $article->slug
    ];

    // Retire & échappe tout HTML pour être sûr que le contenu va être validé.
    $body = h(strip_tags($article->body));
    $body = $this->Text->truncate($body, 400, [
        'ending' => '...',
        'exact'  => true,
        'html'   => true,
    ]);

    echo  $this->Rss->item([], [
        'title' => $article->title,
        'link' => $link,
        'guid' => ['url' => $link, 'isPermaLink' => 'true'],
        'description' => $body,
        'pubDate' => $article->created
    ]);
}
```

Vous pouvez voir ci-dessus que nous pouvons utiliser la boucle pour préparer les données devant être transformées en éléments XML. Il est important de filtrer tout caractère de texte non brute en-dehors de la description, spécialement si vous utilisez un éditeur de texte riche pour le corps de votre blog. Dans le code ci-dessus nous utilisons `strip_tags()` et *`h()`* pour retirer/échapper tout caractère spécial XML du contenu, puisqu'ils peuvent entraîner des erreurs de validation. Une fois que nous avons défini les données pour le flux, nous pouvons ensuite utiliser la méthode `RssHelper::item()` pour créer le XML dans le format RSS. Une fois que vous avez toutes ces configurations, vous pouvez tester votre flux RSS en allant à votre `/posts/index.rss` et que vous verrez votre nouveau flux. Il est toujours important que vous validiez votre flux RSS avant de le mettre en live. Ceci peut être fait en visitant

les sites qui valident le XML comme Le Validateur de flux ou le site de w3c à http://validator.w3.org/feed/.

Note : Vous aurez besoin de définir la valeur de "debug" dans votre configuration du cœur à `false` pour obtenir un flux valide, à cause des différentes informations de debug ajoutées automatiquement sous des paramètres de debug plus haut qui cassent la syntaxe XML ou les règles de validation du flux.

Session

class `Cake\View\Helper\`**`SessionHelper`**(*View $view*, *array $config = []*)

Le SessionHelper offre la majorité des fonctionnalités de l'objet Session et les rend disponible dans votre vue.

La grande différence entre l'objet Session et le SessionHelper est que ce dernier *ne peut pas* écrire dans la session.

Comme pour le Component Session, les données sont écrites et lues en utilisant des structures de tableaux avec la *notation avec points*, comme ci-dessous :

```
['User' =>
    ['username' => 'super@example.com']
];
```

Étant donné ce tableau, le nœud sera accessible par `User.username`, le point indiquant le tableau imbriqué. Cette notation est utilisée pour toutes les méthodes du SessionHelper où une variable `$key` est utilisée.

`Cake\View\Helper\SessionHelper::`**`read`**(*string $key*)

Type renvoyé mixed

Lire à partir de la Session. Retourne une chaîne de caractère ou un tableau dépendant des contenus de la session.

`Cake\View\Helper\SessionHelper::`**`check`**(*string $key*)

Type renvoyé boolean

Vérifie si une clé est dans la Session. Retourne un booléen sur l'existence d'un clé.

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 la double ligne retourne et <br> où une simple ligne retourne, sont trouvés :

```
$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
    ]
);
```

(suite sur la page suivante)

(suite de la page précédente)

```
// 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
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ]
);
```

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 facile 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,
  null,
  $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() [124].

Url

class `Cake\View\Helper\`**`UrlHelper`**(*View $view*, *array $config = []*)

UrlHelper vous permet de générer facilement 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.

124. https://api.cakephp.org/3.x/class-Cake.I18n.Time.html#_i18nFormat

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 `full` 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
```

URL (commençant par "/") avec le chemin complet :

```
echo $this->Url->build('/posts', 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(
```

(suite sur la page suivante)

(suite de la page précédente)

```
//      '/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
]);
```

Nouveau dans la version 3.3.5 : `build()` accepte un tableau comme 2ème argument à partir de3.3.5

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');

// Affiche /css/app.css
$this->Url->css('app.css');
```

Nouveau dans la version 3.2.4 : Les méthodes de helper d'asset ont été ajoutées dans la version 3.2.4.

Pour de plus amples informations, voir Router::url [125] 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()
    {
        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 :

125. https://api.cakephp.org/3.x/class-Cake.Routing.Router.html#_url

```
$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()
    {
        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 :

```
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
{

    // Le hook initialize() est disponible depuis 3.2. Pour les versions
    // précédentes, vous pouvez surcharger le constructeur si nécessaire.
    public function initialize(array $config)
    {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        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
{

    use StringTemplateTrait;

    protected $_defaultConfig = [
        'errorClass' => 'error',
        '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 errorClass.
$class = $this->Awesome->config('errorClass');
```

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)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    {
        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()
    {
        $this->loadHelper('Html', [
            'className' => 'MyHtml'
        ]);
    }
}

// src/View/Helper/MyHtmlHelper.php
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;
```

(suite sur la page suivante)

(suite de la page précédente)

```
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()
    {
        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']
```

(suite sur la page suivante)

(suite de la page précédente)

```
        );
    }
}
```

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 des 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*.

Note : Si vous êtes familier avec les versions précédentes de CakePHP, vous devriez lire le *Guide de Migration du Nouvel ORM* pour voir les différences importantes entre CakePHP 3.0 et les versions antérieures de CakePHP.

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`, vous pourriez faire :

```
use Cake\ORM\TableRegistry;

// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->get('Articles');

$query = $articles->find();

foreach ($query as $row) {
    echo $row->title;
}
```

Notez que nous n'avons créé aucun code ou généré aucune configuration. Les conventions dans CakePHP nous permettent d'éviter un code bancal et permettent au framework d'insérer des classes de base lorsque votre application n'a pas créé de classe concrète. Si nous voulions personnaliser notre classe ArticlesTable en ajoutant des associations ou en définissant des méthodes supplémentaires, nous ajouterions ce qui suit dans **src/Model/Table/ArticlesTable.php** après la balise d'ouverture `<?php` :

```
namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{

}
```

Les classes de Table utilisent la version CamelCased du nom de la table avec le suffixe `Table` en nom de classe. Une fois que votre classe a été créée vous obtenez une référence vers celle-ci en utilisant `ORM\TableRegistry` comme avant :

```
use Cake\ORM\TableRegistry;

// Maintenant $articles est une instance de notre classe ArticlesTable.
// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->get('Articles');
```

Maintenant que nous avons une classe de table concrète, nous allons probablement vouloir utiliser une classe entity concrète. Les classes Entity vous laissent définir les méthodes accesseurs et mutateurs, définissant la logique personnalisée pour des enregistrements individuels et plus encore. Nous commencerons par ajouter ce qui suit 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\TableRegistry;

// Maintenant une instance de ArticlesTable.
// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->get('Articles');
$query = $articles->find();

foreach ($query as $row) {
    // Chaque ligne est maintenant une instance de notre classe Article.
    echo $row->title;
}
```

CakePHP utilise des conventions de nommage pour lier ensemble 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 *Entities* pour plus d'informations sur la façon d'utiliser les objets table et les entities dans votre application.

Pour en savoir plus sur les Models

Notions de Base de Base de Données

La couche d'accès à la base de données de CakePHP fournit une abstraction et aide avec la plupart des aspects du traitement des 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 les requêtes envoyées à la base de données.

Tour Rapide

Les fonctions décrites dans ce chapitre illustrent les possibilités de l'API d'accès à la base de données de plus bas niveau. 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::config('default', ['url' => $dsn]);
```

Une fois créé, vous pouvez accéder à l'objet Connection pour commencer à l'utiliser :

```
$connection = ConnectionManager::get('default');
```

Bases de Données Supportées

CakePHP supporte les serveurs de base de données relationnelles suivants :
- MySQL 5.1+
- SQLite 3
- PostgreSQL 8.3+
- SQLServer 2008+
- Oracle (avec un plugin créé par la communauté)

Pour chacun des drivers de base de données ci-dessus, assurez-vous d'avoir la bonne extension PDO installée. Les API procédurales ne sont pas supportées.

La base de données Oracle est supportée via le Driver pour les bases de données Oracle [126] du plugin créé par la communauté.

Exécuter des Instructions Select

Exécuter une instruction SQL pur 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 instructions 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 :

```
$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');
```

126. https://github.com/CakeDC/cakephp-oracle-driver

Exécuter des Instructions Insert

Insérer une ligne dans une base de données est habituellement l'affaire de quelques lignes :

```
use Cake\Datasource\ConnectionManager;

$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 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**. Un exemple d'information sur la configuration ressemblera 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' => 'utf8',
        'timezone' => 'UTC',
        'cacheMetadata' => true,
    ]
],
```

Ce qui est au-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::config()`. Un exemple de ceci serait :

```
use Cake\DataSource\ConnectionManager;

ConnectionManager::config('default', [
    'className' => 'Cake\Database\Connection',
    'driver' => 'Cake\Database\Driver\Mysql',
    'persistent' => false,
    'host' => 'localhost',
    'username' => 'my_app',
    'password' => 'secret',
    'database' => 'my_app',
    'encoding' => 'utf8',
    'timezone' => 'UTC',
    'cacheMetadata' => true,
]);
```

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* :

```
ConnectionManager::config('default', [
    'url' => 'mysql://my_app:secret@localhost/my_app?encoding=utf8&timezone=UTC&
↪cacheMetadata=true',
]);
```

Lorsque vous utilisez une chaine 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*.

Il y a un certain nombre de clés supportées dans la configuration de la base de données. Voici la liste complète :

className Nom de classe complète (incluant le *namespace*) de la classe qui représente une connexion au serveur de base de données. Cette classe a pour rôle de charger le driver 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 driver utilisée pour implémenter les spécificités du moteur de base de données. Cela peut être soit un nom de classe court en utilisant la *syntaxe de plugin*, un nom complet en namespace, soit être une instance de driver 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. A partir de CakePHP 3.4.13, 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. Comme cela complique l'identifier quoting, CakePHP ne supporte pas `.` dans les noms de base de données. 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. Si vous souhaitez utiliser l'encodage UTF-8 avec les connexions MySQL, vous devez utiliser "utf8" sans trait d'union.

timezone La définition du 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 du fichier vers la clé du fichier SSL. (supporté seulement par MySQL).

ssl_cert Le chemin du fichier vers le fichier du certificat SSL. (supporté seulement par MySQL).

ssl_ca Le chemin du fichier vers 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 les mots réservés ou les caractères spéciaux avec les noms de tables ou de colonnes. Activer cette configuration va entraîner la construction des requêtes en utilisant le *Query Builder* avec les identifiers quotés lors de la création de SQL. 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 driver que vous utilisez.

cacheMetadata Soit un booléen `true`, soit une chaîne contenant la configuration du cache pour stocker les meta données. Désactiver la mise en cache des metadata n'est pas conseillé et peut entraîner de faibles performances. Consultez la section sur *La Mise en Cache de Metadata* pour plus d'information.

mask Définit les droits sur le fichier de base de données généré (seulement supporté par SQLite)

Au point où nous sommes, vous pouvez aller voir *Conventions de CakePHP*. Le nommage correct pour vos tables (et pour quelques colonnes) peut vous offrir des fonctionnalités gratuites et vous aider à éviter 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.

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;

$conn = 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 `config()` et `get()` vous pouvez créer à tout moment de nouvelles connexions qui ne sont pas défines dans votre fichier de configuration :

```
ConnectionManager::config('my_connection', $config);
$conn = ConnectionManager::get('my_connection');
```

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\`**`Type`**

Puisque tous les fournisseurs de base de données n'intègrent pas la même définition des types de données, ou les mêmes noms pour des types de données similaires, CakePHP fournit un ensemble de types de données abstraites à utiliser avec la couche de la base de données. Les types supportés par CakePHP sont :

string Généralement construit en colonnes CHAR ou VARCHAR. Utiliser l'option `fixed` va forcer une colonne CHAR. Dans SQL Server, les types NCHAR et NVARCHAR sont utilisés.

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).

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`.

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 DATE sans timezone. La valeur de retour de ce type de colonne est `Cake\I18n\Date` qui étend la classe native `DateTime`.

datetime Correspond au type de colonne DATETIME sans timezone. Dans PostgreSQL et SQL Server, ceci retourne un type TIMESTAMP. La valeur retournée par défaut de ce type de colonne est *`Cake\I18n\Time`* qui étend les classes intégrées `DateTime` et Chronos [127].

timestamp Correspond au type TIMESTAMP.

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. Le type "json" a été ajouté dans la version 3.3.0.

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 le type hints fourni lorsque les requêtes sont faites. Par exemple une colonne qui est marquée en "datetime" va automatiquement convertir les paramètres d'input d'instances `DateTime` en timestamp ou chaines de

127. https://github.com/cakephp/chronos

dates formatées. Egalement, les colonnes "binary" vont accepter un fichier qui gère, et génère le fichier lors de la lecture des données.

Modifié dans la version 3.3.0 : Le type `json` a été ajouté.

Modifié dans la version 3.5.0 : Les types `smallinteger` et `tinyinteger` ont été ajoutés.

Ajouter des Types Personnalisés

static `Cake\Database\Type::map`(*$name*, *$class*)

Si vous avez besoin d'utiliser des types spécifiques qui ne sont pas fournis CakePHP, vous pouvez ajouter des nouveaux types supplémentaires au système de type de CakePHP. Ces classes de type s'attendent à implémenter les méthodes suivantes :

- `toPHP` : Formate la valeur de la base de données pour être utilisée en PHP.
- `toDatabase` : Formate la valeur PHP pour son enregistrement dans la base de données.
- `toStatement` : Convertit la valeur en Statement de base de données.
- `marshal` : Convertit des données plates en objet PHP.

Une façon facile de remplir l'interface basique est d'étendre `Cake\Database\Type`. Par exemple, si vous souhaitez ajouter un type JSON, nous pourrions faire la classe type suivante :

```
// Dans src/Database/Type/JsonType.php

namespace App\Database\Type;

use Cake\Database\Driver;
use Cake\Database\Type;
use PDO;

class JsonType extends Type
{

    public function toPHP($value, Driver $driver)
    {
        if ($value === null) {
            return null;
        }
        return json_decode($value, true);
    }

    public function marshal($value)
    {
        if (is_array($value) || $value === null) {
            return $value;
        }
        return json_decode($value, true);
    }

    public function toDatabase($value, Driver $driver)
    {
        return json_encode($value);
    }

    public function toStatement($value, Driver $driver)
    {
        if ($value === null) {
            return PDO::PARAM_NULL;
```

(suite sur la page suivante)

(suite de la page précédente)

```
        }
        return PDO::PARAM_STR;
    }

}
```

Par défaut, la méthode `toStatement` va traiter les valeurs en chaines qui vont fonctionner pour notre nouveau type. 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\Type;

Type::map('json', 'Cake\Database\Type\JsonType');
```

Nous pouvons ensuite surcharger les données de schema reflected pour utiliser notre nouveau type, et la couche de base de données de CakePHP va automatiquement convertir nos données JSON lors de la création de requêtes. Vous pouvez utiliser les types personnalisés créés en faisant la correspondance des types dans la *méthode _initializeSchema()* de votre Table :

```
use Cake\Database\Schema\TableSchema;

class WidgetsTable extends Table
{

    protected function _initializeSchema(TableSchema $schema)
    {

        $schema->columnType('widget_prefs', 'json');
        return $schema;
    }

}
```

Faire correspondre des types de données personnalisés aux expressions SQL

Nouveau dans la version 3.3.0 : Le support pour le mappage des types de données personnalisés aux expressions SQL a été ajouté dans la version 3.3.0.

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 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. Premièrement, nous allons définir un objet "value" que nous allons pouvoir utiliser pour représenter les données de `POINT` en PHP :

```
// in src/Database/Point.php
namespace App\Database;

// Our value object is immutable.
class Point
{
    protected $_lat;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    protected $_long;

    // Factory method.
    public static function parse($value)
    {
        // Parse the data from MySQL.
        return new static($value[0], $value[1]);
    }

    public function __construct($lat, $long)
    {
        $this->_lat = $lat;
        $this->_long = $long;
    }

    public function lat()
    {
        return $this->_lat;
    }

    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\Expression\FunctionExpression;
use Cake\Database\Type as BaseType;
use Cake\Database\Type\ExpressionTypeInterface;

class PointType extends BaseType implements ExpressionTypeInterface
{
    public function toPHP($value, Driver $d)
    {
        return Point::parse($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)
    {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        if ($value instanceof Point) {
            return new FunctionExpression(
                'POINT',
                [
                    $value->lat(),
                    $value->long()
                ]
            );
        }
        if (is_array($value)) {
            return new FunctionExpression('POINT', [$value[0], $value[1]]);
        }
        // Handle other cases.
    }
}
```

La classe ci-dessus fait quelques éléments intéressants :

- La méthode `toPHP` se charge du parsing des résulats de la requête SQL dans un objet "value".
- La méthode `marshal` se charge de convertir, comme celles données dans 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 choise comme `POINT(10.24, 12.34)`.

Une fois que nous avons construit notre type personnalisé, nous allons *connecter notre type à notre class de table*.

Activer les Objets DateTime Immutables

Nouveau dans la version 3.2 : les objets date/heure immutables ont été ajoutés en 3.2.

Puisque les objets Date/Time sont facilement mutables en place, CakePHP vous permet d'activer les objets immutables. le meilleur endroit pour cela est le fichier **config/bootstrap.php**

```
Type::build('datetime')->useImmutable();
Type::build('date')->useImmutable();
Type::build('time')->useImmutable();
Type::build('timestamp')->useImmutable();
```

Note : Les nouvelles applications auront les objets immutables activés par défaut.

Les Classes de Connection

class `Cake\Database\`**`Connection`**

Les classes de Connection fournissent une interface simple pour interagir avec les connexions à la base de données d'une façon pratique. Elles ont pour objectif d'être une interface plus abstraite à la couche de driver et de fournir des fonctionnalités pour l'exécution des requêtes, le logging des requêtes, et de faire des opérations transactionnelles.

L'exécution des Requêtes

`Cake\Database\Connection::query($sql)`

Une fois que vous avez un objet Connection, vous voudrez probablement réaliser quelques requêtes avec. La couche d'abstraction de CakePHP fournit des fonctionnalités au-dessus de PDO et des drivers natifs. Ces fonctionnalités fournissent une interface similaire à PDO. Il y a quelques différentes façons de lancer les requêtes selon le type de requête que vous souhaitez lancer et selon le type de résultats que vous souhaitez en retour. La méthode la plus basique est `query()` qui vous permet de lancer des requêtes SQL déjà complètes :

```
$stmt = $conn->query('UPDATE posts 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 aux placeholders d'être utilisés :

```
$stmt = $conn->execute(
    'UPDATE posts SET published = ? WHERE id = ?',
    [1, 2]
);
```

Sans aucun typage des informations, `execute` va supposer que tous les placeholders sont des chaînes de valeur. Si vous avez besoin de lier des types spécifiques de données, vous pouvez utiliser leur nom de type abstrait lors de la création d'une requête :

```
$stmt = $conn->execute(
    'UPDATE posts 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 la plus flexible de créer des requêtes est d'utiliser *Query Builder*. Cette approche vous permet de construire des requêtes expressives complexes sans avoir à utiliser une plateforme SQL spécifique :

```
$query = $conn->newQuery();
$query->update('posts')
    ->set(['published' => true])
    ->where(['id' => 2]);
$stmt = $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 = $conn->newQuery();
$query->select('*')
    ->from('posts')
    ->where(['published' => true]);

foreach ($query as $row) {
    // Faire quelque chose avec la ligne.
}
```

Note : Quand vous avez une instance de *Cake\ORM\Query*, vous pouvez utiliser `all()` pour récupérer l'ensemble de résultats pour les requêtes SELECT.

Utiliser les Transactions

Les objets de connexion vous fournissent quelques manières simples pour que vous fassiez des transactions de base de données. La façon la plus basique de faire des transactions est avec les méthodes `begin`, `commit` et `rollback`, qui correspondent à leurs équivalents SQL :

```
$conn->begin();
$conn->execute('UPDATE posts SET published = ? WHERE id = ?', [true, 2]);
$conn->execute('UPDATE posts SET published = ? WHERE id = ?', [false, 4]);
$conn->commit();
```

`Cake\Database\Connection::transactional`(*callable $callback*)

En plus de cette interface, les instances de connexion fournissent aussi la méthode `transactional` ce qui simplifie la gestion des appels begin/commit/rollback :

```
$conn->transactional(function ($conn) {
    $conn->execute('UPDATE posts SET published = ? WHERE id = ?', [true, 2]);
    $conn->execute('UPDATE posts SET published = ? WHERE id = ?', [false, 4]);
});
```

En plus des requêtes basiques, vous pouvez exécuter des requêtes plus complexes en utilisant soit *Query Builder*, soit *Les Objets Table*. La méthode transactionnelle fera ce qui suit :

- Appel de `begin`.
- Appelle la fermeture fournie.
- Si la fermeture lance une exception, un rollback sera délivré. L'exception originelle sera relancée.
- Si la fermeture retourne `false`, un rollback sera délivré.
- Si la fermeture s'exécute avec succès, la transaction sera réalisée.

Interagir avec les Requêtes

Lors de l'utilisation de l'API de plus bas niveau, vous rencontrerez souvent des objets statement (requête). Ces objets vous permettent de manipuler les requêtes préparées sous-jacentes du driver. Après avoir créé et exécuté un objet query, ou en utilisant `execute()`, vous devriez avoir une instance `StatementDecorator`. Elle enveloppe l'objet statement (instruction) basique sous-jacent et fournit quelques fonctionnalités supplémentaires.

Préparer une Requête

Vous pouvez créer un objet statement (requête) en utilisant `execute()`, ou `prepare()`. La méthode `execute()` retourne une requête avec les valeurs fournies en les liant à lui. Alors que `prepare()` retourne une requête incomplète :

```
// Les requêtes à partir de execute auront des valeurs leur étant déjà liées.
$stmt = $conn->execute(
    'SELECT * FROM articles WHERE published = ?',
    [true]
);
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Les Requêtes à partir de prepare seront des paramètres pour les placeholders.
// Vous avez besoin de lier les paramètres avant d'essayer de l'exécuter.
$stmt = $conn->prepare('SELECT * FROM articles WHERE published = ?');
```

Une fois que vous avez préparé une requête, vous pouvez lier les données supplémentaires et l'exécuter.

Lier les Valeurs (Binding)

Une fois que vous avez créé une requête préparée, vous voudrez peut-être lier des données supplémentaires. Vous pouvez lier plusieurs valeurs en une fois en utilisant la méthode `bind`, ou lier les éléments individuels en utilisant `bindValue` :

```
$stmt = $conn->prepare(
    'SELECT * FROM articles WHERE published = ? AND created > ?'
);

// Lier plusieurs valeurs
$stmt->bind(
    [true, new DateTime('2013-01-01')],
    ['boolean', 'date']
);

// Lier une valeur unique
$stmt->bindValue(1, true, 'boolean');
$stmt->bindValue(2, new DateTime('2013-01-01'), 'date');
```

Lors de la création de requêtes, vous pouvez aussi utiliser les clés nommées de tableau plutôt que des clés de position :

```
$stmt = $conn->prepare(
    'SELECT * FROM articles WHERE published = :published AND created > :created'
);

// Lier plusieurs valeurs
$stmt->bind(
    ['published' => true, 'created' => new DateTime('2013-01-01')],
    ['published' => 'boolean', 'created' => 'date']
);

// Lier une valeur unique
$stmt->bindValue('published', true, 'boolean');
$stmt->bindValue('created', new DateTime('2013-01-01'), 'date');
```

Avertissement : Vous ne pouvez pas mixer les clés de position et les clés nommées de tableau dans la même requête.

Executer & Récupérer les Colonnes

Après la préparation d'une requête et après avoir lié les données à celle-ci, vous pouvez l'exécuter et récupérer les lignes. Les requêtes devront être exécutées en utilisant la méthode `execute()`. Une fois exécutée, les résultats peuvent être récupérés en utilisant `fetch()`, `fetchAll()` ou en faisant une itération de la requête :

```
$stmt->execute();

// Lire une ligne.
$row = $stmt->fetch('assoc');

// Lire toutes les lignes.
$rows = $stmt->fetchAll('assoc');

// Lire les lignes en faisant une itération.
foreach ($stmt as $row) {
    // Faire quelque chose
}
```

Note : Lire les lignes avec une itération va récupérer les lignes dans les "deux" modes. Cela signifie que vous aurez à la fois les résultats indexés numériquement et de manière associative.

Récupérer les Compteurs de Ligne

Après avoir exécuté une requête, vous pouvez récupérer le nombre de lignes affectées :

```
$rowCount = count($stmt);
$rowCount = $stmt->rowCount();
```

Vérifier les Codes d'Erreur

Si votre requête n'est pas réussie, vous pouvez obtenir des informations liées à l'erreur en utilisant les méthodes `errorCode()` et `errorInfo()`. Ces méthodes fonctionnent de la même façon que celles fournies par PDO :

```
$code = $stmt->errorCode();
$info = $stmt->errorInfo();
```

Faire des Logs de Requête

Le logs de Requête peuvent être activés lors de la configuration de votre connexion en définissant l'option `log` à `true`. Vous pouvez changer le log de requête à la volée, en utilisant `logQueries` :

```
// Active les logs des requêtes.
$conn->logQueries(true);

// Stoppe les logs des requêtes
$conn->logQueries(false);
```

Quand les logs des requêtes sont activés, les requêtes sont enregistrées dans *`Cake\Log\Log`* en utilisant le niveau de "debug", et le scope de "queriesLog". Vous aurez besoin d'avoir un logger configuré pour capter ce niveau & 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 de la Console
Log::config('queries', [
    'className' => 'Console',
    'stream' => 'php://stderr',
    'scopes' => ['queriesLog']
]);

// Logs des Fichiers
Log::config('queries', [
    'className' => 'File',
    'file' => 'queries.log',
    'scopes' => ['queriesLog']
]);
```

Note : Les logs des requêtes sont seulement à utiliser pour le debuggage/development. Vous ne devriez jamais laisser les logs de requêtes activées en production puisque cela va avoir un impact négatif sur les performances de votre application.

Identifier Quoting

Par défaut CakePHP **ne** quote **pas** les identifiers dans les requêtes SQL générées. La raison pour ceci est que l'ajout de quote autour des identifiers 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 identifiers.

Si vous utilisez un schema datant un peu qui nécessite de quoter les identifiers, vous pouvez l'activer en utilisant le paramètre `quoteIdentifiers` dans votre *Configuration*. Vous pouvez aussi activer cette fonctionnalité à la volée :

```
$conn->getDriver()->enableAutoQuoting();
```

Quand elle est activée, l'identifier quoting va entrainer des requêtes supplémentaires traversales qui convertissent tous les identifiers en objets `IdentifierExpression`.

Note : Les portions de code SQL contenues dans les objets QueryExpression ne seront pas modifiées.

La Mise en Cache de Metadata

L'ORM de CakePHP utilise la reflection de base de données pour déterminer le schema, les indices et les clés étrangères de votre application. Comme cette metadata change peu fréquemment et peut être lourde à accéder, elle est habituellement mise en cache. Par défaut, les metadata sont stockées dans la configuration du 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' => [
```

(suite sur la page suivante)

(suite de la page précédente)

```
        // Autres clés ici.

        // Utilise la config de cache 'orm_metadata' pour les metadata.
        'cacheMetadata' => 'orm_metadata',
    ]
],
```

Vous pouvez aussi configurer les metadata mises en cache 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 metadata. Consultez le chapitre *Shell du Cache du Schéma* pour plus d'information.

Créer des Bases de Données

Si vous voulez créer une connexion (Connection) 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 Connection pour exécuter des requêtes qui créent/modifient les 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 table connection pour accéder au niveau inférieur du constructeur de Requête qui n'inclut pas les fonctionnalités de l'ORM, si nécessaire. Consultez la section *L'exécution des Requêtes* pour plus d'informations :

```
use Cake\ORM\TableRegistry;

// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::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 :

```
// Inside ArticlesController.php

$query = $this->Articles->find();
```

Récupérer les Lignes d'une Table

```
use Cake\ORM\TableRegistry;

// Prior to 3.6 use TableRegistry::get('Articles')
$query = TableRegistry::getTableLocator()->get('Articles')->find();

foreach ($query as $article) {
    debug($article->title);
}
```

Pour les exemples restants, imaginez que `$articles` est une *`ORM\Table`*. Quand vous êtes dans des controllers, vous pouvez utiliser `$this->Articles` plutôt que `$articles`.

Presque chaque méthode dans un objet `Query` va retourner la même requête, cela signifie que les objets `Query` sont lazy, et ne seront pas exécutés à moins que vous 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 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();

foreach ($resultsIteratorObject as $article) {
    debug($article->id);
}

$resultsArray = $articles
    ->find()
    ->where(['id >' => 1])
    ->toArray();

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'extractions ou de traversement.

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 `toArray()` ou n'importe qu'elle méthode héritée de *Collection*, aura pour résultat 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()->extract('title');

foreach ($allTitles as $title) {
    echo $title;
}
```

Vous pouvez aussi récupérer une liste de clé-valeur à partir d'un résultat d'une requête :

```
$list = $articles->find('list');

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 que vous consultiez la section *Collection* pour améliorer vos compétences dans le traversement efficace de 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 « lazily » (paresseusement). Cela signifie qu'une requête n'est pas exécutée jusqu'à ce qu'une des prochaines actions se fasse :

- La requête est itérée avec `foreach()`.
- La méthode `execute()` de query est appelée. Elle retourne l'objet d'instruction sous-jacente, et va être utilisée avec les requêtes insert/update/delete.
- La méthode `first()` de query est appelée. Elle retourne le premier résultat correspondant à l'instruction `SELECT` (ajoute LIMIT 1 à la requête).
- La méthode `all()` de query est appelée. Elle retourne l'ensemble de résultats et peut seulement être utilisée avec les instructions `SELECT`.
- La méthode `toArray()` de query est appelée.

Jusqu'à ce qu'une de ces conditions soit rencontrée, la requête peut être modifiée avec du SQL supplémentaire envoyé à la base de données. Cela signifie que si une Query n'a pas été évaluée, aucun SQL ne sera jamais envoyé à la base de données. Une fois exécutée, la modification et la ré-évaluation d'une requête va entraîner l'exécution de SQL supplémentaire.

Si vous souhaitez jeter un œil sur le SQL que CakePHP génère, vous pouvez activer les *logs de requête* de la base de données.

Les sections suivantes vont vous montrer tout ce qu'il faut savoir sur l'utilisation et la combinaison des méthodes de l'objet Query pour construire des requêtes SQL et extraire les données.

Récupérer vos Données

La plupart des applications web utilisent beaucoup les requêtes de type `SELECT`. CakePHP permet de construire ces requêtes en un clin d'œil. 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 as $row) {
    debug($row->title);
}
```

Vous pouvez également définir des alias pour vos champs en fournissant les champs en tant que tableau associatif :

```
// Résultats dans SELECT id AS pk, title AS aliased_title, body ...
$query = $articles->find();
$query->select(['pk' => 'id', 'aliased_title' => 'title', 'body']);
```

Pour sélectionner des champs distincts, vous pouvez utiliser la méthode `distinct()` :

```
// Résultats dans SELECT DISTINCT country FROM ...
$query = $articles->find();
$query->select(['country'])
    ->distinct(['country']);
```

Pour définir certaines conditions basiques, vous pouvez utiliser la méthode `where()` :

```
// Conditions sont combinées avec AND
$query = $articles->find();
$query->where(['title' => 'First Post', 'published' => true]);

// Vous pouvez aussi appeler where() plusieurs fois
```

(suite sur la page suivante)

(suite de la page précédente)

```
$query = $articles->find();
$query->where(['title' => 'First Post'])
    ->where(['published' => true]);
```

Consultez la section *Conditions Avancées* pour trouver comment construire des conditions `WHERE` plus complexes. Pour appliquer un tri, vous pouvez utiliser la méthode `order` :

```
$query = $articles->find()
    ->order(['title' => 'ASC', 'id' => 'ASC']);
```

Lorsque vous appelez `order()` plusieurs fois sur une même requête, les clauses seront ajoutées les unes après les autres. Cependant, en utilisant les finders, vous pouvez parfois avoir besoin d'écraser le `ORDER BY`. Pour se faire, vous pouvez utiliser le second paramètre de `order()` (mais aussi de `orderAsc()` ou `orderDesc()`) et le définir à `Query::OVERWRITE` ou `true` :

```
$query = $articles->find()
    ->order(['title' => 'ASC']);

// Plus loin dans le code, vous écrasez la clause ORDER BY plutôt
// qu'ajouter une autre clause à l'existante.
$query = $articles->find()
    ->order(['created' => 'DESC'], Query::OVERWRITE);
```

Nouveau dans la version 3.0.12 : En plus de `order`, les méthodes `orderAsc` et `orderDesc` peuvent être utilisées quand vous devez trier selon des expressions complexes :

```
$query = $articles->find();
$concat = $query->func()->concat([
    'title' => 'identifier',
    'synopsis' => 'identifier'
]);
$query->orderAsc($concat);
```

Pour limiter le nombre de lignes ou définir la ligne offset, vous pouvez utiliser les méthodes `limit()` et `page()` :

```
// Récupérer les lignes 50 à 100
$query = $articles->find()
    ->limit(50)
    ->page(2);
```

Comme vous pouvez le voir sur les exemples précédents, toutes les méthodes qui modifient la requête fournissent une interface fluide, vous permettant de construire une requête avec des appels de méthode chaînés.

Sélectionner Tous les Champs d'une Table

Par défaut, une requête va sélectionner tous les champs d'une table sauf si vous appelez la fonction `select()` vous-même et passez certains champs :

```
// Sélectionne seulement id et title de la table articles
$articles->find()->select(['id', 'title']);
```

Si vous souhaitez toujours sélectionner tous les champs d'une table après avoir appelé `select($fields)`, vous pouvez dans cette optique passer l'instance de table à `select()` :

```
// Sélectionne seulement id et title de la table articles
$query = $articlesTable->find();
$query
    ->select(['slug' => $query->func()->concat(['title' => 'identifier', '-', 'id' =>
↪'identifier'])])
    ->select($articlesTable); // Sélectionne tous les champs de articles
```

Utiliser les Fonctions SQL

L'ORM de CakePHP offre une abstraction pour les fonctions les plus communément utilisées par SQL. Utiliser l'abstraction permet à l'ORM de sélectionner l'intégration spécifique de la fonction pour la plateforme que vous souhaitez. Par exemple, `concat` est intégré différemment dans MySQL, PostgreSQL et SQL Server. Utiliser l'abstraction permet à votre code d'être portable :

```
// Résultats dans SELECT COUNT(*) count FROM ...
$query = $articles->find();
$query->select(['count' => $query->func()->count('*')]);
```

Un certain nombre de fonctions communément utilisées peut être créé avec la méthode `func()` :

- `sum()` Calcule une somme. Les arguments sont traités comme des valeurs littérales.
- `avg()` Calcule une moyenne. Les arguments sont traités comme des valeurs littérales.
- `min()` Calcule le min d'une colonne. Les arguments sont traités comme des valeurs littérales.
- `max()` Calcule le max d'une colonne. Les arguments sont traités comme des valeurs littérales.
- `count()` Calcule le count. Les arguments sont traités comme des valeurs littérales.
- `concat()` Concatène deux valeurs ensemble. Les arguments sont traités comme des paramètres liés, à moins qu'ils ne soient marqués comme littéral.
- `coalesce()` Regroupe les valeurs. Les arguments sont traités comme des paramètres liés, à moins qu'ils ne soient marqués comme littéral.
- `dateDiff()` Récupère la différence entre deux dates/times. Les arguments sont traités comme des paramètres liés, à moins qu'ils ne soient marqués comme littéral.
- `now()` Prend soit "time", soit "date" comme argument vous permettant de récupérer soit le time courant, soit la date courante.
- `extract()` Retourne la partie de la date spécifiée de l'expression SQL.
- `dateAdd()` Ajoute l'unité de temps à l'expression de la date.
- `dayOfWeek()` Retourne une FunctionExpression représentant un appel à la fonction SQL WEEKDAY.

Nouveau dans la version 3.1 : Les méthodes `extract()`, `dateAdd()` et `dayOfWeek()` ont été ajoutées.

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'identifaction/littéraux vous permettent de référencer les colonnes ou les autres valeurs littérales de SQL. Les paramètres liés peuvent être utilisés pour ajouter en toute sécurité les données d'utilisateur aux fonctions SQL. Par exemple :

```
$query = $articles->find()->innerJoinWith('Categories');
$concat = $query->func()->concat([
    'Articles.title' => 'identifier',
    ' - CAT: ',
    'Categories.name' => 'identifier',
    ' - Age: ',
    '(DATEDIFF(NOW(), Articles.created))' => 'literal',
]);
$query->select(['link_title' => $concat]);
```

En modifiant les arguments avec une valeur de `literal`, l'ORM va savoir que la clé doit être traitée comme une valeur SQL littérale. En modifiant les arguments avec une valeur d'`identifier`, l'ORM va savoir que la clé doit

être traitée comme un identifieur de champ. 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.

En plus des fonctions ci-dessus, la méthode `func()` peut être utilisée pour créer toute fonction générique SQL comme `year`, `date_format`, `convert`, etc... Par exemple :

```
$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;
```

Vous devriez penser à utiliser le constructeur de fonctions à chaque fois que vous devez mettre des données non fiables dans des fonctions SQL ou des procédures stockées :

```
// Utilise une procédure stockée
$query = $articles->find();
$lev = $query->func()->levenshtein([$search, 'LOWER(title)' => 'literal']);
$query->where(function ($exp) use ($lev) {
    return $exp->between($lev, 0, $tolerance);
});

// Le SQL généré serait
WHERE levenshtein(:c0, lower(street)) BETWEEN :c1 AND :c2
```

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()
    ->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 SMALL, MEDIUM, ou LARGE, nous pourrions faire ce qui suit :

```
$query = $cities->find()
    ->where(function ($exp, $q) {
        return $exp->addCase(
            [
                $q->newExpr()->lt('population', 100000),
                $q->newExpr()->between('population', 100000, 999000),
                $q->newExpr()->gte('population', 999001),
            ],
            ['SMALL',  'MEDIUM', 'LARGE'], # les valeurs correspondantes aux␣
↪conditions
            ['string', 'string', 'string'] # type de chaque valeur
        );
    });
# WHERE CASE
#   WHEN population < 100000 THEN 'SMALL'
#   WHEN population BETWEEN 100000 AND 999000 THEN 'MEDIUM'
```

(suite sur la page suivante)

(suite de la page précédente)

```
#   WHEN population >= 999001 THEN 'LARGE'
#   END
```

A chaque fois qu'il y a moins de conditions qu'il n'y a de valeurs, `addCase` va automatiquement produire une instruction `if .. then .. else` :

```
$query = $cities->find()
    ->where(function ($exp, $q) {
        return $exp->addCase(
            [
                $q->newExpr()->eq('population', 0),
            ],
            ['DESERTED', 'INHABITED'], # valeurs correspondantes aux conditions
            ['string', 'string'] # type de chaque valeur
        );
    });
# WHERE CASE
#   WHEN population = 0 THEN 'DESERTED' ELSE 'INHABITED' END
```

Récupérer des Tableaux plutôt que des Entities

Bien que les ensembles de résultats en objet de l'ORM soient puissants, créer des entities n'est parfois pas nécessaire. Par exemple, quand vous accédez aux données agrégées, la construction d'une Entity peut ne pas être utile. 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->hydrate(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' => 'First Article', 'body' => 'Article 1 body' ...],
    ['id' => 2, 'title' => 'Second Article', 'body' => 'Article 2 body' ...],
    ...
]
```

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 de mapper les ensembles de résultats. Si vous avez besoin de plus de contrôle sur le processus, ou que vous souhaitez réduire les résultats, vous devriez utiliser la fonctionnalité de *Map/Reduce* à la place. Si vous faîtes une requête d'une liste de personnes, vous pourriez calculer leur âge avec le formateur de résultats :

```
// 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['birth_date']->diff(new \DateTime)->y;
        return $row;
    });
});
```

Comme vous pouvez le voir dans l'exemple ci-dessus, 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 sont nécessaires pour retourner un objet itérateur, qui sera utilisé comme valeur retournée pour la requête. Les fonctions de formateurs sont appliquées après que toutes les routines Map/Reduce soient exécutées. Les formateurs de résultat peuvent aussi être appliqués dans les associations `contain`. CakePHP va s'assurer que vos formateurs sont bien scopés. Par exemple, faire ce qui suit fonctionnera comme vous pouvez vous y attendre :

```
// Dans une méthode dans la table Articles
$query->contain(['Authors' => function ($q) {
    return $q->formatResults(function (\Cake\Collection\CollectionInterface $authors)
↪{
        return $authors->map(function ($author) {
            $author['age'] = $author['birth_date']->diff(new \DateTime)->y;
            return $author;
        });
    });
}]);

// Récupère les résultats
$results = $query->all();

// Affiche 29
echo $results->first()->author->age;
```

Comme vu précédemment, les formateurs attachés aux constructeurs de requête associées sont limités pour agir seulement sur les données dans 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()`, `andWhere()` et `orWhere()`. La méthode `where()` fonctionne de manière similaire aux tableaux de conditions des versions précédentes de CakePHP :

```
$query = $articles->find()
    ->where([
        'author_id' => 3,
        'OR' => [['view_count' => 2], ['view_count' => 3]],
    ]);
```

Ce qui précède générerait le code SQL :

```
SELECT * FROM articles WHERE author_id = 3 AND (view_count = 2 OR view_count = 3)
```

Si vous préférez éviter des tableaux imbriqués profondément, vous pouvez utiliser les méthodes `orWhere()` et `andWhere()` pour construire vos requêtes. Chaque méthode définit l'opérateur de combinaison utilisé entre les conditions courante et précédente. Par exemple :

```
$query = $articles->find()
    ->where(['author_id' => 2])
    ->orWhere(['author_id' => 3]);
```

Ce qui précède générerait le code SQL :

```
SELECT * FROM articles WHERE (author_id = 2 OR author_id = 3)
```

En combinant `orWhere()` et `andWhere()`, vous pouvez exprimer des conditions complexes qui utilisent un mix d'opérateurs :

```
$query = $articles->find()
    ->where(['author_id' => 2])
    ->orWhere(['author_id' => 3])
    ->andWhere([
        'published' => true,
        'view_count >' => 10
    ])
    ->orWhere(['promoted' => true]);
```

Ce qui précède générerait le code SQL :

```
SELECT *
FROM articles
WHERE (
    (
        (author_id = 2 OR author_id = 3)
        AND
        (published = 1 AND view_count > 10)
    )
    OR promoted = 1
)
```

En utilisant les fonctions en paramètres pour `orWhere()` et `andWhere()`, vous pouvez organiser ensemble les conditions avec les objets expression :

```
$query = $articles->find()
    ->where(['title LIKE' => '%First%'])
    ->andWhere(function ($exp) {
        return $exp->or([
            'author_id' => 2,
            'is_highlighted' => true
        ]);
    });
```

Ce qui précède générerait le code SQL :

```
SELECT *
FROM articles
WHERE (
    title LIKE '%First%'
    AND
    (author_id = 2 OR is_highlighted = 1)
)
```

Les objets expression qui sont passés dans les fonctions `where()` ont deux types de méthodes. Les premiers types de méthode sont des **combinateurs**. Les méthodes `and()` et `or()` créent de nouveaux objets expression qui changent **la façon dont** les conditions sont combinées. Les seconds types de méthode sont les **conditions**. Les conditions sont ajoutées dans une expression où elles sont combinées avec le combinateur courant.

Par exemple, appeler `$exp->and(...)` va créer un nouvel objet `Expression` qui combine toutes les conditions qu'il contient avec `AND`. Alors que `$exp->or()` va créer un nouvel objet `Expression` qui combine toutes les conditions qui lui sont ajoutées avec `OR`. Un exemple d'ajout de conditions avec une objet `Expression` serait :

```
$query = $articles->find()
    ->where(function ($exp) {
        return $exp
            ->eq('author_id', 2)
            ->eq('published', true)
            ->notEq('spam', true)
            ->gt('view_count', 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 (
author_id = 2
AND published = 1
AND spam != 1
AND view_count >= 10)
```

Obsolète depuis la version 3.5.0 : Depuis la version 3.5.0, la méthode `orWhere()` est dépréciée. Cette méthode crée des requêtes SQL difficiles à prédire en fonction de l'état actuel de la requête. Utilisez plutôt `where()` car son comportement est plus prévisible et plus facile à comprendre.

Cependant, si nous souhaitons utiliser les deux conditions `AND` & `OR`, nous pourrions faire ce qui suit :

```
$query = $articles->find()
    ->where(function ($exp) {
        $orConditions = $exp->or(['author_id' => 2])
            ->eq('author_id', 5);
        return $exp
            ->add($orConditions)
            ->eq('published', true)
            ->gte('view_count', 10);
    });
```

Ce qui générerait le code SQL suivant :

```
SELECT *
FROM articles
WHERE (
(author_id = 2 OR author_id = 5)
AND published = 1
AND view_count > 10)
```

Les méthodes `or()` et `and()` vous permettent aussi d'utiliser les fonctions en paramètres. C'est souvent plus facile à lire que les méthodes chaînées :

```
$query = $articles->find()
    ->where(function ($exp) {
        $orConditions = $exp->or(function ($or) {
            return $or->eq('author_id', 2)
                ->eq('author_id', 5);
        });
        return $exp
            ->not($orConditions)
```

(suite sur la page suivante)

(suite de la page précédente)

```
            ->lte('view_count', 10);
    });
```

Vous pouvez faire une négation des sous-expressions en utilisant `not()` :

```
$query = $articles->find()
    ->where(function ($exp) {
        $orConditions = $exp->or(['author_id' => 2])
            ->eq('author_id', 5);
        return $exp
            ->not($orConditions)
            ->lte('view_count', 10);
    });
```

Ce qui générerait le code SQL suivant :

```
SELECT *
FROM articles
WHERE (
NOT (author_id = 2 OR author_id = 5)
AND view_count <= 10)
```

Il est aussi possible de construire les expressions en utilisant les fonctions SQL :

```
$query = $articles->find()
    ->where(function ($exp, $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 = $cities->find()
    ->where(function ($exp, $q) {
        return $exp->eq('population', '10000');
    });
# WHERE population = 10000
```

— `notEq()` Crée une condition d'inégalité :

```
$query = $cities->find()
    ->where(function ($exp, $q) {
        return $exp->notEq('population', '10000');
```

(suite sur la page suivante)

(suite de la page précédente)

```
    });
# WHERE population != 10000
```

— `like()` Crée une condition en utilisant l'opérateur `LIKE` :

```
$query = $cities->find()
    ->where(function ($exp, $q) {
        return $exp->like('name', '%A%');
    });
# WHERE name LIKE "%A%"
```

— `notLike()` Crée une condition négative de type `LIKE` :

```
$query = $cities->find()
    ->where(function ($exp, $q) {
        return $exp->notLike('name', '%A%');
    });
# WHERE name NOT LIKE "%A%"
```

— `in()` Crée une condition en utilisant `IN` :

```
$query = $cities->find()
    ->where(function ($exp, $q) {
        return $exp->in('country_id', ['AFG', 'USA', 'EST']);
    });
# WHERE country_id IN ('AFG', 'USA', 'EST')
```

— `notIn()` Crée une condition négative en utilisant `IN` :

```
$query = $cities->find()
    ->where(function ($exp, $q) {
        return $exp->notIn('country_id', ['AFG', 'USA', 'EST']);
    });
# WHERE country_id NOT IN ('AFG', 'USA', 'EST')
```

— `gt()` Crée une condition > :

```
$query = $cities->find()
    ->where(function ($exp, $q) {
        return $exp->gt('population', '10000');
    });
# WHERE population > 10000
```

— `gte()` Crée une condition >= :

```
$query = $cities->find()
    ->where(function ($exp, $q) {
        return $exp->gte('population', '10000');
    });
# WHERE population >= 10000
```

— `lt()` Crée une condition < :

```
$query = $cities->find()
    ->where(function ($exp, $q) {
        return $exp->lt('population', '10000');
    });
# WHERE population < 10000
```

— `lte()` Crée une condition <= :

```
$query = $cities->find()
    ->where(function ($exp, $q) {
        return $exp->lte('population', '10000');
    });
# WHERE population <= 10000
```

— `isNull()` Crée une condition `IS NULL` :

```
$query = $cities->find()
    ->where(function ($exp, $q) {
        return $exp->isNull('population');
    });
# WHERE (population) IS NULL
```

— `isNotNull()` Crée une condition négative `IS NULL` :

```
$query = $cities->find()
    ->where(function ($exp, $q) {
        return $exp->isNotNull('population');
    });
# WHERE (population) IS NOT NULL
```

— `between()` Crée une condition `BETWEEN` :

```
$query = $cities->find()
    ->where(function ($exp, $q) {
        return $exp->between('population', 999, 5000000);
    });
# WHERE population BETWEEN 999 AND 5000000,
```

— `exists()` Crée une condition en utilisant `EXISTS` :

```
$subquery = $cities->find()
    ->select(['id'])
    ->where(function ($exp, $q) {
            return $exp->equalFields('countries.id', 'cities.country_id');
    })
    ->andWhere(['population >', 5000000]);

$query = $countries->find()
    ->where(function ($exp, $q) use ($subquery) {
        return $exp->exists($subquery);
    });
# WHERE EXISTS (SELECT id FROM cities WHERE countries.id = cities.country_id AND␣
↪population > 5000000)
```

— `notExists()` Crée une condition négative en utilisant `EXISTS` :

```
$subquery = $cities->find()
    ->select(['id'])
    ->where(function ($exp, $q) {
            return $exp->equalFields('countries.id', 'cities.country_id');
    })
    ->andWhere(['population >', 5000000]);

$query = $countries->find()
    ->where(function ($exp, $q) use ($subquery) {
        return $exp->notExists($subquery);
    });
# WHERE NOT EXISTS (SELECT id FROM cities WHERE countries.id = cities.country_id␣
↪AND population > 5000000)
```

(suite sur la page suivante)

(suite de la page précédente)

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. 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.

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[]']);

// Or include IN to automatically cast to an array.
$query = $articles->find()
    ->where(['id IN' => $ids]);
```

Ce qui est au-dessus va automatiquement créer `id IN (...)` plutôt que `id = ?`. Ceci 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 type*. Ceci fonctionne aussi avec les 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 d'une condition s'attend à être `null` ou tout autre valeur, vous pouvez utiliser l'opérateur `IS` pour créer automatiquement la bonne expression :

```
$query = $categories->find()
    ->where(['parent_id IS' => $parentId]);
```

Ce qui précède va créer `parent_id` = :c1` ou `parent_id IS NULL` selon le type de `$parentId`.

Création Automatique de IS NOT NULL

Quand une valeur d'une condition s'attend à être `null` ou tout autre valeur, vous pouvez utiliser l'opérateur `IS NOT` pour créer automatiquement la bonne expression :

```
$query = $categories->find()
    ->where(['parent_id IS NOT' => $parentId]);
```

Ce qui précède va créer `parent_id` != :c1` ou `parent_id IS NOT NULL` selon le type de `$parentId`.

Raw Expressions

Quand vous ne pouvez pas construire le code SQL, vous devez utiliser le constructeur de requête, vous pouvez utiliser les objets `Expression` pour ajouter des extraits de code SQL à vos requêtes :

```
$query = $articles->find();
$expr = $query->newExpr()->add('1 + 1');
$query->select(['two' => $expr]);
```

Les objets `Expression` peuvent être utilisés avec les méthodes du constructeur de requête comme `where()`, `limit()`, `group()`, `select()` et d'autres méthodes.

> **Avertissement :** Utiliser les objets `Expression` vous laisse vulnérable aux injections SQL. Vous devrez évitez d'interpoler les données d'utilisateur dans les expressions.

Récupérer les Résultats

Une fois que vous avez fait votre requête, vous voudrez récupérer des lignes résultantes. Il y a plusieurs façons de faire ceci :

```
// Itérer la requête
foreach ($query as $row) {
    // Do stuff.
}

// Récupérer les résultats
$results = $query->all();
```

Vous pouvez utiliser toutes les méthodes *des Collections* sur vos objets query pour traiter préalablement 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 des Enregistrements

En utilisant un objet query unique, il est possible d'obtenir le nombre total de lignes trouvées pour un ensemble de conditions :

```
$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 savoir la taille totale de l'ensemble des résultats en avance, sans avoir à construire un autre objet `Query`. De cette façon, tous les résultats formatés et les routines map-reduce sont ignorées quand vous utilisez la méthode `count()`.

De plus, il est possible de retourner le nombre total pour une requête contenant des clauses group by sans avoir à réécrire la requête en aucune façon. Par exemple, considérons la requête qui permet de récupérer les ids d'article et leur nombre de commentaires :

```
$query = $articles->find();
$query->select(['Articles.id', $query->func()->count('Comments.id')])
    ->matching('Comments')
    ->group(['Articles.id']);
$total = $query->count();
```

Après avoir compté, la requête peut toujours être utilisée pour récupérer les enregistrements associés :

```
$list = $query->all();
```

Parfois, vous voulez fournir une méthode alternative pour compter le nombre total d'enregistrements d'une requête. Un cas d'utilisation courante pour ceci est pour fournir une valeur mise en cache ou un nombre estimé total de lignes, ou pour 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 appelle la méthode count, elle va recevoir le nombre de lignes estimées écrit en dur.

Mettre en Cache les Résultats Chargés

Quand vous récupérez les 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 l'ensemble des résultats de la requête. Si un seul argument est fourni à `cache()` alors la configuration du cache "default" va être utilisée. Vous pouvez contrôler la configuration du 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 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'évènement.

Quand les résultats pour une requête mise en cache sont récupérés, ce qui suit va arriver :

1. L'évènement `Model.beforeFind` est déclenché.
2. Si la requête a des ensembles de résultats, ceux-ci vont être retournés.
3. La clé du cache va être déterminée et les données du cache vont être lues. Si les données du cache sont vides, ces résultats vont être retournés.
4. Si le cache n'est pas présent, la requête sera exécutée et un nouveau `ResultSet` sera créé. Ce `ResultSet` sera écrit dans le cache et sera retourné.

Note : Vous ne pouvez pas mettre en cache un résultat de requête streaming.

Chargement des Associations

Le constructeur peut vous aider à retrouver les données de plusieurs tables en même temps avec le minimum de requêtes. Pour pouvoir récupérer les données associées, vous devez 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**.

Chaque Eager loading évite plusieurs problèmes potentiels de chargement lors du lazy loading dans un ORM. Les requêtes générées par le eager loading peuvent augmenter l'impact des jointures, permettant de faire des requêtes plus efficaces. Dans CakePHP vous définissez des associations chargées en eager en utilisant la méthode "contain" :

```
// 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']);
```

Ce qui est au-dessus va charger les auteurs et commentaires liés pour chaque article de l'ensemble de résultats. Vous pouvez charger les associations imbriquées en utilisant les tableaux imbriqués pour définir les associations à charger :

```
$query = $articles->find()->contain([
    'Authors' => ['Addresses'], 'Comments' => ['Authors']
]);
```

D'une autre façon, vous pouvez exprimer des associations imbriquées en utilisant la notation par point :

```
$query = $articles->find()->contain([
    'Authors.Addresses',
    'Comments.Authors'
]);
```

Vous pouvez charger les associations en eager aussi profondément que vous le souhaitez :

```
$query = $products->find()->contain([
    'Shops.Cities.Countries',
    'Shops.Managers'
]);
```

Vous pouvez sélectionner des champs de toutes les associations en utilisant plusieurs appels à `contain()` :

```
$query = $this->find()->select([
    'Realestates.id',
    'Realestates.title',
    'Realestates.description'
])
->contain([
    'RealestateAttributes' => [
        'Attributes' => [
            'fields' => [
                // Les champs alias dans contain() doivent
                // inclure le préfixe du modèle à mapper correctement.
                'Attributes__name' => 'attr_name'
            ]
        ]
    ]
])
->contain([
    'RealestateAttributes' => [
        'fields' => [
            'RealestateAttributes.realestate_id',
            'RealestateAttributes.value'
        ]
    ]
])
->where($condition);
```

Si vous avez besoin de remettre les contain sur une requête, vous pouvez définir le second argument à `true` :

```
$query = $articles->find();
$query->contain(['Authors', 'Comments'], true);
```

Passer des Conditions à Contain

Avec l'utilisation de `contain()`, vous pouvez restreindre les données retournées par les associations et les filtrer par conditions :

```
// Dans un controller ou une méthode de table.

$query = $articles->find()->contain('Comments', function ($q) {
   return $q
        ->select(['body', 'author_id'])
        ->where(['Comments.approved' => true]);
});
```

Cela fonctionne aussi pour la pagination au niveau du Controller :

```
$this->paginate['contain'] = [
    'Comments' => function (\Cake\ORM\Query $query) {
        return $query->select(['body', 'author_id'])
        ->where(['Comments.approved' => true]);
    }
];
```

Note : Quand vous limitez les champs qui sont récupérés d'une association, vous **devez** vous assurer que les colonnes de clé étrangère soient sélectionnées. Ne pas sélectionner les champs de clé étrangère va entraîner la non présence des données associées dans le résultat final.

Il est aussi possible de restreindre les associations imbriquées profondément en utilisant la notation par point :

```
$query = $articles->find()->contain([
    'Comments',
    'Authors.Profiles' => function ($q) {
        return $q->where(['Profiles.is_published' => true]);
    }
]);
```

Dans l'exemple ci-dessus, 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 méthodes de finder personnalisées 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 des enregistrements associés. Pour le reste des types d'association, vous pouvez utiliser chaque clause que

l'objet query fournit.

Si vous devez prendre le contrôle total d'une requête qui est générée, vous pouvez appeler `contain()` pour ne pas ajouter les contraintes `foreignKey` à la requête générée. Dans ce cas, vous devez utiliser un tableau en passant `foreignKey` et `queryBuilder` :

```
$query = $articles->find()->contain([
    'Authors' => [
        'foreignKey' => false,
        'queryBuilder' => function ($q) {
            return $q->where(...); // Full conditions for filtering
        }
    ]
]);
```

Si vous avez limité les champs que vous chargez avec `select()` mais que vous souhaitez aussi charger les champs enlevés des associations avec contain, vous pouvez passer l'objet association à `select()` :

```
// Sélectionne id & title de articles, mais tous les champs enlevés pour Users.
$query = $articles->find()
    ->select(['id', 'title'])
    ->select($articles->Users)
    ->contain(['Users']);
```

D'une autre façon, si vous pouvez faire des associations multiples, vous pouvez utiliser `enableAutoFields()` :

```
// Sélectionne id & title de articles, mais tous les champs enlevés de
// Users, Comments et Tags.
$query->select(['id', 'title'])
    ->contain(['Comments', 'Tags'])
    ->enableAutoFields(true) // Avant 3.4.0 utilisez autoFields(true)
    ->contain(['Users' => function($q) {
        return $q->autoFields(true);
    }]);
```

Nouveau dans la version 3.1 : La sélection des colonnes via un objet association a été ajouté dans 3.1

Ordonner les Associations Contain

Quand vous chargez des associations HasMany et BelongsToMany, vous pouvez utiliser l'option `sort` pour ordonner les données dans ces associations :

```
$query->contain([
    'Comments' => [
        'sort' => ['Comments.created' => 'DESC']
    ]
]);
```

Filtering by Associated Data

Un cas de requête couramment fait avec les associations est de trouver les enregistrements qui "matchent" les données associées spécifiques. Par exemple si vous avez "Articles belongsToMany Tags", vous aurez probablement envie de trouver les Articles qui ont le tag CakePHP. C'est extrêmement simple à faire avec l'ORM de CakePHP :

```
// Dans un controller ou table de méthode.

$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 avec les articles récemment publiés en utilisant ce qui suit :

```
$query = $authors->find();
$query->matching('Articles', function ($q) {
    return $q->where(['Articles.created >=' => new DateTime('-10 days')]);
});
```

Filtrer des associations imbriquées est étonnamment facile, et la syntaxe doit déjà vous être familière :

```
// Dans un controller ou une table de méthode.
$query = $products->find()->matching(
    'Shops.Cities.Countries', function ($q) {
        return $q->where(['Countries.name' => 'Japan']);
    }
);

// Récupère les articles uniques qui étaient commentés par 'markstory'
// en utilisant la variable passée
// Les chemins avec points doivent être utilisés plutôt que les appels
// imbriqués de matching()
$username = 'markstory';
$query = $articles->find()->matching('Comments.Users', function ($q) use ($username) {
    return $q->where(['username' => $username]);
});
```

Note : Comme cette fonction va créer un `INNER JOIN`, vous pouvez appeler `distinct` sur le find de la requête puisque vous aurez des lignes dupliquées si les conditions ne les excluent pas déjà. Ceci peut être le cas, par exemple, quand les mêmes utilisateurs commentent plus d'une fois un article unique.

Les données des associations qui sont "matchés" (appairés) 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 recevoir 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 vouliez utiliser `matching()` mais que vous n'êtes pas intéressé par le chargement des champs dans un ensemble de résultats. 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' => 'Japan']);
    }
);
```

De même, la seule différence est qu'aucune colonne supplémentaire ne sera ajoutée à l'ensemble de résultats et aucune propriété `_matchingData` ne sera définie.

Nouveau dans la version 3.1 : Query : :innerJoinWith() a été ajoutée dans 3.1

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' => 'boring']);
    });
```

L'exemple ci-dessus va trouver tous les articles qui n'étaient pas taggés avec le mot `boring`. Vous pouvez aussi utiliser cette méthode avec les associations HasMany. Vous pouvez, par exemple, trouver tous les auteurs qui n'ont 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 les associations profondes. 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) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        return $q->where(['username' => 'jose']);
    });
```

Puisque les articles avec aucun commentaire satisfont aussi la condition du dessus, vous pouvez combiner `matching()` et `notMatching()` dans la même requête. L'exemple suivant va trouver 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 find puisque sinon vous allez avoir des lignes dupliquées.

Gardez à l'esprit que le contraire de la fonction `matching()`, `notMatching()` ne va pas ajouter toutes les données à la propriété `_matchingData` dans les résultats.

Nouveau dans la version 3.1 : Query : :notMatching() a été ajoutée dans 3.1

Utiliser leftJoinWith

Dans certaines situations, vous aurez à calculer un résultat selon une association, sans avoir à charger tous les enregistrements. Par exemple, si vous voulez charger le nombre total de commentaires qu'un article a, ainsi que toutes les 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); // Avant 3.4.0 utilisez autoFields(true)
```

Le résultat de la requête ci-dessus va contenir les données de l'article et la propriété `total_comments` pour chacun d'eux.

`leftJoinWith()` peut aussi être utilisée avec des associations profondes. C'est utile par exemple pour rapporter le nombre d'articles taggés par l'auteur 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' => 'awesome']);
    })
    ->group(['Authors.id'])
    ->enableAutoFields(true); // Avant 3.4.0 utilisez autoFields(true)
```

Cette fonction ne va charger aucune colonne des associations spécifiées dans l'ensemble de résultats.

Nouveau dans la version 3.1 : Query : :leftJoinWith() a été ajoutée dans 3.1

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()
    ->hydrate(false)
    ->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()
    ->hydrate(false)
    ->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()
    ->hydrate(false)
    ->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 d'un tableau basé sur les conditions, 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
$query = $articles->find();
$query->leftJoin(
    ['Authors' => 'authors'],
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ['Authors.id = Articles.author_id']
);

// Jointure avec un alias, tableau de conditions, et de types
$query = $articles->find();
$query->innerJoin(
    ['Authors' => 'authors'],
    [
        'Authors.promoted' => true,
        'Authors.created' => new DateTime('-5 days'),
        'Authors.id = Articles.author_id'
    ],
    ['Authors.promoted' => 'boolean', 'Authors.created' => 'datetime']
);
```

Notez que si vous définissez l'option `quoteIdentifiers` à `true` quand vous configurez votre `Connection`, les conditions mettant en relation deux champs de tables différentes 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` sont bien quotés dans la requête générée, permettant d'éviter des erreurs avec certains drivers (PostgreSQL notamment).

Insérer des Données

Au contraire des exemples précédents, vous ne devez pas utiliser `find()` pour créer des requêtes d'insertion. A la place, créez un nouvel objet `Query` en utilisant `query()` :

```
$query = $articles->query();
$query->insert(['title', 'body'])
    ->values([
        'title' => 'First post',
        'body' => 'Some body text'
    ])
    ->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(['title', 'body'])
    ->values([
        'title' => 'First post',
        'body' => 'Some body text'
    ])
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ->values([
        'title' => 'Second post',
        'body' => 'Another body text'
    ])
    ->execute();
```

Généralement, il est plus facile d'insérer des données en utilisant les entities et *ORM\Table::save()*. En composant des requêtes `SELECT` et `INSERT` ensemble, vous pouvez créer des requêtes de style `INSERT INTO ... SELECT` :

```
$select = $articles->find()
    ->select(['title', 'body', 'published'])
    ->where(['id' => 3]);

$query = $articles->query()
    ->insert(['title', 'body', 'published'])
    ->values($select)
    ->execute();
```

Note : Ajouter des enregistrements avec le constructeur de requêtes ne va pas déclencher les events comme `Model.afterSave`. À la place, vous pouvez utiliser *l'ORM pour sauvegardes 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 `ORM\Table::patchEntity()`.

Note : Mettre à jour des enregistrements avec le constructeur de requêtes ne va pas déclencher les events 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. A 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 *ORM\Table::delete()*.

Prévention contre les Injections SQL

Alors que l'ORM et les couches d'abstraction de base de données empêchent la plupart des problèmes relatifs aux injections SQL, il est toujours possible que vous soyez vulnérables face à une utilisation incorrecte.

Lorsque vous utilisez des tableaux de conditions, la clé (la partie à gauche) ou bien une valeur seule ne doivent pas contenir de données utilisateur :

```
$query->where([
    // Utiliser cette clé est dangereux car elle sera insérée dans la
    // requête générée telle quelle
    $userData => $value,

    // Le même commentaire s'applique pour les valeurs seule : il est
    // dangereux de les utiliser avec des données utilisateur
    $userData,
    "MATCH (comment) AGAINST ($userData)",
    'created < NOW() - ' . $userData
]);
```

Lorsque vous utilisez le constructeur de fonctions, les noms de colonnes ne doivent pas contenir de données provenant d'utilisateurs :

```
$query->where(function ($exp) use ($userData, $values) {
    // Les noms de colonnes dans toutes les expressions ne sont pas sûrs.
    return $exp->in($userData, $values);
});
```

Lorsque vous construisez des expressions, 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(['two' => $expr]);
```

Lier les Valeurs (Binding)

Il est possible de protéger vos requêtes en utilisant le « binding ». De la même manière que vous pouvez *« binder » des valeurs pour les requêtes préparées*, des valeurs peuvent être « bindées » aux requêtes en utilisant la méthode `Cake\Database\Query::bind()`.

L'exemple ci-dessous est une alternative sûre par rapport à la version donnée plus haut, qui serait vulnérable à une injection SQL :

```
$query
    ->where([
        'MATCH (comment) AGAINST (:userData)',
        'created < NOW() - :moreUserData'
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ])
    ->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 (`:`) !

Plus de Requêtes Complexes

Le constructeur de requête est capable de construire des requêtes complexes comme les requêtes `UNION` et sous-requêtes.

Unions

Les unions sont créées en composant une ou plusieurs requêtes select ensemble :

```
$inReview = $articles->find()
    ->where(['need_review' => 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(['need_review' => true]);

$unpublished = $articles->find()
    ->where(['published' => false]);

$unpublished->unionAll($inReview);
```

Sous-Requêtes

Les sous-requêtes sont une fonctionnalité puissante dans les bases de données relationnelles et les construire dans CakePHP est assez intuitif. En composant les requêtes ensemble, vous pouvez faire des sous-requêtes :

```
$matchingComment = $articles->association('Comments')->find()
    ->select(['article_id'])
    ->distinct()
    ->where(['comment LIKE' => '%CakePHP%']);

$query = $articles->find()
    ->where(['id' => $matchingComment]);
```

Les sous-requêtes sont acceptées partout où une expression query peut être utilisée. Par exemple, dans les méthodes `select()` et `join()`.

Adding Locking Statements

Most relational database vendors support taking out locks when doing select operations. You can use the `epilog()` method for this :

```
// In MySQL
$query->epilog('FOR UPDATE');
```

The `epilog()` method allows you to append raw SQL to the end of queries. You should never put raw user data into `epilog()`.

Exécuter des Requêtes Complexes

Bien que le constructeur de requêtes facilite la construction de la plupart des requêtes, les requêtes très complexes peuvent être fastidieuses et compliquées à construire. Vous voudrez surement vous référer à *l'exécution directe du SQL souhaité*.

Exécuter directement le SQL vous permet d'affiner la requête qui sera utilisée. 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 *connection 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)
    {
        $this->setTable('my_table');

        // Avant 3.4.0
        $this->table('my_table');
    }

}
```

Aucune convention d'inflection 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)
    {
        $this->setPrimaryKey('my_id');

        // Avant 3.4.0
        $this->primaryKey('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)
    {
        $this->setEntityClass('App\Model\Entity\PO');

        // Avant 3.4.0
        $this->entityClass('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 `TableRegistry` :

```
// Dans un controller ou dans une méthode de table.
use Cake\ORM\TableRegistry;

// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->get('Articles');
```

La classe TableRegistry fournit les divers 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 TableRegistry* 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
// Prior to 3.6 use TableRegistry::get('PluginName.Articles')
$articlesTable = TableRegistry::getTableLocator()->get('PluginName.Articles');

// Table de plugin préfixé par Vendor
// Prior to 3.6 use TableRegistry::get('VendorName/PluginName.Articles')
$articlesTable = TableRegistry::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 des hook utiles si vous souhaitez et ajouter de la logique dans l'ORM sans faire de sous-classe ou sans surcharger les méthodes. Les écouteurs d'event peuvent être définis dans les classes table ou behavior. Vous pouvez aussi utiliser un gestionnaire d'event de table pour lier les écouteurs dedans.

Lors de l'utilisation des méthodes callback des behaviors attachés dans la méthode `initialize()` va voir ses écouteurs lancés **avant** que les méthodes de callback de la table ne soient déclenchées. Ceci suit la même séquence que les controllers & les components.

Pour ajouter un écouteur d'event à une classe Table ou un Behavior, implémentez simplement les signatures de méthode comme décrit ci-dessus. Consultez les *Événements système* pour avoir plus de détails sur la façon d'utiliser le sous-système d'events.

Liste des Events

```
— Model.initialize
— Model.beforeMarshal
— 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`(*Event $event*, *ArrayObject $data*, *ArrayObject $options*)

L'event `Model.initialize` est déclenché après que les méthodes de constructeur et initialize sont 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 array(
            'Model.initialize' => 'initializeEvent',
        );
    }
    public function initializeEvent($event)
    {
        $table = $event->getSubject();
        // faire quelque chose ici
    }
}
```

et attacher l'écouteur à `EventManager` comme ce qui suit :

```
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`(*Event $event*, *ArrayObject $data*, *ArrayObject $options*)

L'event `Model.beforeMarshal` est déclenché avant que les données de request ne soient converties en entities. Consultez la documentation *Modifier les Données Requêtées Avant de Construire les Entities* pour plus d'informations.

beforeFind

`Cake\ORM\Table::beforeFind`(*Event $event*, *Query $query*, *ArrayObject $options*, *$primary*)

L'event `Model.beforeFind` est lancé avant chaque opération find. En stoppant l'event et en fournissant une valeur de retour, vous pouvez outrepasser entièrement l'opération find. 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. Toutes les associations participant à une requête vont avoir un event `Model.beforeFind` déclenché. Pour les associations qui utilisent les joins, une requête factice sera fournie. Dans votre écouteur d'event, vous pouvez définir des champs supplémentaires, des conditions, des joins ou des formateurs de résultat. Ces options/fonctionnalités seront copiées dans la requête racine.

Vous pouvez utiliser ce callback pour restreindre les opérations find basées sur le rôle de l'utilisateur, ou prendre des décisions de mise en cache basées sur le chargement courant.

Dans les versions précédentes de CakePHP, il y avait un callback `afterFind`, ceci a été remplacé par les fonctionnalités de *Modifier les Résultats avec Map/Reduce* et les constructeurs d'entity.

buildValidator

`Cake\ORM\Table::buildValidator`(*Event $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`(*Event $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`(*Event $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`(*Event $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`(*Event $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`(*Event $event*, *EntityInterface $entity*, *ArrayObject $options*)

L'event `Model.afterSave` est déclenché après qu'une entity ne soit sauvegardée.

afterSaveCommit

`Cake\ORM\Table::afterSaveCommit`(*Event $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`(*Event $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. La manière de stopper un event est documentée *ici*.

afterDelete

`Cake\ORM\Table::afterDelete`(*Event $event*, *EntityInterface $entity*, *ArrayObject $options*)

L'event `Model.afterDelete` est déclenché après qu'une entity a été supprimée.

afterDeleteCommit

`Cake\ORM\Table::afterDeleteCommit` (*Event $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.

Behaviors

`Cake\ORM\Table::addBehavior` (*$name*, *array $options = []*)

Les Behaviors fournissent une façon facile 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 première raison est les 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)
    {
        $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)
    {
        $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 quelles tables utilisent quelles connexions. C'est avec la méthode `defaultConnectionName()` :

```
namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{
    public static function defaultConnectionName() {
        return 'replica_db';
    }
}
```

Note : La méthode `defaultConnectionName()` **doit** être statique.

Utiliser le TableRegistry

class `Cake\ORM\`**`TableRegistry`**

Comme nous l'avons vu précédemment, la classe TableRegistry 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

static `Cake\ORM\TableRegistry::`**`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 = TableRegistry::getTableLocator()->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 connection 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 `config()`. Les données de configuration sont stockées *par alias*, et peuvent être surchargées par une méthode `initialize()` de l'objet :

```
TableRegistry::config('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

static `Cake\ORM\TableRegistry::`**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 :

```
TableRegistry::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**

Alors 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 propriétés et des méthodes persistantes pour manipuler et accéder aux données qu'ils contiennent.

Les entities sont créées pour vous par CakePHP à chaque fois que vous utilisez `find()` sur un objet table.

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 pourrions 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 propriétés avec les données que vous voulez y stocker :

```
use App\Model\Entity\Article;

$article = new Article([
    'id' => 1,
    'title' => 'New Article',
    'created' => new DateTime('now')
]);
```

Une autre approche pour récupérer une nouvelle entity est d'utiliser la méthode `newEntity()` de l'objet `Table` :

```
use Cake\ORM\TableRegistry;

// Prior to 3.6 use TableRegistry::get('Articles')
$article = TableRegistry::getTableLocator()->get('Articles')->newEntity();

$article = TableRegistry::getTableLocator()->get('Articles')->newEntity([
    'id' => 1,
    'title' => 'New Article',
    'created' => new DateTime('now')
]);
```

Accéder aux Données de l'Entity

Les entities fournissent quelques 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()` :

```
$article->set('title', 'Ceci est mon premier post');
echo $article->get('title');
```

Quand vous utilisez `set()`, vous pouvez mettre à jour plusieurs propriétés en une 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.

Accesseurs & Mutateurs

En plus de l'interface simple get/set, les entities vous permettent de fournir des méthodes accesseurs et mutateurs. Ces méthodes vous laissent personnaliser la façon dont les propriétés sont lues ou définies.

Les accesseurs utilisent la convention `_get` suivi par la version en camel case du nom du champ.

`Cake\ORM\Entity::get($field)`

Ils reçoivent la valeur basique stockée dans le tableau `_properties` pour seul argument. 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. Par exemple :

```
namespace App\Model\Entity;

use Cake\ORM\Entity;

class Article extends Entity
{
    protected function _getTitle($title)
    {
        return ucwords($title);
    }
}
```

Les accesseurs seront utilisés quand vous récupérerez la propriété via une de ces deux manières :

```
echo $article->title;
echo $article->get('title');
```

Vous pouvez personnaliser la façon dont les propriétés sont récupérées/définies en définissant un mutateur :

`Cake\ORM\Entity::set`(*$field = null*, *$value = null*)

Les méthodes mutateurs doivent toujours retourner la valeur qui doit être stockée dans la propriété. Comme vous pouvez le voir au-dessus, vous pouvez aussi utiliser les mutateurs pour définir d'autres propriétés calculées. En faisant cela, attention à ne pas introduire de boucles, puisque CakePHP n'empêchera pas les méthodes mutateur de faire des boucles infinies. Les mutateurs vous permettent de convertir les propriétés lorsqu'elles sont définies ou de créer des données calculées. Les mutateurs et accesseurs sont appliqués quand les propriétés sont lues en utilisant la notation objet ou en utilisant get() et set(). Par exemple :

```
namespace App\Model\Entity;

use Cake\ORM\Entity;
use Cake\Utility\Text;

class Article extends Entity
{

    protected function _setTitle($title)
    {
        return Text::slug($title);
    }

}
```

Les mutateurs seront utilisés lorsque vous définirez une propriété via une de ces deux manières :

```
$user->title = 'foo' // slug sera aussi défini
$user->set('title', 'foo'); // slug sera aussi défini
```

Avertissement : Les accesseurs sont également appelés avant que l'entity ne soit persistée en 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 Propriétés Virtuelles

En définissant des accesseurs, vous pouvez fournir un accès aux champs/propriétés qui n'existent pas réellement. Par exemple si votre table users a `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->_properties['first_name'] . '  ' .
            $this->_properties['last_name'];
    }

}
```

Vous pouvez accéder aux propriétés virtuelles puisqu'elles existent sur l'entity. Le nom de la propriété sera la version en minuscule et en underscore de la méthode :

```
echo $user->full_name;
```

Souvenez-vous que les propriétés virtuelles ne peuvent pas être utilisées dans les finds. Si vous voulez que les propriétés virtuelles fassent parties des données affichées lorsque vous affichez les représentations JSON ou en tableau de vos entités, reportez-vous à la section *Montrer les Propriétés Virtuelles*.

Vérifier si une Entity a été Modifiée

`Cake\ORM\Entity::dirty`(*$field = null*, *$dirty = null*)

Vous pourriez vouloir écrire du code conditionnel basé sur si oui ou non les propriétés ont été modifiées 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->dirty('title');
```

Vous pouvez également marquer un champ comme ayant été modifié. C'est pratique lorsque vous ajoutez des données dans un tableau de propriétés :

```
// Ajoute un commentaire et marque le champ comme modifié.
$article->comments[] = $newComment;
$article->dirty('comments', true);
```

De plus, vous pouvez également baser votre code conditionnel sur les valeurs initiales des propriétés 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 une des propriétés de l'entity a été modifiée :

```
// Vérifier si l'entity a changé
$article->dirty();
```

Pour retirer le marquage dirty 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' => 'New Article'], ['markClean' => true]);
```

Pour récupérer la liste des propriétés *dirty* (modifiées) d'une `Entity`, vous pouvez utiliser la méthode `getDirty()` :

```
$dirtyFields = $entity->getDirty();
```

Nouveau dans la version 3.4.3 : `getDirty()` a été ajoutée.

Erreurs de Validation

`Cake\ORM\Entity::errors($field = null, $errors = null)`

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()` et `getError()` :

```
// Récupère toutes les erreurs
$errors = $user->getErrors();
// Avant 3.4.0
$errors = $user->errors();

// Récupère les erreurs pour un champ unique.
$errors = $user->getError('password');
// Avant 3.4.0
$errors = $user->errors('password');
```

Les méthodes `setErrors()` et `setError()` peuvent aussi être utilisées pour définir les erreurs sur une entity, facilitant les tests du code qui fonctionne avec les messages d'erreur :

```
$user->setError('password', ['Password is required.']);
$user->setErrors(['password' => ['Password is required'], 'username' => ['Username is␣
↪required']]);

// Avant 3.4.0
$user->errors('password', ['Password is required.']);
```

Assignement de Masse

Alors que la définition des propriétés des entities en masse est simple et pratique, elle peut créer des problèmes importants 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 (voir toutes) les colonnes. Quand vous utilisez les classes entity anonymes, CakePHP ne protège pas contre l'assignement en masse. Vous pouvez vous protéger de l'assignement de masse en utilisant *la commande bake* pour générer vos entities.

La propriété `_accessible` vous permet de fournir une liste des champs et si oui ou non ils peuvent être assignés en masse. 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 $_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;
```

(suite sur la page suivante)

(suite de la page précédente)

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

Note : Si la propriété `*` n'est pas définie, elle sera par défaut à `false`.

Eviter la Protection Contre l'Assignement de Masse

lors de la création d'un nouvelle entity via 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 l'Exécution

Vous pouvez modifier la liste des champs protégés à la volée en utilisant la méthode `accessible` :

```
// Rendre user_id accessible.
$article->accessible('user_id', true);

// Rendre title protégé.
$article->accessible('title', false);
```

Note : Modifier des champs accessibles agit seulement sur l'instance de la méthode sur laquelle il est appelé.

Lorsque vous utilisez les méthodes `newEntity()` et `patchEntity()` dans les objets `Table` vous avez également le contrôle sur la protection de masse. Référez vous à la section to the *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 :

```
$article->set($properties, ['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 `isNew()` en tant que setter :

```
$article->isNew(false);

$article->isNew(true);
```

Lazy Loading des Associations

Alors que les associations chargées en eager loading sont généralement la façon la plus efficace pour accéder à vos associations, il peut arriver que vous ayez besoin d'utiliser le lazy loading des données associées. Avant de voir comment utiliser le Lazy loading d'associations, nous devrions discuter des différences entre le chargement des associations eager et lazy :

Eager loading Le Eager loading utilise les joins (si 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 courant d'objets.

Lazy loading Le Lazy loading diffère le chargement des données de l'association jusqu'à ce que ce soit complètement nécessaire. Alors que ceci peut sauver du temps CPU car des données possiblement non utilisées ne sont pas hydratées dans les objets, cela peut résulter en 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 étant itérés.

Bien que le lazy loading n'est pas inclus dans l'ORM de CakePHP, vous pouvez utiliser un des plugins de la communauté. Nous recommandons le plugin LazyLoad [128]

Après avoir ajouté le plugin à votre entity, vous pourrez le faire avec ce qui suit :

```
$article = $this->Articles->findById($id);

// La propriété comments a été chargé en lazy
foreach ($article->comments as $comment) {
    echo $comment->body;
}
```

128. https://github.com/jeremyharris/cakephp-lazyload

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 la table et les objets entity.

Par exemple si nous avons un plugin SoftDeletable, il pourrait fournir un trait. Ce trait pourrait donner des méthodes pour rendre les entities comme "supprimé", la méthode `softDelete` pourrait 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 entity en l'intégrant et en l'incluant :

```
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, vous avez peut-être besoin de convertir des entities en tableaux ou en données JSON. CakePHP facilite cela :

```
// Obtenir un tableau.
// Les associations seront aussi converties avec 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 champ virtuel & caché sont utilisées. Les entities sont converties aussi de façon récursive en JSON. Cela signifie que si les entities et leurs associations sont chargées en eager loading, CakePHP va correctement gérer la conversion des données associées dans le bon format.

Montrer les Propriétés Virtuelles

Par défaut, les propriétés virtuelles ne sont pas exportées lors de la conversion des entities en tableaux ou JSON. Afin de montrer les propriétés virtuelles, vous devez les rendre visibles. Lors de la définition de votre classe entity, vous pouvez fournir une liste de propriétés virtuelles qui doivent être exposées :

```
namespace App\Model\Entity;

use Cake\ORM\Entity;

class User extends Entity
{

    protected $_virtual = ['full_name'];

}
```

Cette liste peut être modifiée à la volée en utilisant `virtualProperties` :

```
$user->virtualProperties(['full_name', 'is_admin']);
```

Cacher les Propriétés

Il arrive souvent que vous ne souhaitiez pas exporter certains champs dans des formats JSON ou tableau. Par exemple il n'est souvent pas sage de montrer les hashs de mot de passe ou les questions pour retrouver son compte. Lors de la définition d'une classe entity, définissez les propriétés qui doivent être cachées :

```
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 `hiddenProperties` :

```
$user->hiddenProperties(['password', 'recovery_question']);
```

Stocker des Types Complexes

Les méthodes « accesseurs » et « mutateurs » n'ont pas pour objectif de contenir de la logique pour sérialiser et désérialiser 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**

Alors que les objets table fournissent une abstraction autour d'un "dépôt" ou d'une collection d'objets, quand vous faîtes une requête pour des enregistrements individuels, vous récupérez des objets "entity". Cette section parle des différentes façons pour trouver et charger les entities, mais vous pouvez lire la section *Entities* pour plus d'informations sur les entities.

Débugger les Queries et les ResultSets

Etant donné que l'ORM retourne maintenant des Collections et des Entities, débugger ces objets peut être plus compliqué qu'avec les versions précédentes de CakePHP. Il y a maintenant 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($query->toArray())` Une façon facile de montrer chacun des résultats.
- `debug(json_encode($query, JSON_PRETTY_PRINT))` Résultats plus lisibles.
- `debug($query->first())` Montre les propriétés de la première entity que votre requête retournera.
- `debug((string)$query->first())` Montre les propriétés de la première entity que votre requête retournera en JSON.

Récupérer une Entity Unique avec une Clé Primaire

Cake\ORM\Table::**get**(*$id*, *$options = []*)

C'est souvent pratique de charger une entity unique à partir de la base de données quand vous modifiez ou visualisez les entities et leurs données liées. Vous pouvez faire ceci en utilisant `get()` :

```
// Dans un controller ou dans une méthode 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, ou 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 table.

// Utilise toute config de cache ou une instance de CacheEngine & une clé générée
$article = $articles->get($id, [
    'cache' => 'custom',
]);

// Utilise toute config de cache ou une instance de CacheEngine & une clé spécifique
$article = $articles->get($id, [
```

(suite sur la page suivante)

(suite de la page précédente)

```
    'cache' => 'custom', 'key' => 'mykey'
]);

// Désactive explicitement la mise en cache
$article = $articles->get($id, [
    'cache' => false
]);
```

En option, vous pouvez faire un `get()` d'une entity en utilisant *Méthodes Finder Personnalisées*. Par exemple vous souhaitez récupérer toutes les traductions pour une entity. Vous pouvez le faire en utilisant l'option `finder` :

```
$article = $articles->get($id, [
    'finder' => 'translations',
]);
```

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 facile et extensible pour trouver les données qui vous intéressent :

```
// Dans un controller ou dans une méthode table.

// Trouver tous les articles
$query = $articles->find('all');
```

La valeur retournée de toute méthode `find` est toujours un objet `Cake\ORM\Query`. La classe Query vous permet de redéfinir une requête plus tard 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 table.

// Trouver tous les articles.
// A ce niveau, la requête n'est pas lancée.
$query = $articles->find('all');

// L'itération va exécuter la requête.
foreach ($query as $row) {
}

// Appeler all() va exécuter la requête
// et retourne l'ensemble de résultats.
$results = $query->all();

// Once we have a result set we can get all the rows
$data = $results->toArray();

// Convertir la requête en tableau 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, d'ajouter des conditions supplémentaires, des limites, ou d'inclure des associations en

utilisant l'interface courante.

```
// Dans un controller ou dans une méthode table.
$query = $articles->find('all')
    ->where(['Articles.created >' => new DateTime('-10 days')])
    ->contain(['Comments', 'Authors'])
    ->limit(10);
```

Vous pouvez aussi fournir plusieurs options couramment utilisées avec `find()`. Ceci peut aider pour le test puisqu'il y a peu de méthodes à mocker :

```
// Dans un controller ou dans une méthode table.
$query = $articles->find('all', [
    'conditions' => ['Articles.created >' => new DateTime('-10 days')],
    'contain' => ['Authors', 'Comments'],
    'limit' => 10
]);
```

La liste d'options supportées par find() sont :

- `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. Charger seulement quelques champs peut faire que les entities se comportent de manière incorrecte.
- `group` ajoute une clause GROUP BY à votre requête. C'est utile quand vous utilisez les fonctions d'agrégation.
- `having` ajoute une clause HAVING à votre requête.
- `join` définit les jointures personnalisées supplémentaires.
- `order` ordonne l'ensemble des résultats.

Toute option qui n'est pas dans la liste sera passée aux écouteurs de beforeFind où ils peuvent être utilisés pour modifier l'objet requête. Vous pouvez utiliser la méthode `getOptions` sur un objet query pour récupérer les options utilisées. Alors que vous pouvez très facilement passer des objets requête à vos controllers, nous recommandons que vous fassiez plutôt des packages de vos requêtes en tant que *Méthodes Finder Personnalisées*. Utiliser des méthodes finder personnalisées va vous laisser réutiliser vos requêtes et faciliter les tests.

Par défaut, les requêtes et les ensembles de résultat seront retournés en objets *Entities*. Vous pouvez récupérer des tableaux basiques en désactivant l'hydratation :

```
$query->enableHydration(false);

// Avant 3.4.0
$query->hydrate(false);

// $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 à partir d'une query. Si la query n'a pas été exécutée, une clause `LIMIT 1` sera appliquée :

```
// Dans un controller ou dans une méthode table.
$query = $articles->find('all', [
    'order' => ['Article.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 chargez les entities avec leur clé primaire.

Note : La méthode `first()` va retourner `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 nombre de 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 des Enregistrements* pour l'utilisation supplémentaire de la méthode `count()`.

Trouver les Paires de Clé/Valeur

C'est souvent pratique pour générer un tableau associatif de données à partir des données de votre application. Par exemple, c'est très utile quand vous créez des elements <*select*>. CakePHP fournit une méthode simple à utiliser pour générer des "lists" 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 => 'First post',
    2 => 'Second article I wrote',
];
```

Avec aucune option supplémentaire, les clés de `$data` seront la clé primaire de votre table, alors que les valeurs seront le "displayField" (champAAfficher) de la table. Vous pouvez utiliser la méthode `setDisplayField()` sur un objet table pour configurer le champ à afficher sur une table :

```
class Articles extends Table
{

    public function initialize(array $config)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    {
        $this->setDisplayField('title');

        // Avant 3.4.0
        $this->displayField('title');
    }
}
```

Quand vous appelez `list`, vous pouvez configurer les champs utilisés pour la clé et la valeur avec respectivement les options `keyField` et `valueField` :

```
// Dans un controller ou dans une méthode de table.
$query = $articles->find('list', [
    'keyField' => 'slug',
    'valueField' => 'title'
]);
$data = $query->toArray();

// Les données ressemblent maintenant à
$data = [
    'first-post' => 'First post',
    'second-article-i-wrote' => 'Second article I wrote',
];
```

Les résultats peuvent être groupés en des ensembles imbriqués. C'est utile quand vous voulez des ensembles bucketed 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' => 'title',
    'groupField' => 'author_id'
]);
$data = $query->toArray();

// Les données ressemblent maintenant à
$data = [
    1 => [
        'first-post' => 'First post',
        'second-article-i-wrote' => 'Second article I wrote',
    ],
    2 => [
        // Plus de données.
    ]
];
```

Vous pouvez aussi créer une liste de données à partir des associations qui peuvent être atteintes avec les jointures :

```
$query = $articles->find('list', [
    'keyField' => 'id',
    'valueField' => 'author.name'
])->contain(['Authors']);
```

Personnaliser la Sortie Clé-Valeur

Enfin il est possible d'utiliser les closures pour accéder aux méthodes de mutation des entities dans vos finds list.

```
// Dasn votre Entity Authors, créez un champ virtuel à utiliser en tant que
champ à afficher:
protected function _getLabel()
{
    return $this->_properties['first_name'] . ' ' . $this->_properties['last_name']
      . ' / ' . __('User ID %s', $this->_properties['user_id']);
}
```

Cet exemple montre l'utilisation de la méthode accesseur `_getLabel()` à partir de l'entity Author.

```
// Dans vos finders/controller:
$query = $articles->find('list', [
    'keyField' => 'id',
    'valueField' => function ($article) {
        return $article->author->get('label');
    }
]);
```

Vous pouvez aussi récupérer le label dans la liste directement en utilisant.

```
// Dans AuthorsTable::initialize():
$this->setDisplayField('label'); // Va utiliser Author::_getLabel()
// Dans votre finders/controller:
$query = $authors->find('list'); // Va utiliser AuthorsTable::getDisplayField()
```

Trouver des Données Threaded

Le finder `find('threaded')` retourne les entities imbriquées qui sont threaded ensemble à travers un champ 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 matchent 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 sur lesquels le threading va être.

Astuce : Si vous devez gérer des données en arbre plus compliquées, pensez à utiliser *Tree* à la place.

Méthodes Finder Personnalisées

Les exemples ci-dessus montrent la façon d'utiliser les finders intégrés `all` et `list`. Cependant, il est possible et recommandé d'intégrer vos propres méthodes finder. Les méthodes finder sont idéales pour faire des packages de requêtes utilisées couramment, vous permettant de faire abstraction de détails de a requête en une méthode facile à utiliser. Les méthodes finder sont définies en créant les méthodes en suivant la convention `findFoo` où `Foo` est le nom du finder que vous souhaitez créer. Par exemple si nous voulons ajouter un finder à notre table articles pour trouver des articles publiés, nous ferions ce qui suit :

```
use Cake\ORM\Query;
use Cake\ORM\Table;

class ArticlesTable extends Table
{

    public function findPublished(Query $query, array $options)
    {
        $query->where([
            'Articles.published' => true,
            'Articles.moderated' => true
        ]);
        return $query;
    }

}

// Dans un controller ou dans une méthode table.
// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->get('Articles');
$query = $articles->find('published');
```

Les méthodes finder peuvent modifier la requête comme ceci, ou utiliser `$options` pour personnaliser l'opération finder avec la logique d'application concernée. Vous pouvez aussi "stack" les finders, vous permettant de faire des requêtes complexes sans efforts. En supposant que vous avez à la fois les finders "published" et "recent", vous pouvez faire ce qui suit :

```
// Dans un controller ou dans une méthode de table.
// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->get('Articles');
$query = $articles->find('published')->find('recent');
```

Alors que les exemples pour l'instant ont montré les méthodes finder sur les classes table, les méthodes finder peuvent aussi être définies sur les *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* pour modifier les résultats. Les fonctionnalités de map reduce remplacent le callback "afterFind" qu'on avait dans les versions précédentes de CakePHP.

Finders Dynamiques

L'ORM de CakePHP fournit des méthodes de finder construites dynamiquement qui vous permettent d'exprimer des requêtes simples sans aucun code supplémentaire. Par exemple si vous vouliez trouver un utilisateur selon son username, vous pourriez faire :

```
// Dans un controller
// Les deux appels suivants sont équivalents.
$query = $this->Users->findByUsername('joebob');
$query = $this->Users->findAllByUsername('joebob');

// Dans une méthode de table
// Prior to 3.6 use TableRegistry::get('Users')
$users = TableRegistry::getTableLocator()->get('Users');
// Les deux appels suivants sont équivalents.
$query = $users->findByUsername('joebob');
$query = $users->findAllByUsername('joebob');
```

Lors de l'utilisation de finders dynamiques, vous pouvez faire des contraintes sur plusieurs champs :

```
$query = $users->findAllByUsernameAndApproved('joebob', 1);
```

Vous pouvez aussi créer des conditions `OR` :

```
$query = $users->findAllByUsernameOrEmail('joebob', 'joe@example.com');
```

Alors que vous pouvez utiliser des conditions OR ou AND, vous ne pouvez pas combiner les deux dans un finder unique dynamique. Les autres options de requête comme `contain` ne sont aussi pas supportées avec les finders dynamiques. Vous devrez utiliser *Méthodes Finder Personnalisées* pour encapsuler plus de requêtes complexes. Dernier point, vous pouvez aussi combiner les finders dynamiques avec des finders personnalisés :

```
$query = $users->findTrollsByUsername('bro');
```

Ce qui est au-dessus se traduirait dans ce qui suit :

```
$users->find('trolls', [
    'conditions' => ['username' => 'bro']
]);
```

Une fois que vous avez un objet query à partir d'un finder dynamique, vous devrez appeler `first()` si vous souhaitez le premier résultat.

Note : Alors que les finders dynamiques facilitent la gestion des requêtes, ils entraînent des coûts de performance supplémentaires.

Récupérer les Données Associées

Quand vous voulez récupérer des données associées, ou filtrer selon les données associées, il y a deux façons :

— utiliser les fonctions query de l'ORM de CakePHP comme `contain()` et `matching()`
— utiliser les fonctions de jointures comme `innerJoin()`, `leftJoin()`, et `rightJoin()`

Vous pouvez utiliser `contain()` quand vous voulez charger le model primaire et ses données associées. Alors que `contain()` va vous laisser appliquer des conditions supplémentaires aux associations chargées, vous ne pouvez pas donner des contraintes au model primaire selon les associations. Pour plus de détails sur `contain()`, consultez *Eager Loading des Associations Via Contain*.

Vous pouvez utiliser `matching()` quand vous souhaitez donner des contraintes au model primaire selon les associations. Par exemple, vous voulez charger tous les articles qui ont 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 charger dans vos résultats.

Chaque Eager loading évite plusieurs problèmes potentiels de chargement lors du lazy loading dans un ORM. Les requêtes générées par le eager loading peuvent augmenter l'impact des jointures, permettant de faire des requêtes plus efficaces. Dans CakePHP vous définissez des associations chargées en eager en utilisant la méthode "contain" :

```
// 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']);
```

Ce qui est au-dessus va charger les auteurs et commentaires liés pour chaque article de l'ensemble de résultats. Vous pouvez charger les associations imbriquées en utilisant les tableaux imbriqués pour définir les associations à charger :

```
$query = $articles->find()->contain([
    'Authors' => ['Addresses'], 'Comments' => ['Authors']
]);
```

D'une autre façon, vous pouvez exprimer des associations imbriquées en utilisant la notation par point :

```
$query = $articles->find()->contain([
    'Authors.Addresses',
    'Comments.Authors'
]);
```

Vous pouvez charger les associations en eager aussi profondément que vous le souhaitez :

```
$query = $products->find()->contain([
    'Shops.Cities.Countries',
    'Shops.Managers'
]);
```

Vous pouvez sélectionner des champs de toutes les associations en utilisant plusieurs appels à `contain()` :

```
$query = $this->find()->select([
    'Realestates.id',
    'Realestates.title',
    'Realestates.description'
])
->contain([
    'RealestateAttributes' => [
        'Attributes' => [
            'fields' => [
                // Les champs alias dans contain() doivent
                // inclure le préfixe du modèle à mapper correctement.
                'Attributes__name' => 'attr_name'
            ]
        ]
    ]
])
->contain([
    'RealestateAttributes' => [
        'fields' => [
            'RealestateAttributes.realestate_id',
            'RealestateAttributes.value'
        ]
    ]
])
->where($condition);
```

Si vous avez besoin de remettre les contain sur une requête, vous pouvez définir le second argument à `true` :

```
$query = $articles->find();
$query->contain(['Authors', 'Comments'], true);
```

Passer des Conditions à Contain

Avec l'utilisation de `contain()`, vous pouvez restreindre les données retournées par les associations et les filtrer par conditions :

```
// Dans un controller ou une méthode de table.

$query = $articles->find()->contain('Comments', function ($q) {
   return $q
        ->select(['body', 'author_id'])
        ->where(['Comments.approved' => true]);
});
```

Cela fonctionne aussi pour la pagination au niveau du Controller :

```
$this->paginate['contain'] = [
    'Comments' => function (\Cake\ORM\Query $query) {
        return $query->select(['body', 'author_id'])
        ->where(['Comments.approved' => true]);
    }
];
```

Note : Quand vous limitez les champs qui sont récupérés d'une association, vous **devez** vous assurer que les colonnes de clé étrangère soient sélectionnées. Ne pas sélectionner les champs de clé étrangère va entraîner la non présence des

données associées dans le résultat final.

Il est aussi possible de restreindre les associations imbriquées profondément en utilisant la notation par point :

```
$query = $articles->find()->contain([
    'Comments',
    'Authors.Profiles' => function ($q) {
        return $q->where(['Profiles.is_published' => true]);
    }
]);
```

Dans l'exemple ci-dessus, 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 méthodes de finder personnalisées 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 des enregistrements associés. Pour le reste des types d'association, vous pouvez utiliser chaque clause que l'objet query fournit.

Si vous devez prendre le contrôle total d'une requête qui est générée, vous pouvez appeler `contain()` pour ne pas ajouter les contraintes `foreignKey` à la requête générée. Dans ce cas, vous devez utiliser un tableau en passant `foreignKey` et `queryBuilder` :

```
$query = $articles->find()->contain([
    'Authors' => [
        'foreignKey' => false,
        'queryBuilder' => function ($q) {
            return $q->where(...); // Full conditions for filtering
        }
    ]
]);
```

Si vous avez limité les champs que vous chargez avec `select()` mais que vous souhaitez aussi charger les champs enlevés des associations avec contain, vous pouvez passer l'objet association à `select()` :

```
// Sélectionne id & title de articles, mais tous les champs enlevés pour Users.
$query = $articles->find()
    ->select(['id', 'title'])
    ->select($articles->Users)
    ->contain(['Users']);
```

D'une autre façon, si vous pouvez faire des associations multiples, vous pouvez utiliser `enableAutoFields()` :

```
// Sélectionne id & title de articles, mais tous les champs enlevés de
// Users, Comments et Tags.
$query->select(['id', 'title'])
    ->contain(['Comments', 'Tags'])
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ->enableAutoFields(true) // Avant 3.4.0 utilisez autoFields(true)
    ->contain(['Users' => function($q) {
        return $q->autoFields(true);
    }]);
```

Nouveau dans la version 3.1 : La sélection des colonnes via un objet association a été ajouté dans 3.1

Ordonner les Associations Contain

Quand vous chargez des associations HasMany et BelongsToMany, vous pouvez utiliser l'option `sort` pour ordonner les données dans ces associations :

```
$query->contain([
    'Comments' => [
        'sort' => ['Comments.created' => 'DESC']
    ]
]);
```

Filtrer par les Données Associées Via Matching et Joins

Un cas de requête couramment fait avec les associations est de trouver les enregistrements qui "matchent" les données associées spécifiques. Par exemple si vous avez "Articles belongsToMany Tags", vous aurez probablement envie de trouver les Articles qui ont le tag CakePHP. C'est extrêmement simple à faire avec l'ORM de CakePHP :

```
// Dans un controller ou table de méthode.

$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 avec les articles récemment publiés en utilisant ce qui suit :

```
$query = $authors->find();
$query->matching('Articles', function ($q) {
    return $q->where(['Articles.created >=' => new DateTime('-10 days')]);
});
```

Filtrer des associations imbriquées est étonnamment facile, et la syntaxe doit déjà vous être familière :

```
// Dans un controller ou une table de méthode.
$query = $products->find()->matching(
    'Shops.Cities.Countries', function ($q) {
        return $q->where(['Countries.name' => 'Japan']);
    }
);

// Récupère les articles uniques qui étaient commentés par 'markstory'
// en utilisant la variable passée
// Les chemins avec points doivent être utilisés plutôt que les appels
// imbriqués de matching()
$username = 'markstory';
```

(suite sur la page suivante)

(suite de la page précédente)

```
$query = $articles->find()->matching('Comments.Users', function ($q) use ($username) {
    return $q->where(['username' => $username]);
});
```

Note : Comme cette fonction va créer un `INNER JOIN`, vous pouvez appeler `distinct` sur le find de la requête puisque vous aurez des lignes dupliquées si les conditions ne les excluent pas déjà. Ceci peut être le cas, par exemple, quand les mêmes utilisateurs commentent plus d'une fois un article unique.

Les données des associations qui sont "matchés" (appairés) 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 recevoir 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 vouliez utiliser `matching()` mais que vous n'êtes pas intéressé par le chargement des champs dans un ensemble de résultats. 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' => 'Japan']);
    }
);
```

De même, la seule différence est qu'aucune colonne supplémentaire ne sera ajoutée à l'ensemble de résultats et aucune propriété `_matchingData` ne sera définie.

Nouveau dans la version 3.1 : Query : :innerJoinWith() a été ajoutée dans 3.1

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' => 'boring']);
    });
```

L'exemple ci-dessus va trouver tous les articles qui n'étaient pas taggés avec le mot `boring`. Vous pouvez aussi utiliser cette méthode avec les associations HasMany. Vous pouvez, par exemple, trouver tous les auteurs qui n'ont 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 les associations profondes. 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 avec aucun commentaire satisfont aussi la condition du dessus, vous pouvez combiner `matching()` et `notMatching()` dans la même requête. L'exemple suivant va trouver 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 find puisque sinon vous allez avoir des lignes dupliquées.

Gardez à l'esprit que le contraire de la fonction `matching()`, `notMatching()` ne va pas ajouter toutes les données à la propriété `_matchingData` dans les résultats.

Nouveau dans la version 3.1 : Query : :notMatching() a été ajoutée dans 3.1

Utiliser leftJoinWith

Dans certaines situations, vous aurez à calculer un résultat selon une association, sans avoir à charger tous les enregistrements. Par exemple, si vous voulez charger le nombre total de commentaires qu'un article a, ainsi que toutes les 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); // Avant 3.4.0 utilisez autoFields(true)
```

Le résultat de la requête ci-dessus va contenir les données de l'article et la propriété `total_comments` pour chacun d'eux.

`leftJoinWith()` peut aussi être utilisée avec des associations profondes. C'est utile par exemple pour rapporter le nombre d'articles taggés par l'auteur 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' => 'awesome']);
    })
    ->group(['Authors.id'])
    ->enableAutoFields(true); // Avant 3.4.0 utilisez autoFields(true)
```

Cette fonction ne va charger aucune colonne des associations spécifiées dans l'ensemble de résultats.

Nouveau dans la version 3.1 : Query : :leftJoinWith() a été ajoutée dans 3.1

Changer les Stratégies de Récupération

Comme vous le savez peut-être déjà, les associations `belongsTo` et `hasOne` sont chargées en utilisant un `JOIN` dans la requête du finder principal. Bien que ceci améliore la requête et la vitesse de récupération des données et permet de créer des conditions plus parlantes lors de la récupération des données, cela peut devenir un problème quand vous devez appliquer certaines clauses à la requête finder pour l'association, comme `order()` ou `limit()`.

Par exemple, si vous souhaitez récupérer le premier commentaire d'un article en association :

```
$articles->hasOne('FirstComment', [
     'className' => 'Comments',
     'foreignKey' => 'article_id'
]);
```

Afin de récupérer correctement les données de cette association, nous devrons dire à la requête d'utiliser la stratégie `select`, puisque nous voulons trier selon une colonne en particulier :

```
$query = $articles->find()->contain([
    'FirstComment' => [
            'strategy' => 'select',
            'queryBuilder' => function ($q) {
                return $q->order(['FirstComment.created' =>'ASC'])->limit(1);
            }
    ]
]);
```

Changer la stratégie de façon dynamique de cette façon va seulement l'appliquer pour une requête spécifique. Si vous souhaitez rendre le changement de stratégie permanent, vous pouvez faire :

```
$articles->FirstComment->setStrategy('select');

// Avant 3.4.0
$articles->FirstComment->strategy('select');
```

Utiliser la stratégie `select` est aussi une bonne façon de faire des associations avec des tables d'une autre base de données, puisqu'il ne serait pas possible de récupérer des enregistrements en utilisant `joins`.

Récupération Avec la Stratégie de Sous-Requête

Avec la taille de vos tables qui grandit, la récupération des associations peut devenir lente, spécialement si vous faîtes des 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 des bases de données qui limitent le nombre de paramètres liés par requête, comme le **Serveur Microsoft SQL**.

Vous pouvez aussi rendre la stratégie pour les associations permanente en faisant :

```
$articles->Comments->setStrategy('subquery');

// Avant 3.4.0
$articles->Comments->strategy('subquery');
```

Lazy loading des Associations

Bien que CakePHP facilite le chargement en eager de vos associations, il y a des cas où vous devrez charger en lazy les associations. Vous devez vous référer aux sections *Lazy Loading des Associations* et *Chargement d'Associations Additionnelles* pour plus d'informations.

Travailler avec des Ensembles de Résultat

Une fois qu'une requête est exécutée avec `all()`, vous récupèrerez une instance de `Cake\ORM\ResultSet`. Cet objet permet de manipuler les données résultantes de vos requêtes. Comme les objets Query, les ensembles de résultats sont une *Collection* et vous pouvez utiliser toute méthode de collection sur des objets ResultSet.

Les objets ResultSet vont charger lazily les lignes à partir de la requête préparée sous-jacente. Par défaut, les résultats seront mis en mémoire vous permettant d'itérer un ensemble de résultats plusieurs fois, ou de mettre en cache et d'itérer les résultats. Si vous devez travailler sur un ensemble de données qui ne rentre pas dans la mémoire, vous pouvez désactiver la mise en mémoire sur la requête pour faire un stream des résultats :

```
$query->enableBufferedResults(false);

// Avant 3.4.0
$query->bufferResults(false);
```

Stopper la mise en mémoire tampon nécessite quelques mises en garde :

1. Vous ne pourrez plus itérer un ensemble de résultats plus d'une fois.
2. Vous ne pourrez plus aussi itérer et mettre en cache les résultats.

3. La mise en mémoire tampon ne peut pas être désactivé 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 pour que les requêtes dépendantes puissent être générées.

Avertissement : Les résultats de streaming alloueront toujours l'espace mémoire nécessaire pour les résultats complets lorsque vous utilisez PostgreSQL et SQL Server. Ceci est dû à des limitations dans PDO.

Les ensembles de résultat vous permettent de mettre en cache/serializer ou d'encoder en JSON les résultats pour les résultats d'une API :

```
// Dans un controller ou une méthode de table.
$results = $query->all();

// Serializé
$serialized = serialize($results);

// Json
$json = json_encode($results);
```

Les sérialisations et encodage en JSON des ensembles de résultats fonctionne comme vous pouvez vous attendre. Les données sérialisées peuvent être désérializées en un ensemble de résultats de travail. Convertir en JSON garde les configurations de champ caché & virtuel sur tous les objets entity dans un ensemble de résultat.

En plus de faciliter la sérialisation, les ensembles de résultats sont un objet "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.
// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->get('Articles');
$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 des méthodes de collection utilisées avec des ensembles de données :

```
// 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
// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->get('Articles');
$results = $articles->find()->contain(['Authors'])->all();

$authorList = $results->combine('id', 'author.name');
```

Le chapitre *Collections* comporte plus de détails sur ce qui peut être fait avec les ensembles de résultat en utilisant les fonctionnalités des collections.

Récupérer le Premier & Dernier enregistrement à partir d'un ResultSet

Vous pouvez utiliser les méthodes `first()` et `last()` pour récupérer les enregistrements respectifs à partir d'un ensemble de résultats :

```
$result = $articles->find('all')->all();

// Récupère le premier et/ou le dernier résultat.
$row = $result->first();
$row = $result->last();
```

Récupérer un Index arbitraire à partir 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 Query ou un ResultSet est vide

Vous pouvez utiliser la méthode `isEmpty()` sur un objet Query ou ResultSet pour voir s'il contient au moins une colonne. Appeler `isEmpty()` sur un objet Query va évaluer la requête :

```
// VérifieCheck une requête.
$query->isEmpty();

// Vérifie les résultats.
$results = $query->all();
$results->isEmpty();
```

Chargement d'Associations Additionnelles

Une fois que vous avez créé un ensemble de résultats, vous pourriez vouloir charger en eager des associations additionnelles. C'est le moment idéal pour charger des données. Vous pouvez charger des associations additionnelles 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'entites.

Modifier les Résultats avec Map/Reduce

La plupart du temps, les opérations `find` nécessitent un traitement postérieur des données qui se trouvent dans la base de données. Alors que les méthodes `getter` des `entities` peuvent s'occuper de la plupart de la génération de propriété virtuelle ou un formatage de données spéciales, parfois vous devez changer la structure des données d'une façon plus fondamentale.

Pour ces cas, l'objet `Query` offre la méthode `mapReduce()`, qui est une façon de traiter les résultats une fois qu'ils ont été récupérés dans la base de données.

Un exemple habituel de changement de structure de données est le groupement de résultats basé sur certaines conditions. Pour cette tâche, nous pouvons utiliser la fonction `mapReduce()`. Nous avons besoin de deux fonctions appelables `$mapper` et `$reducer`. La callable `$mapper` reçoit le résultat courant de la base de données en premier argument, la clé d'itération en second paramètre et finalement elle reçoit une instance de la routine `MapReduce` qu'elle lance :

```
$mapper = function ($article, $key, $mapReduce) {
    $status = 'published';
    if ($article->isDraft() || $article->isInReview()) {
        $status = 'unpublished';
    }
    $mapReduce->emitIntermediate($article, $status);
};
```

Dans l'exemple ci-dessus, `$mapper` calcule le statut d'un article, soit publié (published) soit non publié (unpublished), ensuite il appelle `emitIntermediate()` sur l'instance `MapReduce`. Cette méthode stocke l'article dans la liste des articles avec pour label soit publié (published) ou non publié (unpublished).

La prochaine étape dans le processus de map-reduce est de consolider les résultats finaux. Pour chaque statut créé dans le mapper, la fonction `$reducer` va être appelée donc vous pouvez faire des traitements supplémentaires. Cette fonction va recevoir la liste des articles dans un « bucket » particulier en premier paramètre, le nom du « bucket » dont il a besoin pour faire le traitement en second paramètre, et encore une fois, comme dans la fonction `mapper()`, l'instance de la routine `MapReduce` en troisième paramètre. Dans notre exemple, nous n'avons pas fait de traitement supplémentaire, donc nous avons juste `emit()` les résultats finaux :

```
$reducer = function ($articles, $status, $mapReduce) {
    $mapReduce->emit($articles, $status);
};
```

Finalement, nous pouvons mettre ces deux fonctions ensemble pour faire le groupement :

```
$articlesByStatus = $articles->find()
    ->where(['author_id' => 1])
    ->mapReduce($mapper, $reducer);

foreach ($articlesByStatus as $status => $articles) {
    echo sprintf("There are %d %s articles", count($articles), $status);
}
```

Ce qui est au-dessus va afficher les lignes suivantes :

```
There are 4 published articles
There are 5 unpublished articles
```

Bien sûr, ceci est un exemple simple qui pourrait être solutionné 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.

Calculer les mots mentionnés le plus souvent, où les articles contiennent l'information 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 `bucket` où chaque id d'article sera stocké. Maintenant réduisons nos résultats pour extraire seulement le compte :

```
$reducer = function ($occurrences, $word, $mapReduce) {
    $mapReduce->emit(count($occurrences), $word);
}
```

Finalement, nous mettons tout ensemble :

```
$wordCount = $articles->find()
    ->where(['published' => true])
    ->andWhere(['published_date >=' => new DateTime('2014-01-01')])
    ->enableHydrate(false) // Avant 3.4.0 utilisez hydrate(false)
    ->mapReduce($mapper, $reducer)
    ->toArray();
```

Ceci pourrait retourner un tableau très grand si nous ne nettoyons pas les mots interdits, mais il pourrait ressembler à ceci :

```
[
    'cakephp' => 100,
    'awesome' => 39,
    'impressive' => 57,
    'outstanding' => 10,
    'mind-blowing' => 83
]
```

Un dernier exemple et vous serez un expert de map-reduce. Imaginez que vous avez une table de `friends` et que vous souhaitiez trouver les « fake friends » dans notre base de données ou, autrement dit, les 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],
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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;
        }
    }

    if ($fakeFriends) {
        $mr->emit($fakeFriends, $user);
    }
};
```

Et nous fournissons nos fonctions à la requête :

```
$fakeFriends = $friends->find()
    ->enableHydrate(false) // Avant 3.4.0 utilisez hydrate(false)
    ->mapReduce($mapper, $reducer)
    ->toArray();
```

Ceci retournerait un tableau similaire à ceci :

```
[
    1 => [2, 4],
    3 => [6]
    ...
]
```

Les tableaux résultants signifient, par exemple, que l'utilisateur avec l'id `1` suit les utilisateurs `2` and `4`, mais ceux-ci ne suivent pas `1` de leur côté.

Stacking Multiple Operations

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 à 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)
{
    // Same as in the common words example in the previous section
    $mapper = ...;
    $reducer = ...;
    return $query->mapReduce($mapper, $reducer);
}

$commonWords = $articles
    ->find('commonWords')
    ->find('published')
    ->find('recent');
```

En plus, il est aussi possible d'empiler plus d'une opération `mapReduce` pour une requête unique. Par exemple, si nous souhaitons avoir les mots les plus couramment utilisés pour les articles, mais ensuite les filtrer pour seulement retourner 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);
```

Retirer Toutes les Opérations Map-reduce Empilées

Dans les mêmes circonstances vous voulez modifier un objet `Query` pour que les opérations `mapReduce` ne soient pas exécutées du tout. Ceci peut être fait 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 *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 soit converties en entity, les règle de validation concernant le type de données et leur format peuvent être appliquées.
2. Avant que les données ne soient sauvegardées, les règles du domaine ou de l'application peuvent être appliquées. Ces règles aident à garantir que les données de votre application restent cohérentes.

Valider les Données Avant de Construire les Entities

Durant la transformation des données en entities, vous pouvez valider les données. La validation des données vous permet de vérifier le type, la forme et la taille des données. Par défaut les données requêtées seront validées avant qu'elles ne soient converties en entities. Si une des règles de validation échoue, l'entity retournée contiendra des erreurs. Les champs avec des erreurs ne seront pas présents dans l'entity retournée :

```
$article = $articles->newEntity($this->request->getData());
if ($article->errors()) {
    // validation de l'entity a échouée.
}
```

Nouveau dans la version 3.4.0 : La méthode `getErrors()` a été ajoutée.

Quand vous construisez une entity avec la validation activée, les choses suivantes vont se produire :

1. L'objet validator est créé.
2. Les providers de validation `table` et `default` sont attachés.
3. La méthode de validation nommée est appelée. Par exemple, `validationDefault()`.
4. L'event `Model.buildValidator` va être déclenché.
5. Les données Requêtées vont être validées.
6. Les données Requêtées vont être castées en types qui correspondent aux types de colonne.
7. Les erreurs vont être définies dans l'entity.
8. Les données valides vont être définies dans l'entity, alors que les champs qui échouent la validation seront laissés de côté.

Si vous voulez désactiver la validation lors de la conversion des données requêtées, définissez l'option `validate` à false :

```
$article = $articles->newEntity(
    $this->request->data,
    ['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 aide à trouver les données qui doivent être validées en correspondance avec l'endroit où elles seront sauvegardées.

Pour créer un objet validation 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
            ->requirePresence('title', 'create')
            ->notEmpty('title');

        $validator
            ->allowEmpty('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 validation sont principalement conçus pour être utilisés pour la validation des données provenant d'utilisateurs, par exemple les formulaires ou n'importe quelles données postées.

Utiliser un Ensemble de Validation Différent

En plus de désactiver la validation, vous pouvez choisir l'ensemble de règles de validation que vous souhaitez appliquer :

```
$article = $articles->newEntity(
    $this->request->data,
    ['validate' => 'update']
);
```

Ce qui est au-dessus va appeler la méthode `validationUpdate()` sur l'instance table pour construire les règles requises. Par défaut la méthode `validationDefault()` sera utilisée. Un exemple de méthode de validator pour notre Table articles serait :

```
class ArticlesTable extends Table
{
    public function validationUpdate($validator)
    {
        $validator
```

(suite sur la page suivante)

(suite de la page précédente)

```
            ->add('title', 'notEmpty', [
                'rule' => 'notEmpty',
                'message' => __('Vous devez fournir un titre'),
            ])
            ->add('body', 'notEmpty', [
                'rule' => 'notEmpty',
                'message' => __('un corps 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 règle de validation.

Utiliser un Ensemble de Validation Différent pour les Associations

Les ensembles de validation peuvent également être définis par association. Lorsque vous utilisez les méthodes `newEntity()` ou `patchEntity()`, vous pouvez passer des options supplémentaires à chaque association qui doit être convertie :

```
$data = [
     'title' => 'My title',
     'body' => 'The text',
     'user_id' => 1,
     'user' => [
         'username' => 'mark'
     ],
     'comments' => [
         ['body' => 'First comment'],
         ['body' => 'Second comment'],
     ]
 ];

 $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, il est facile de diviser leur process de construction en de petites étapes réutilisables :

```
// UsersTable.php

public function validationDefault(Validator $validator)
{
    $validator->notEmpty('username');
    $validator->notEmpty('password');
```

(suite sur la page suivante)

(suite de la page précédente)

```
    $validator->add('email', 'valid-email', ['rule' => 'email']);
    ...

    return $validator;
}

public function validationeHardened(Validator $validator)
{
    $validator = $this->validationeDefault($validator);

    $validator->add('password', 'length', ['rule' => ['lengthBetween', 8, 100]]);
    return $validator;
}
```

En prenant en compte la configuration ci-dessus, lors de l'utilisation de l'ensemble de validation `hardened`, il contiendra également les règles de l'ensemble `default`.

Validation Providers

Les règles de validation peuvent utiliser les fonctions définies sur tout provider connu. Par défaut, CakePHP définit quelques providers :

1. Les méthodes sur la classe table, ou ses behaviors sont disponible sur le provider `table`.
2. La classe de `Validation\Validation` du coeur est configurée avec le provider `default`.

Quand une règle de validation est créée, vous pouvez nommer 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
{

    public function validationDefault(Validator $validator)
    {
        $validator
            ->add('role', 'validRole', [
                'rule' => 'isValidRole',
                'message' => __('Vous devez fournir un rôle valide'),
                'provider' => 'table',
            ]);
        return $validator;
    }

    public function isValidRole($value, array $context)
    {
        return in_array($value, ['admin', 'editor', 'author'], true);
    }

}
```

Vous pouvez également utiliser des closures en tant que règle de validation :

```
$validator->add('name', 'myRule', [
    'rule' => function ($data, $provider) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        if ($data > 1) {
            return true;
        }
        return 'Valeur incorrecte.';
    }
]);
```

Les méthodes de validation peuvent renvoyer des messages lorsqu'elles échouent. C'est un moyen simple de créer des messages d'erreur dynamiques basés sur la valeur fournie.

Récupérer des Validators depuis les Tables

Une fois que vous avez créé quelques ensembles de validation dans votre classe table, vous pouvez récupérer l'objet résultant via son nom :

```
$defaultValidator = $usersTable->validator('default');

$hardenedValidator = $usersTable->validator('hardened');
```

Obsolète depuis la version 3.5.0 : `validator()` est dépréciée. Utilisez `getValidator()` à la place.

Classe Validator par Défault

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 table :

```
// Dans votre class Table
public function initialize(array $config)
{
    $this->_validatorClass = '\FullyNamespaced\Custom\Validator';
}
```

Appliquer des Règles pour l'Application

Alors qu'une validation basique des données est faite quand *les données requêtées sont converties en entities*, de nombreuses applications ont aussi d'autres validations plus complexes qui doivent être appliquées seulement après qu'une validation basique a été terminée.

Alors que la validation s'assure que le formulaire ou la syntaxe de vos données sont corrects, les règles s'attellent à comparer les données avec un état existant de votre application et/ou de votre réseau.

Ces types de règles sont souvent appelées "règles de domaine" ou "règles de l'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.
- Etats de transition ou étapes de flux de travail, par exemple pour mettre à jour un statut de facture.
- Eviter la modification ou la suppression soft d'articles.
- 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 la Table.

Créer un Vérificateur de Règles

Les classes de vérificateur de Règles sont généralement définies par la méthode `buildRules()` dans votre classe de table. Les behaviors et les autres souscripteurs d'event peuvent utiliser l'event `Model.buildRules` pour ajouter des règles au vérificateur pour une classe de Table donnée :

```
use Cake\ORM\RulesChecker;

// Dans une classe de table
public function buildRules(RulesChecker $rules)
{
    // Ajoute une règle qui est appliquée pour la création et la mise à jour d
↪'opérations
    $rules->add(function ($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');

    return $rules;
}
```

Vos fonctions de règles ont pour paramètres l'Entity à vérifier et un tableau d'options. Le tableau d'options va contenir `errorField`, `message` et `repository`. L'option `repository` va contenir la classe de table sur laquelle les règles sont attachées. Comme les règles acceptent tout `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 le champ pour lequel la règle est faite, et le message d'erreur en options :

```
$rules->add([$this, 'isValidState'], 'validState', [
    'errorField' => 'status',
    'message' => 'Cette facture ne peut pas être déplacée pour ce statut.'
]);
```

L'erreur sera visible lors de l'appel à la méthode `errors()` dans l'entity :

```
$entity->errors(); // Contient les messages d'erreur des règles du domaine
```

Créer des Règles de Champ Unique

Comme les règles uniques sont couramment utilisées, CakePHP inclut une classe de Règle simple qui vous permet de définir des ensembles de champ unique :

```
use Cake\ORM\Rule\IsUnique;

// Un champ unique.
$rules->add($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. Cela signifie que définir `$user->account->id` ne va pas déclencher la règle ci-dessus.

Règles des Clés Etrangères

Alors que vous pourriez compter sur les erreurs de la base de données pour imposer des contraintes, utiliser des règles peut vous aider à 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 il faut vérifier l'existence dans la table liée doivent faire parti de la clé primaire.

Vous pouvez forcer `existsIn` à passer quand des parties qui peuvent être nulles de votre clé étrangère composite sont nulles :

```
// Example: Une clé primaire composée dans NodesTable est (id, site_id).
// Un "Node" peut faire référence à un parent Node mais ce n'est pas obligatoire.
// Dans un cas d'utilisation, parent_id est null.
// Nous permettons à cette règle de passer, même si les champs qui sont nullable,␣
↪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 permettent à plusieurs valeurs null d'exister car `NULL` n'est pas égal à lui même. Même si autoriser plusieurs valeurs null est le comportement par défaut de CakePHP, vous pouvez inclure des valeurs null dans vos validations en utilisant `allowMultipleNulls` :

```
// Seulement une valeur null peut exister dans `parent_id` et `site_id`
$rules->add($rules->existsIn(
    ['parent_id', 'site_id'],
    'ParentNodes',
    ['allowMultipleNulls' => false]
));
```

Nouveau dans la version 3.3.0 : Les options `allowNullableNulls` et `allowMultipleNulls` ont été ajoutées.

Règles sur le Nombre de Valeurs d'une Association

Si vous devez valider qu'une propriété ou une association contient un bon 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 pouvez avoir seulement 5 tags
↪'));
```

Quand vous définissez des règles qui concernent le nombre, le troisième paramètre vous permet de définir l'opérateur de comparaison à utiliser. ==, >=, <=, >, <, and != sont les opérateurs acceptés. Pour vous assurer qu'un nombre 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 moins 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'));
```

Nouveau dans la version 3.3.0 : La méthode `validCount()` a été ajoutée dans la version 3.3.0.

Utiliser les Méthodes Entity en tant que Règles

Vous pouvez utiliser les méthodes entity en tant que règles de domaine :

```
$rules->add(function ($entity, $options) {
    return $entity->isOkLooking();
}, 'ruleName');
```

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)
{
    $rules->add(new IsUniqueWithNulls(['parent_id', 'instance_id', 'name']),
↪'uniqueNamePerParent', [
```

(suite sur la page suivante)

(suite de la page précédente)

```
        'errorField' => 'name',
        'message' => 'Name must be unique per parent.'
    ]);
    return $rules;
}
```

Regardez les règles du coeur pour plus d'informations 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)
    {
        // Do work
        return false;
    }
}

// Ajoute la règle personnalisée
use App\Model\Rule\CustomRule;

$rules->add(new CustomRule(...), 'ruleName');
```

En ajoutant des classes de règle personnalisée, vous pouvez garder votre code DRY et faciliter le test des règles de votre domaine.

Désactiver les Règles

Quand vous sauvegardez une entity, vous pouvez désactiver les règles si cela est nécessaire :

```
$articles->save($article, ['checkRules' => false]);
```

Validation vs. Application Rules

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 objectif d'opérer d'une façon stateless. Elles permettent de s'assurer que la forme, les types de données et le format des données sont corrects.

La seconde couche sont les règles d'application. Les règles d'application permettent de vérifier les propriétés stateful de vos entities. Par exemple, les règles de validation peuvent permettre de s'assurer qu'une adresse email est valide, alors 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 via l'objet `Validator` lors de l'appel à `newEntity()` ou `patchEntity()` :

```
$validatedEntity = $articlesTable->newEntity(
    $unsafeData,
    ['validate' => 'customName']
);
$validatedEntity = $articlesTable->patchEntity(
    $entity,
    $unsafeData,
    ['validate' => 'customName']
);
```

Dans l'exemple ci-dessus, nous allons utiliser un validateur "custom", qui est défini en utilisant la méthode `validationCustomName()` :

```
public function validationCustomName($validator)
{
    $validator->add(...);
    return $validator;
}
```

La validation fait l'hypothèse que les chaînes de caractères et les tableaux sont passés puisque c'est ce qui est reçu par 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 mot de passe ne sont pas égaux',
    ]);

    ...
    return $validator;
}
```

La validation **n'est pas** déclenchée lorsqu'une propriété est définie directement dans 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 fois que `save()` ou `delete()` sont appelées :

```
// Dans src/Model/Table/UsersTable.php
public function buildRules(RulesChecker $rules)
{
    $rules->add($rules->isUnique(['email']));
    return $rules;
}

// Autre part dans le code de votre application
$userEntity->email = 'a@duplicated.email';
$usersTable->save($userEntity); // Retourne false
```

Alors que la validation est conçue pour les données provenant directement d'utilisateurs, les règles d'application sont spécifiques aux transitions de données générées à l'intérieur de l'application :

```
// Dans src/Model/Table/OrdersTable.php
public function buildRules(RulesChecker $rules)
{
    $check = function ($order) {
        return $order->price < 100 && $order->shipping_mode === 'free';
    };
    $rules->add($check, [
        'errorField' => 'shipping_mode',
        'message' => 'Pas de frais de port gratuit pour une commande de moins de 100!'
    ]);
    return $rules;
}

// Autre part dans le code de l'application
$order->price = 50;
$order->shipping_mode = 'free';
$ordersTable->save($order); // Retourne false
```

Utiliser la Validation en tant que Règle d'Application

Dans certaines situations, vous voudrez peut-être lancer les mêmes routines pour des données générées à la fois par un utilisateur et à 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->add('email', 'valid', [
        'rule' => 'email',
        'message' => 'Email invalide'
    ]);
    ...
    return $validator;
}

public function buildRules(RulesChecker $rules)
{
    // Ajoute des règles de  validation
    $rules->add(function ($entity) {
        $data = $entity->extract($this->schema()->columns(), true);
        $validator = $this->validator('default');
        // Prior to 3.9 use $validator->errors()
        $errors = $validator->validate($data, $entity->isNew());
        $entity->errors($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->errors('email'); // Email invalide
```

le même résultat est attendu lors de l'utilisation de `newEntity()` ou `patchEntity()` :

```
$userEntity = $usersTable->newEntity(['email' => 'Pas un email!!']);
$userEntity->errors('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 Enregistrement des Données

Les applications ont habituellement deux façons d'enregistrer les données. La première est évidemment via des formulaires web et l'autre en générant ou modifiant 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\TableRegistry;

// Prior to 3.6 use TableRegistry::get('Articles')
$articlesTable = TableRegistry::getTableLocator()->get('Articles');
$article = $articlesTable->newEntity();

$article->title = 'A New 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 mise à jour est aussi simple et la méthode `save()` sert également ce but :

```
use Cake\ORM\TableRegistry;

// Prior to 3.6 use TableRegistry::get('Articles')
$articlesTable = TableRegistry::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 un ajout ou une mise à jour en se basant sur le résultat de la méthode `isNew()`. Les entities qui sont récupérées via `get()` ou `find()` renverrons toujours `false` lorsque la méthode `isNew()` est appelée sur eux.

Enregistrements avec Associations

Par défaut, la méthode `save()` ne sauvegardera qu'un seul niveau d'association :

```
// Prior to 3.6 use TableRegistry::get('Articles')
$articlesTable = TableRegistry::getTableLocator()->get('Articles');
$author = $articlesTable->Authors->findByUserName('mark')->first();

$article = $articlesTable->newEntity();
$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->newEntity();
$firstComment->body = 'Un super article';

$secondComment = $articlesTable->Comments->newEntity();
$secondComment->body = 'J aime lire ceci!';

$tag1 = $articlesTable->Tags->findByName('cakephp')->first();
$tag2 = $articlesTable->Tags->newEntity();
$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->newEntity();
$tag2->name = 'Génial';

$articlesTable->Tags->link($article, [$tag1, $tag2]);
```

Sauvegarder des Données dans la Table de Jointure

L'enregistrement de données dans la table de jointure est réalisé en utilisant la propriété spéciale `_joinData`. Cette propriété doit être une instance d'`Entity` de la table de jointure :

```
// Lie les enregistrements pour la première fois.
$tag1 = $articlesTable->Tags->findByName('cakephp')->first();
$tag1->_joinData = $articlesTable->ArticlesTags->newEntity();
$tag1->_joinData->tagComment = 'Je pense que cela est lié à CakePHP';

$articlesTable->Tags->link($article, [$tag1]);

// Mise à jour d'une association existante.
$article = $articlesTable->get(1, ['contain' => ['Tags']]);
$article->tags[0]->_joinData->tagComment = 'Fresh comment.'

// Nécessaire car nous changeons une propriété directement
$article->dirty('tags', true);

$articlesTable->save($article, ['associated' => ['Tags']]);
```

Vous pouvez aussi créer/mettre à jour les informations de la table jointe quand vous utilisez `newEntity()` ou `patchEntity()`. Vos données POST devraient ressembler à ceci :

```
$data = [
    'title' => 'My great blog post',
    'body' => 'Some content that goes on for a bit.',
    'tags' => [
        [
            'id' => 10,
            '_joinData' => [
                'tagComment' => 'Great article!',
            ]
        ],
    ]
];
$articlesTable->newEntity($data, ['associated' => ['Tags']]);
```

Délier les Enregistrements Many To Many

Délier des enregistrements Many to Many (plusieurs à plusieurs) est réalisable via la méthode `unlink()` :

```
$tags = $articlesTable
    ->Tags
    ->find()
    ->where(['name IN' => ['cakephp', 'awesome']])
    ->toArray();

$articlesTable->Tags->unlink($article, $tags);
```

Lors de la modification d'enregistrements en définissant ou modifiant directement leurs propriétés il n'y aura pas de validation, ce qui est problématique pour l'acceptation de données de formulaire. La section suivante va vous expliquer 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 sauvegarder à nouveau les données dans la base de données, vous devrez convertir les données requêtées (qui se trouvent dans $this->request->getData()) à partir du format de tableau qui se trouvent dans la requête, et les entities que l'ORM utilise. La classe Table facilite la conversion d'une ou de plusieurs entities à partir des données requêtées. Vous pouvez convertir une entity unique en utilisant :

```
// Dans un controller.

// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->get('Articles');
// Valide et convertit en un objet Entity
$entity = $articles->newEntity($this->request->getData());
```

Les données requêtées doivent suivre la structure de vos entities. Par exemple si vous avez un article qui appartient à un utilisateur, et si vous avez plusieurs commentaires, vos données requêtées devraient ressembler à ceci :

```
$data = [
    'title' => 'My title',
    'body' => 'The text',
    'user_id' => 1,
    'user' => [
        'username' => 'mark'
    ],
    'comments' => [
        ['body' => 'First comment'],
        ['body' => 'Second comment'],
    ]
];
```

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 les données d'être validées, 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 prises en compte :

```
// Dans un controller

// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->get('Articles');

// Nouvelle entity avec des associations imbriquées
$entity = $articles->newEntity($this->request->getData(), [
    'associated' => [
        'Tags', 'Comments' => ['associated' => ['Users']]
    ]
]);
```

Ce qui est au-dessus indique que les "Tags", "Comments" et "Users" pour les Comments doivent être prises en compte. D'une autre façon, vous pouvez utiliser la notation par point pour être plus bref :

```
// Dans un controller

// Prior to 3.6 use TableRegistry::get('Articles')
```

(suite sur la page suivante)

(suite de la page précédente)

```
$articles = TableRegistry::getTableLocator()->get('Articles');

// Nouvelle entity avec des associations imbriquées en utilisant
// la notation par point
$entity = $articles->newEntity($this->request->getData(), [
    'associated' => ['Tags', 'Comments.Users']
]);
```

Vous pouvez aussi désactiver le marshalling d'associations imbriquées comme ceci :

```
$entity = $articles->newEntity($data, ['associated' => []]);
// ou...
$entity = $articles->patchEntity($entity, $data, ['associated' => []]);
```

Les données associées sont également validées par défaut à moins que le contraire ne lui soit spécifié. Vous pouvez également changer l'ensemble de validation utilisé par association :

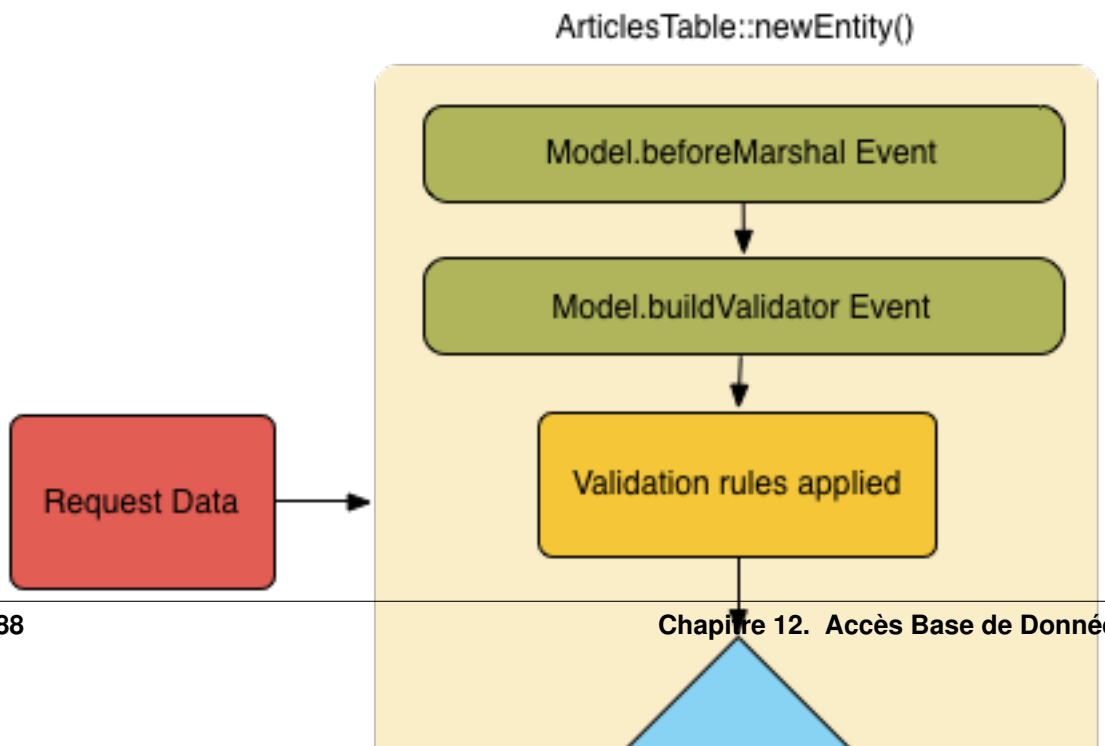
```
// Dans un controller

// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->get('Articles');

// Ne fait pas la 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* a plus d'informations sur la façon d'utiliser les différents validateurs pour des marshalling associés.

Le diagramme suivant donne un aperçu de ce qui se passe à l'intérieur de la méthode `newEntity()` ou `patchEntity()` :



Vous récupérerez toujours une entity en retour de `newEntity()`. Si la va-

lidation échoue, votre entité contiendra des erreurs et tous les champs invalides seront absents de l'entity créée.

Convertir des Données BelongsToMany

Si vous sauvegardez des associations belongsToMany, vous pouvez soit utiliser une liste de données d'entity ou une liste d'ids. Quand vous utilisez une liste de données d'entity, vos données requêtées devraient ressembler à ceci :

```
$data = [
    'title' => 'My title',
    'body' => 'The text',
    'user_id' => 1,
    'tags' => [
        ['name' => 'CakePHP'],
        ['name' => 'Internet'],
    ]
];
```

Le code ci-dessus créera 2 nouveaux tags. Si vous voulez créer un lien d'un article vers des tags existants, vous pouvez utiliser une lite des ids. Vos données de requête doivent ressembler à ceci :

```
$data = [
    'title' => 'My title',
    'body' => 'The text',
    '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' => 'My title',
    'body' => 'The text',
    'user_id' => 1,
    'tags' => [
        ['name' => 'A new tag'],
        ['name' => 'Another new tag'],
        ['id' => 5],
        ['id' => 21]
    ]
];
```

Quand les données ci-dessus seront converties en entities, il y aura 4 tags. Les deux premiers seront de nouveaux objets, et les deux seconds seront des références à des tags existants.

Quand les données de belongsToMany sont converties, 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 de belongsToMany pour utiliser uniquement la clé `_ids` et ignorer toutes les autres données.

Nouveau dans la version 3.1.0 : L'option `onlyIds` a été ajoutée dans 3.1.0

Convertir des Données HasMany

Si vous souhaitez mettre à jour les associations hasMany existantes et mettre à jour leurs propriétés, vous devriez d'abord vous assurer que votre entity est chargée avec l'association hasMany remplie. Vous pouvez ensuite utiliser les données de la requête 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 voulez lier des enregistrements existants à un nouveau parent, vous pouvez utiliser le format `_ids` :

```
$data = [
    'title' => 'My new article',
    'body' => 'The text',
    'user_id' => 1,
    'comments' => [
        '_ids' => [1, 2, 3, 4]
    ]
];
```

Quand les données de hasMany sont converties, 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.

Nouveau dans la version 3.1.0 : L'option `onlyIds` a été ajoutée dans 3.1.0

Convertir des Enregistrements Multiples

Lorsque vous créez des formulaires de création/mise à jour d'enregistrements multiples en une seule opération vous pouvez utiliser `newEntities()` :

```
// Dans un controller.

// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->get('Articles');
$entities = $articles->newEntities($this->request->getData());
```

Dans cette situation, les données de requête pour plusieurs articles doivent ressembler à ceci :

```
$data = [
    [
        'title' => 'First post',
        'published' => 1
    ],
    [
        'title' => 'Second post',
        'published' => 1
    ],
];
```

Une fois que vous avez converti les données requêtées dans des entities, vous pouvez leur faire un `save()` ou un `delete()` :

```
// Dans un controller.
foreach ($entities as $entity) {
    // Save entity
    $articles->save($entity);

    // Supprime l'entity
    $articles->delete($entity);
}
```

Ce qui est au-dessus va lancer une transaction séparée pour chaque entity sauvegardée. Si vous voulez traiter toutes les entities en transaction unique, vous pouvez utiliser `transactional()` :

```
// Dans un controller.
$articles->getConnection()->transactional(function () use ($articles, $entities) {
    foreach ($entities as $entity) {
        $articles->save($entity, ['atomic' => false]);
    }
});
```

Changer les Champs Accessibles

Il est également possible de permettre à `newEntity()` d'écrire dans des champs non accessibles. Par exemple, `id` est généralement absent de la propriété `_accessible`. Dans ce cas, vous pouvez utiliser l'option `accessibleFields`. Cela est particulièrement intéressant pour conserver les associations existantes entre certaines entities :

```
// Dans un controller.
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->get('Articles');
$entity = $articles->newEntity($this->request->getData(), [
    'associated' => [
        'Tags', 'Comments' => [
            'associated' => [
                'Users' => [
                    'accessibleFields' => ['id' => true]
                ]
            ]
        ]
    ]
]);
```

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 quelques unes ou toutes les données dans les entities résultantes, vérifiez deux fois que les colonnes que vous souhaitez définir sont listées dans la propriété `$_accessible` de votre entity.

Fusionner les Données Requêtées dans les Entities

Afin de mettre à jour les entities, vous pouvez choisir d'appliquer les données requêtées directement dans une entity existante. Ceci a l'avantage que seuls les champs qui changent réellement seront sauvegardés, au lieu d'envoyer tous les champs à la base de données, même ceux qui sont identiques. Vous pouvez fusionner un tableau de données brutes dans une entity existante en utilisant la méthode `patchEntity()` :

```
// Dans un controller.

// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::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 qu'elles soient copiées 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 montré ci-dessous :

```
// Dans un controller.

// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::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 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 requêtées 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 le champ de clé primaire dans la source entities avec les champs correspondants dans le tableau de données. Les associations vont construire de nouvelles entities si aucune entity précédente n'est trouvé pour la propriété cible.

Pa exemple, prenons les données requêtées comme ce qui suit :

```
$data = [
    'title' => 'My title',
    'user' => [
        'username' => 'mark'
    ]
];
```

Essayer de faire un patch d'une entity sans entity dans la propriété user va créer une nouvelle entity user :

```
// Dans un controller.
$entity = $articles->patchEntity(new Article, $data);
echo $entity->user->username; // Echoes 'mark'
```

La même chose peut être dite pour les associations hasMany et belongsToMany, mais avec une mise en garde importante.

Note : Pour les associations belongsToMany, vérifiez que les entities associées sont bien présentes dans la propriété `$_accessible`

Si Product belongsToMany Tag :

```
// Dans l'entity Product
protected $_accessible = [
    // .. autre propriété
   'tags' => true,
];
```

Note : Pour les associations hasMany et belongsToMany, s'il y avait des entities qui ne pouvaient pas correspondre avec leur clé primaire à aucun enregistrement dans le tableau de données, alors ces enregistrements seraient annulés de l'entity résultante.

Rappelez-vous que l'utilisation de `patchEntity()` ou de `patchEntities()` ne fait pas persister les données, il modifie juste (ou créé) les entities données. Afin de sauvegarder l'entity, vous devrez appeler la méthode `save()` de la table.

Par exemple, considérons le cas suivant :

```
$data = [
    'title' => 'My title',
    'body' => 'The text',
    'comments' => [
        ['body' => 'First comment', 'id' => 1],
        ['body' => 'Second comment', 'id' => 2],
    ]
];
$article = $articles->newEntity($data);
$articles->save($article);

$newData = [
    'comments' => [
        ['body' => 'Changed comment', 'id' => 1],
        ['body' => 'A new comment'],
    ]
];
$entity = $articles->newEntity($data);
$articles->save($entity);

$newData = [
    'comments' => [
        ['body' => 'Changed comment', 'id' => 1],
        ['body' => 'A new comment'],
    ]
];
$articles->patchEntity($entity, $newData);
$articles->save($entity);
```

A la fin, si l'entity est à nouveau convertie en tableau, vous obtiendrez le résultat suivant :

```
[
    'title' => 'My title',
    'body' => 'The text',
    'comments' => [
        ['body' => 'Changed comment', 'id' => 1],
        ['body' => 'A new comment'],
    ]
];
```

Comme vous l'avez vu, le commentaire avec l'id 2 n'est plus ici, puisqu'il ne correspondait à rien dans le tableau `$newData`. Ceci arrive car CakePHP reflète le nouvel état décrit dans les données requêtées.

Des avantages supplémentaires à cette approche sont qu'elle réduit le nombre d'opérations à exécuter quand on fait persister l'entity à nouveau.

Notez bien que ceci ne signifie pas que le commentaire avec l'id 2 a été supprimé de la base de données, si vous

souhaitez retirer les commentaires pour cet article qui ne sont pas présents dans l'entity, vous pouvez collecter les clés primaires et exécuter une suppression batch pour celles qui ne sont pas dans la liste :

```
// Dans un controller.

// Prior to 3.6 use TableRegistry::get('Comments')
$comments = TableRegistry::getTableLocator()->get('Comments');
$present = (new Collection($entity->comments))->extract('id')->filter()->toArray();
$comments->deleteAll([
    'article_id' => $article->id,
    'id NOT IN' => $present
]);
```

Comme vous pouvez le voir, ceci permet aussi de créer des solutions lorsqu'une association a besoin d'être implémentée comme un ensemble unique.

Vous pouvez aussi faire un patch de plusieurs entities en une fois. Les considérations faîtes pour les associations hasMany et belongsToMany s'appliquent pour le patch de plusieurs entities : Les correspondances sont faites avec la valeur du champ de la clé primaire et les correspondances manquantes dans le tableau original des entities seront retirées et non présentes dans les résultats :

```
// Dans un controller.

// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->get('Articles');
$list = $articles->find('popular')->toArray();
$patched = $articles->patchEntities($list, $this->request->getData());
foreach ($patched as $entity) {
    $articles->save($entity);
}
```

De la même façon que pour l'utilisation de `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 Requêtées Avant de Construire les Entities

Si vous devez modifier les données requêtées avant qu'elles ne soient converties en entities, vous pouvez utiliser l'event `Model.beforeMarshal`. Cet event vous laisse manipuler les données requêtées juste avant que les entities ne soient créées :

```
// Mettez des use en haut de votre fichier.
use Cake\Event\Event;
use ArrayObject;

// Dans une classe table ou behavior
public function beforeMarshal(Event $event, ArrayObject $data, ArrayObject $options)
{
    if (isset($data['username'])) {
        $data['username'] = mb_strtolower($data['username']);
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }
}
```

Le paramètre `$data` est une instance `ArrayObject`, donc vous n'avez pas à la retourner pour changer les données 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 mises 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, telle que la whitelist des champs. La validation est lancée juste après que cet événement soit terminé. Un exemple commun de modification des données avant qu'elles soient validées est la suppression des espaces superflus d'un champ avant l'enregistrement :

```
// Mettez des use en haut de votre fichier.
use Cake\Event\Event;
use ArrayObject;

// Dans une table ou un behavior
public function beforeMarshal(Event $event, ArrayObject $data, ArrayObject $options)
{
    foreach ($data as $key => $value) {
        if (is_string($value)) {
            $data[$key] = trim($value);
        }
    }
}
```

A cause de la manière dont le process de marshalling fonctionne, 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.

Valider les Données Avant de Construire les Entities

Le chapitre *Valider des Données* recèle plus d'information sur l'utilisation des fonctionnalités de validation de CakePHP pour garantir que vos données restent correctes et cohérentes.

Eviter les Attaques d'Assignement en Masse de Propriétés

Lors de la création ou la fusion des entities à partir de données requêtées, vous devez faire attention à ce que vous autorisez à changer ou à ajouter dans les entities à vos utilisateurs. Par exemple, en envoyant un tableau dans la requête contenant `user_id`, un pirate pourrait changer le propriétaire d'un article, ce qui entraînerait des effets indésirables :

```
// Contient ['user_id' => 100, 'title' => 'Hacked!'];
$data = $this->request->data;
$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 `fieldList` lors de la création ou la fusion de données dans une entity :

```
// Contient ['user_id' => 100, 'title' => 'Hacked!'];
$data = $this->request->data;

// Permet seulement de changer le title
$entity = $this->patchEntity($entity, $data, [
    'fieldList' => ['title']
]);
$this->save($entity);
```

Vous pouvez aussi contrôler les propriétés qui peuvent être assignées pour les associations :

```
// Permet seulement le changement de title et de tags
// et le nom du tag est la seule colonne qui peut être définie
$entity = $this->patchEntity($entity, $data, [
    'fieldList' => ['title', 'tags'],
    'associated' => ['Tags' => ['fieldList' => ['name']]]
]);
$this->save($entity);
```

Utiliser cette fonctionnalité est pratique quand vous avez différentes fonctions auxquelles vos utilisateurs peuvent accéder et que vous voulez laisser vos utilisateurs modifier différentes données basées sur leurs privilèges.

L'option `fieldList` est aussi acceptée par les méthodes `newEntity()`, `newEntities()` et `patchEntities()`.

Sauvegarder les Entities

`Cake\ORM\Table::save`(*Entity $entity*, *array $options = []*)

Quand vous sauvegardez les données requêtées dans votre base de données, vous devez d'abord hydrater une nouvelle entity en utilisant `newEntity()` pour passer dans `save()`. Pare exemple :

```
// Dans un controller

// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->get('Articles');
$article = $articles->newEntity($this->request->getData());
if ($articles->save($article)) {
    // ...
}
```

L'ORM utilise la méthode `isNew()` sur une entity pour déterminer si oui ou non une insertion ou une mise à jour doit être faite. Si la méthode `isNew()` retourne `true` et que l'entity a une valeur de clé primaire, une requête "exists" sera faîte. La requête "exists" peut être supprimée en passant `'checkExisting' => false` à l'argument `$options` :

```
$articles->save($article, ['checkExisting' => false]);
```

Une fois que vous avez chargé quelques entities, vous voudrez probablement les modifier et les mettre à jour dans votre base de données. C'est un exercice simple dans CakePHP :

```
// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->get('Articles');
$article = $articles->find('all')->where(['id' => 2])->first();

$article->title = 'My new title';
$articles->save($article);
```

Lors de la sauvegarde, CakePHP va *appliquer vos règles de validation*, et entourer l'opération de sauvegarde dans une transaction de base de données. Cela va aussi seulement mettre à jour les propriétés qui ont changé. Le `save()` ci-dessus va générer le code SQL suivant :

```
UPDATE articles SET title = 'My new title' WHERE id = 2;
```

Si vous avez une nouvelle entity, le code SQL suivant serait généré :

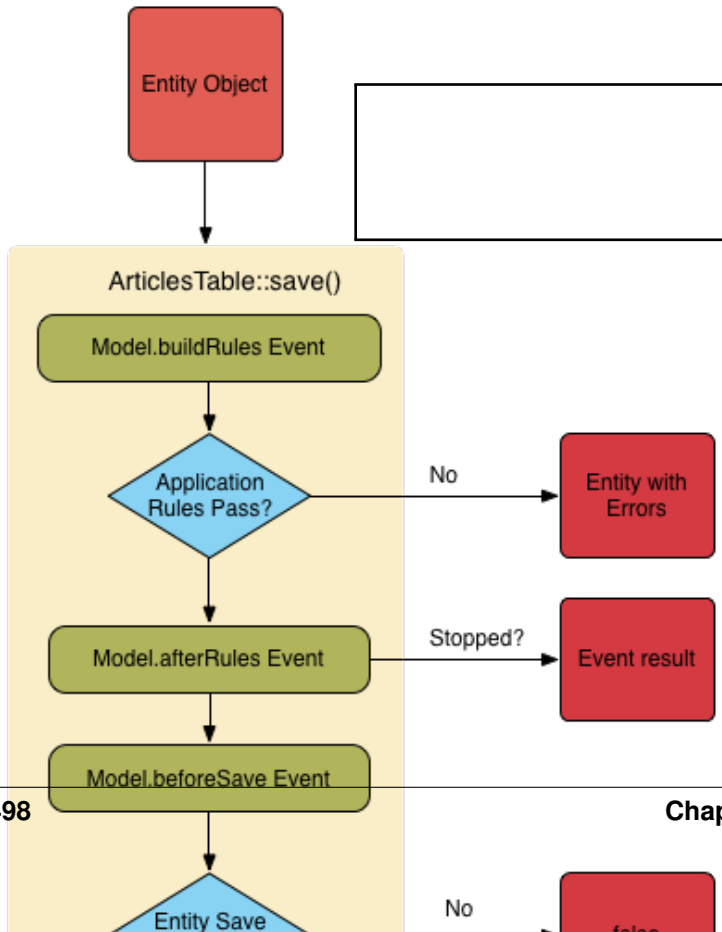
```
INSERT INTO articles (title) VALUES ('My new title');
```

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 va connaitre un échec et retourner `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 Enfant 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 le procédé ci-dessus :

Consultez la section *Appliquer des Règles pour l'Application* pour plus d'informations sur la création et l'utilisation des règles.



Avertissement : Si aucun
fait à l'entity quand elle es
callbacks ne vont pas être
cune sauvegarde n'est faîte

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` pendant la sauvegarde :

Sauvegarder les Associations

Quand vous sauvegardez une entity, vous pouvez aussi choisir d'avoir quelques unes ou toutes les entities associées. Par défaut, toutes les entities de premier niveau seront sauvegardées. Par exemple sauvegarder un Article, va aussi automatiquement mettre à jour tout entity modifiée qui n'est pas directement liée à la table articles.

Vous pouvez régler finement les associations qui sont sauvegardées en utilisant l'option `associated` :

Vous pouvez définir une sauvegarde distante ou des associations imbriquées profondément en utilisant la notation par point :

Si vous avez besoin de lancer un ensemble de règle de validation différente pour une association, vous pouvez le spécifier dans un tableau d'options pour l'association :

En plus, vous pouvez combiner la notation par point pour les associations avec le tableau d'options :

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 une donnée d'association après avoir construit vos entities, vous devrez marquer la propriété d'association comme étant modifiée avec `dirty()` :

Sauvegarder les Associations BelongsTo

Lors de la sauvegarde des associations belongsTo, l'ORM s'attend à une entity imbriquée unique avec le nom de l'association au singulier et *en underscore*. Par exemple :

Sauvegarder les Associations HasOne

Lors de la sauvegarde d'associations hasOne, l'ORM s'attend à une entity imbriquée unique avec le nom de l'association au singulier et *en underscore*. Par exemple :

```
↪TableRegistry::getTableLocator()-
↪>get('Users');
```

Sauvegarder les Associations HasMany

Lors de la sauvegarde d'associations hasMany, l'ORM s'attend à une entity imbriquée unique avec le nom de l'association au pluriel et *en underscore*. Par exemple :

```
// Dans un controller.
$data = [
    'title' => 'First Post',
    'comments' => [
        ['body' => 'Best post ever'],
        ['body' => 'I really like this.']
    ]
];

// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::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, si 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, la stratégie de sauvegarde `append` est utilisée. Consultez *Associations HasMany* pour plus de détails sur la définition de `saveStrategy`.

Peu importe le moment où vous ajoutez de nouveaux enregistrements dans une association existante, vous devez toujours marquer la propriété de l'association comme "dirty". Ceci dit à l'ORM que la propriété de l'association doit persister :

```
$article->comments[] = $comment;
$article->dirty('comments', true);
```

Sans l'appel à `dirty()`, les commentaires mis à jour ne seront pas sauvegardés.

Sauvegarder les Associations BelongsToMany

Lors de la sauvegarde d'associations hasMany, l'ORM s'attend à une entity imbriquée unique avec le nom de l'association au pluriel et *en underscore*. Par exemple :

```
// Dans un controller.
$data = [
    'title' => 'First Post',
    'tags' => [
        ['tag' => 'CakePHP'],
        ['tag' => 'Framework']
    ]
];

// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->get('Articles');
$article = $articles->newEntity($data, [
    'associated' => ['Tags']
]);
$articles->save($article);
```

Quand vous convertissez les données requêtées en entities, les méthodes `newEntity()` et `newEntities()` vont gérer les deux tableaux de propriétés, ainsi qu'une liste d'ids avec la clé `_ids`. Utiliser la clé `_ids` facilite la construction d'un box select ou d'un checkbox basé sur les contrôles pour les associations belongs to many. 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 ne va pas détruire les liens existants même s'ils ne sont pas présents dans le tableau d'entities à sauvegarder.

replace Lors de la sauvegarde, les liens existants seront retirés 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 entities que l'on souhaite sauvegarder, ces liens seront mis à jour, non supprimés et re-sauvegardés.

Consultez *Associations BelongsToMany* pour plus de détails sur la définition de `saveStrategy`.

Par défaut la stratégie `replace` est utilisée. Quand vous avez de nouveaux enregistrements dans une association existante, vous devez toujours marquer la propriété de l'association en "dirty". Ceci dit à l'ORM que la propriété de l'association doit persister :

```
$article->tags[] = $tag;
$article->dirty('tags', true);
```

Sans appel à `dirty()`, les tags mis à jour ne seront pas sauvegardés.

Vous vous voudrez probablement souvent créer une association entre deux entities existantes, par exemple un utilisateur co-auteur d'un article. Cela est possible en utilisant la méthode `link()` comme ceci :

```
$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 additionnelles dans la table de jointure. Dans l'exemple précédent des tags, ça pourrait être le type de vote `vote_type` de la personne qui a voté sur cet article. Le `vote_type` peut être `upvote` ou `downvote` et 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 ensuite en sauvegardant 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 à la Table de Jointure

Dans certaines situations, la table de jointure de l'association BelongsToMany, aura des colonnes supplémentaires. CakePHP facilite la sauvegarde des propriétés dans ces colonnes. Chaque entity dans une association belongsToMany a une propriété `_joinData` qui contient les colonnes supplémentaires sur 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 sur 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 les entities construites à partir de données requêtées. Lorsque vous sauvegardez des données de tables jointes depuis les données requêtées, vos données POST doivent ressembler à ceci :

```
$data = [
    'first_name' => 'Sally',
    'last_name' => 'Parker',
    'courses' => [
        [
            'id' => 10,
            '_joinData' => [
                'grade' => 80.12,
                'days_attended' => 30
            ]
```

(suite sur la page suivante)

(suite de la page précédente)

```
        ],
        // d'autres cours (courses).
    ]
];
$student = $this->Students->newEntity($data, [
    'associated' => ['Courses._joinData']
]);
```

Regardez le chapitre sur les *inputs pour les données associées* pour savoir comment construire des inputs avec le `FormHelper` correctement.

Sauvegarder les Types Complexes

Les tables peuvent stocker des données représentées dans des types basiques, comme les chaînes, les integers, floats, booleans, etc… Mais elles peuvent aussi être étendues pour accepter plus de types complexes comme les tableaux ou les objets et sérialiser ces données en types plus simples qui peuvent être sauvegardés dans la base de données.

Cette fonctionnalité se fait en utilisant le système personnalisé de types. Consulter la section *Ajouter des Types Personnalisés* pour trouver comment construire les Types de colonne personnalisés :

```
// Dans config/bootstrap.php

use Cake\Database\Type;

Type::map('json', 'Cake\Database\Type\JsonType');

// Dans src/Model/Table/UsersTable.php
use Cake\Database\Schema\TableSchema;

class UsersTable extends Table
{

    protected function _initializeSchema(TableSchema $schema)
    {
        $schema->columnType('preferences', 'json');
        return $schema;
    }

}
```

Le code ci-dessus correspond à la colonne `preferences` pour le type personnalisé `json`. Cela signifie que quand on récupère des données pour cette colonne, elles seront désérialisées à partir d'une chaîne JSON dans la base de données et mises dans une entity en tant que tableau.

Comme ceci, lors de la sauvegarde, le tableau sera transformé à nouveau en sa représentation JSON :

```
$user = new User([
    'preferences' => [
        'sports' => ['football', 'baseball'],
        'books' => ['Mastering PHP', 'Hamlet']
    ]
]);
$usersTable->save($user);
```

Lors de l'utilisation de types complexes, il est important de vérifier que les données que vous recevez de l'utilisateur final sont valides. Ne pas gérer correctement les données complexes va permettre à des utilisateurs mal intentionnés

d'être capable de stocker des données qu'ils ne pourraient pas stocker normalement.

Strict Saving

`Cake\ORM\Table::saveOrFail`(*$entity*, *$options = []*)

Utiliser 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_.

Utiliser cette méthode peut être utile pour effectuer des opérations complexes en 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();
}
```

Puisque cette méthode utilise la méthode `Cake\ORM\Table::save()`, tous les événements de `save` seront déclenchés.

Nouveau dans la version 3.4.1.

Sauvegarder Plusieurs Entities

`Cake\ORM\Table::saveMany`(*$entities*, *$options = []*)

En utilisant cette méthode, vous pouvez sauvegarder plusieurs entities de façon atomique. `$entities` peuvent être un tableau d'entities créé avec `newEntities()` / `patchEntities()`. `$options` peut avoir les mêmes options que celles acceptées par `save()` :

```
$data = [
    [
        'title' => 'First post',
        'published' => 1
    ],
    [
        'title' => 'Second post',
        'published' => 1
    ],
];

// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->get('Articles');
$entities = $articles->newEntities($data);
$result = $articles->saveMany($entities);
```

Le résultat sera la mise à jour des entities en cas de succès ou `false` en cas d'échec.

Nouveau dans la version 3.2.8.

Mises à Jour en Masse

`Cake\ORM\Table::updateAll`(*$fields*, *$conditions*)

Il peut arriver que la mise à jour de lignes individuellement n'est pas efficace ou pas nécessaire. Dans ces cas, il est plus efficace d'utiliser une mise à jour en masse pour modifier plusieurs lignes en une fois :

```
// Publie tous les articles non publiés.
function publishAllUnpublished()
{
    $this->updateAll(
        ['published' => true], // champs
        ['published' => false]); // conditions
}
```

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()
{
    $expression = new QueryExpression('view_count = view_count + 1');
    $this->updateAll([$expression], ['published' => true]);
}
```

Une mise à jour en masse sera considérée comme un succès 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 uniquement 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**(*Entity $entity*, *$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
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($entity, $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.

Nouveau dans la version 3.4.1.

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. CakePHP facilite la gestion de ces associations. 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)
    {
        $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)
    {
        $this->belongsTo('Authors', [
                'className' => 'Publishing.Authors'
            ])
            ->setForeignKey('authorid')
            ->setProperty('person');
    }
}
```

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)
    {
        $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)
    {
        $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)
    {
       $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)
    {
        $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)
    {
        $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** : Définit la stratégie de requête à utiliser. Par défaut à "join". L'autre valeur valide est "select", qui utilise une requête distincte à la place.
- **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)
    {
        $this->belongsTo('Users');
    }
}
```

Nous pouvons aussi définir une relation plus spécifique en utilisant les setters :

```
class AddressesTable extends Table
{

    public function initialize(array $config)
    {
        $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** : Définit la stratégie de requête à utiliser. Par défaut à "join". L'autre valeur valide est "select", qui utilise une requête distincte à la place.
— **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)
    {
        $this->hasMany('Comments');
    }
}
```

Nous pouvons également définir une relation plus spécifique en utilisant les setters :

```
class ArticlesTable extends Table
{

    public function initialize(array $config)
    {
        $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

Note : A partir de la version 3.0, `hasAndBelongsToMany` / `HABTM` a été renommé en `belongsToMany` / `BTM`.

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 models dans une association BelongsToMany n'est pas exclusif. par exemple nous joignons notre table Articles avec la table Tags. En utilisant "funny" comme un Tag pour mon Article, n"« utilise » pas le tag. Je peux aussi 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
{

    public function initialize(array $config)
    {
        $this->belongsToMany('Tags');
    }
}

// Dans src/Model/Table/TagsTable.php
class TagsTable extends Table
{

    public function initialize(array $config)
    {
        $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)
    {
        $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)
    {
        $this->belongsToMany('Courses', [
            'through' => 'CoursesMemberships',
        ]);
    }
}

class CoursesTable extends Table
{
    public function initialize(array $config)
    {
        $this->belongsToMany('Students', [
            'through' => 'CoursesMemberships',
        ]);
    }
}

class CoursesMembershipsTable extends Table
{
    public function initialize(array $config)
    {
        $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 une façon facile 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 première raison est les 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)
    {
        $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)
    {
        $this->addBehavior('Timestamp', [
            '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)
    {
        $this->addBehavior('CounterCache', [
            'Articles' => ['comment_count']
        ]);
    }
}
```

La configuration de CounterCache doit être composée de noms de relations et de la configuration spécifique pour cette relation.

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 utilisez `updateAll()` ou `deleteAll()`, ou que vous executez du SQL que vous

avez écrit.

Usage Avancée

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' => [
```

(suite sur la page suivante)

(suite de la page précédente)

```
        '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.

Il est cependant possible de le faire fonctionner pour les associations `belongsToMany`. Vous devez activer le comportement CounterCache dans une table `through` personnalisée configurée en tant qu'option d'association. Référez-vous à la configuration des tables de jointure en *utilisant l'option "through"*.

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)
    {
        $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)
    {
        $this->addBehavior('Timestamp', [
```

(suite sur la page suivante)

(suite de la page précédente)

```
            '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);
```

Oui, aussi facilement. 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 [129]).

Astuce : 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.

129. https://en.wikipedia.org/wiki/UN_M.49

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)
    {
        $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)
    {
        $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');
$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)
    {
        $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)
    {
        $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);
```

Depuis la version 3.3.0, le travail avec plusieurs traductions a été amélioré. Vous pouvez 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)
    {
        $this->addBehavior('Translate', [
            'fields' => ['title'],
            'validator' => 'translated'
        ]);
    }
}
```

Ce qui est au-dessus va utiliser le validateur créé par les entities traduites validées `validationTranslated`.

Nouveau dans la version 3.3.0 : La validation des entities traduites et l'amélioration de la sauvegarde des traductions ont été ajoutées dans la version 3.3.0

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 [130]

> **Avertissement :** TreeBehavior ne supporte pas les clés primaires composites pour le moment.

130. http://www.sitepoint.com/hierarchical-data-database-2/

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
{
    public function initialize(array $config)
    {
        $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);
```

Obtenir une liste aplatie des descendants pour un nœud est également facile :

```
$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])
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ->find('threaded')
    ->toArray();

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)
{
    $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)
    {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        $this->addBehavior('Tree', [
            '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

Nouveau dans la version 3.0.14.

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)
{
    // 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)
    {
        $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', [
    'implementedFinders' => [
        'slugged' => 'findSlug',
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ]
]);
```

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.

Nouveau dans la version 3.3.0 : La possibilité pour les behaviors de participer au marshalling a été ajoutée dans la version 3.3.0

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 $options)
    {
        parent::initialize($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', [
  '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',
  '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 nécessaires. 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 [131]. 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 [132]. Wincache offre des fonctionnalités et des performances semblables à APC, mais optimisées pour Windows et IIS.
— `XcacheEngine` Xcache [133]. est une extension PHP qui fournit des fonctionnalités similaires à APC.
— `MemcachedEngine` Utilise l'extension Memcached [134].
— `RedisEngine` Utilise l'extension phpredis [135]. Redis fournit un système de cache cohérent et rapide similaire à Memcached et il permet aussi les opérations atomiques.

131. http://php.net/apcu
132. http://php.net/wincache
133. http://xcache.lighttpd.net/
134. http://php.net/memcached
135. https://github.com/nicolasff/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',
    'path' => CACHE,
    'prefix' => 'cake_short_'
```

(suite sur la page suivante)

(suite de la page précédente)

```
]);

// 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',
    '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.

Nouveau dans la version 3.5.0 : Les fallbacks pour moteur de cache ont été ajoutés.

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
]);

// $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 facilite 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);

// 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 de manière simple. 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 [136].

136. https://book.cakephp.org/bake/1.x/fr/

CHAPITRE 15

Outils de Console, Shells, & Tasks

CakePHP n'est pas seulement un framework web, c'est aussi une console de framework pour la création d'applications. Les applications par la console sont idéales pour la gestion des tâches d'arrière-plan comme la maintenance et l'achèvement du travail en-dehors du cycle de requête-réponse. Les applications par la console CakePHP vous permettent de réutiliser les classes de votre application à partir de lignes de commande.

CakePHP dispose d'un certain nombre d'applications fournies pour la console. Certaines de ces applications sont utilisées de concert avec les fonctionnalités de CakePHP (comme i18n), et d'autres sont pour une utilisation générale pour que votre travail se fasse plus vite.

La console de CakePHP

Cette section est une introduction sur la ligne de commande dans CakePHP. Les outils de la Console sont idéals pour l'utilisation de tâches cron, ou pour les utilitaires basés sur les lignes de commandes qui n'ont pas besoin d'être accessibles par un navigateur.

PHP fournit un puissant client CLI qui rend l'interfaçage avec votre système de fichier et vos applications plus facile. La Console CakePHP fournit un framework de création de scripts shell. La console utilise un ensemble de répartiteur de types pour charger un shell ou une tâche, et lui passer des paramètres.

Note : Une installation de PHP construite avec la ligne de commande (CLI) doit être disponible sur le système si vous prévoyez d'utiliser la Console.

Avant d'entrer dans les spécificités, assurons-nous que vous pouvez exécuter la console CakePHP. Tout d'abord, vous devrez ouvrir un shell système. Les exemples présentés dans cette section seront en bash, mais la console CakePHP est également compatible avec Windows. Exécutons le programme Console depuis le bash. Cet exemple suppose que l'utilisateur est actuellement connecté dans l'invité bash et qu'il est en root sur une installation CakePHP.

Une application CakePHP contient les répertoires **src/Shell** et **src/Shell/Task** qui contiennent tous ses shells et tasks. Il est aussi livré avec un exécutable dans le répertoire **bin** :

```
$ cd /path/to/app
$ bin/cake
```

Note : Sur Windows, cette commande doit être `bin\cake` (notez l'antislash).

Quand vous lancez la Console sans argument, cela affiche ce message d'aide :

```
Welcome to CakePHP v3.6.0 Console
---------------------------------------------------------------
App : App
Path: /Users/markstory/Sites/cakephp-app/src/
---------------------------------------------------------------
Current Paths:

 -app: src
 -root: /Users/markstory/Sites/cakephp-app
 -core: /Users/markstory/Sites/cakephp-app/vendor/cakephp/cakephp

Changing Paths:

Your working path should be the same as your application path. To change your path␣
↪use the '-app' param.
Example: -app relative/path/to/myapp or -app /absolute/path/to/myapp

Available Shells:

- version
- help
- cache
- completion
- i18n
- schema_cache
- plugin
- routes
- server
- bug
- console
- event
- orm
- bake
- bake.bake
- migrations
- migrations.migrations

To run a command, type `cake shell_name [args|options]`
To get help on a specific command, type `cake shell_name --help`
```

La première information affichée est en rapport avec les chemins. Ceci est particulièrement pratique si vous exécutez la Console depuis différents endroits de votre système de fichier.

Vous pouvez lancer n'importe quel shell listé en utilisant son nom :

```
# lance le shell server
bin/cake server

# lance le shell migrations
```

(suite sur la page suivante)

(suite de la page précédente)

```
bin/cake migrations -h

# lance le shell bake (avec le préfixe plugin)
bin/cake bake.bake -h
```

Les shells de plugins peuvent être invoqués sans le préfixe du plugin si le nom du shell n'entre pas en collision avec un shell de l'application ou du framework. Dans le cas où deux plugins fournissent un shell du même nom, c'est le premier chargé qui récupérera l'alias court. Vous pouvez toujours utiliser le format `plugin.shell` pour référencer un shell sans ambiguïté.

class Cake\Console**Shell**

Créer un Shell

Créons un shell pour l'utilisation dans la Console. Pour cet exemple, nous créerons un simple shell Hello world. Dans le répertoire **src/Shell** de votre application, créez **HelloShell.php**. Mettez le code suivant dedans :

```
namespace App\Shell;

use Cake\Console\Shell;

class HelloShell extends Shell
{
    public function main()
    {
        $this->out('Hello world.');
    }
}
```

Les conventions pour les classes de shell sont que les noms de classe doivent correspondre au nom du fichier, avec Shell en suffixe. Dans notre shell, nous avons créé une méthode `main()`. Cette méthode est appelée quand un shell est appelé avec aucune commande supplémentaire. Nous allons ajouter quelques commandes en plus dans un moment, mais pour l'instant lançons juste notre shell. Depuis le répertoire de votre application, lancez :

```
bin/cake hello
```

Vous devriez voir la sortie suivante :

```
Hello world.
```

Comme mentionné précédemment, la méthode `main()` dans les shells est une méthode spéciale appelée tant qu'il n'y a pas d'autres commandes ou arguments donnés au shell. Comme notre méthode principale n'était pas très intéressante, ajoutons une autre commande qui fait quelque chose :

```
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);
    }
}
```

Après avoir enregistré ce fichier, vous devriez pouvoir lancer la commande suivante et voir votre nom affiché :

```
bin/cake hello hey_there your-name
```

Toute méthode publique non préfixée par un _ peut être appelée d'une ligne de commande. Comme vous pouvez le voir, les méthodes appelées avec la ligne de commande sont transformées d'un argument en underscore du shell en un bon nom de méthode en camel-case de la classe.

Dans notre méthode `heyThere()`, nous pouvons voir que les arguments de position sont envoyés à notre fonction `heyThere()`. Les arguments de position sont aussi disponible dans la propriété `args`. Vous pouvez accéder switches ou aux options des applications du shell, qui sont disponibles dans `$this->params`, mais nous étudierons ce point plus tard.

Lorsque vous utilisez la méthode `main()`, vous ne pouvez pas utiliser les arguments de position ou les paramètres. Cela parce que le premier argument de position ou l'option est interprété en tant que nom de commande. Si vous voulez utiliser des arguments et des options, vous devriez utiliser un autre nom de méthode que `main`.

Les Tâches Shell

Il y aura des fois où quand vous construirez des applications plus poussées via la console, vous voudrez composer des fonctionnalités dans des classes réutilisables qui peuvent être partagées à travers plusieurs shells. Les tâches vous permettent d'extraire des commandes dans des classes. Par exemple, la commande `bake` est fait entièrement de tâches. Vous définissez les tâches d'un shell en utilisant la propriété `$tasks` :

```
class UserShell extends Shell
{
    public $tasks = ['Template'];
}
```

Vous pouvez utiliser les tâches à partir de plugins en utilisant la *syntaxe de plugin* standard. Les tâches sont stockées dans **src/Shell/Task/** dans les fichiers nommées d'après leur classe. Ainsi si vous étiez sur le point de créer une nouvelle tâche "FileGenerator", vous pourriez créer **src/Shell/Task/FileGeneratorTask.php**.

Chaque tâche doit au moins intégrer une méthode `main()`. Le ShellDispatcher appellera cette méthode quand la tâche est invoquée. Une classe de tâche ressemble à cela :

```
namespace App\Shell\Task;

use Cake\Console\Shell;

class FileGeneratorTask extends Shell
{
    public function main()
    {

    }
}
```

Un shell peut aussi accéder à ses tâches en tant que propriétés, ce qui rend les tâches meilleures pour la réutilisation de fonctions identiques à *Components (Composants)* :

```
// Dans src/Console/Command/SeaShell.php
class SeaShell extends Shell
{
    // Se trouve dans src/Shell/Task/SoundTask.php
    public $tasks = ['Sound'];

    public function main()
    {
        $this->Sound->main();
    }
}
```

Vous pouvez aussi accéder aux tâches directement à partir de la ligne de commande :

```
$ cake sea sound
```

Note : Afin d'accéder aux tâches directement à partir de ligne de commande, la tâche **doit** être inclue dans la propriété $tasks de la classe shell.

De plus, le nom de la task doit être ajouté en tout que sous commande dans l'OptionParser du Shell :

```
public function getOptionParser()
{
    $parser = parent::getOptionParser();
    $parser->addSubcommand('sound', [
        // Fournit un texte d'aide pour la liste des commandes
        'help' => 'Execute The Sound Task.',
        // Lie les parsers d'option.
        'parser' => $this->Sound->getOptionParser(),
    ]);
    return $parser;
}
```

Chargement à la volée des tâches avec TaskCollection

Vous pouvez charger les tâches à la volée en utilisant l'objet Task Collection. Vous pouvez charger les tâches qui ne sont pas déclarées dans $tasks de cette façon :

```
$project = $this->Tasks->load('Project');
```

Chargera et retournera une instance ProjectTask. Vous pouvez charger les tâches à partir des plugins en utilisant :

```
$progressBar = $this->Tasks->load('ProgressBar.ProgressBar');
```

Utiliser les Models dans vos shells

Vous avez souvent besoin d'accéder à la logique métier de votre application dans les utilitaires de shell. CakePHP rend cela super facile. Vous pouvez charger les models dans les shells, juste comme vous le feriez dans un controller en utilisant `loadModel()`. Les models définis sont chargés en propriétés attachées à votre shell :

```
namespace App\Shell;

use Cake\Console\Shell;

class UserShell extends Shell
{

    public function initialize()
    {
        parent::initialize();
        $this->loadModel('Users');
    }

    public function show()
    {
        if (empty($this->args[0])) {
            // Utilisez error() avant CakePHP 3.2
            return $this->abort("Merci de rentrer un nom d'utilisateur.");
        }
        $user = $this->Users->findByUsername($this->args[0])->first();
        $this->out(print_r($user, true));
    }
}
```

Le shell ci-dessus récupérera un utilisateur par son username et affichera l'information stockée dans la base de données.

Shell Helpers

Si vous avez une logique complexe de génération de sortie, vous pouvez utiliser les *Shell Helpers* pour encapsuler cette logique d'une manière réutilisable.

Appeler d'autres Shells à partir de votre Shell

Il y a effectivement beaucoup de cas où vous voulez appeler un shell à partir d'un autre. `Shell::dispatchShell()` vous donne la possibilité d'appeler d'autres shells en fournissant le `argv` pour le shell sub. Vous pouvez fournir des arguments et des options soit en variables args ou en chaînes de caractères :

```
// En chaînes de caractère
$this->dispatchShell('schema create Blog --plugin Blog');

// En tableau
$this->dispatchShell('schema', 'create',  'Blog', '--plugin',  'Blog');
```

Ce qui est au-dessus montre comment vous pouvez appeler le shell schema pour un plugin à partir du shell de votre plugin.

Passer des paramètres supplémentaires au Shell appelé

Nouveau dans la version 3.1.

Il peut parfois être utile de passer des paramètres supplémentaires (qui ne seraient pas des arguments du Shell) aux Shells appelés. Pour ce faire, vous pouvez maintenant passer un tableau à `dispatchShell()`. Le tableau devra avoir une clé `command` ainsi qu'une clé `extra` :

```
// En passant la commande via une chaîne
$this->dispatchShell([
   'command' => 'schema create Blog --plugin Blog',
   'extra' => [
        'foo' => 'bar'
    ]
]);

// En passant la commande via un tableau
$this->dispatchShell([
   'command' => ['schema', 'create', 'Blog', '--plugin', 'Blog'],
   'extra' => [
        'foo' => 'bar'
    ]
]);
```

Les paramètres ajoutés via `extra` seront fusionnés dans la propriété `Shell::$params` et accessibles via la méthode `Shell::param()`. Par défaut, un paramètre supplémentaire `requested` est automatiquement ajouté quand un Shell est appelé via `dispatchShell()`. Ce paramètre empêche la console de CakePHP d'afficher le message de bienvenue à chaque Shell appelé via `dispatchShell()`.

Récupérer les Entrées de l'Utilisateur

`Cake\Console\Shell::in`(*$question*, *$choices = null*, *$default = null*)

Quand vous construisez des applications pour console, vous avez besoin des entrées de l'utilisateur. CakePHP a un moyen facile de le faire :

```
// Récupère un texte arbitraire de l'utilisateur.
$color = $this->in('What color do you like?');

// Récupère un choix de l'utilisateur.
$selection = $this->in('Red or Green?', ['R', 'G'], 'R');
```

La validation de la sélection est sensible à la casse.

Créer des Fichiers

`Cake\Console\Shell::createFile`(*$path*, *$contents*)

Beaucoup d'applications Shell aident à automatiser les tâches de développement et de déploiement. Créer des fichiers est souvent important dans ces cas d'utilisation. CakePHP fournit un moyen facile pour créer un fichier pour un chemin donné :

```
$this->createFile('bower.json', $stuff);
```

Si le Shell est interactif, un avertissement sera généré, et il sera demandé à l'utilisateur s'il veut écraser le fichier s'il existe déjà. Si la propriété interactive du shell est à `false`, aucune question ne sera posée et le fichier sera simplement écrasé.

Sortie de la Console

La classe `Shell` fournit quelques méthodes pour afficher le contenu :

```
// Ecrire avec stdout
$this->out('Message Normal');

// Ecrire avec stderr
$this->err('Message d\'erreur');

// Ecrire avec stderr et lancer une exception
$this->abort('Fatal error');

// Avant CakePHP 3.2. Ecrire avec stderr et exit()
$this->error('Erreur Fatale');
```

Elle fournit aussi deux méthodes pratiques en ce qui concerne le niveau de sortie :

```
// Apparaîtrait seulement quand la sortie verbeuse est activée (-v)
$this->verbose('Message verbeux');

// Apparaîtrait à tous les niveaux.
$this->quiet('Message silencieux');
```

Le Shell a aussi quelques méthodes pour nettoyer la sortie, créer des lignes blanches, ou dessiner une ligne de tirets :

```
// Affiche 2 newlines
$this->out($this->nl(2));

// Nettoie l'écran de l'utilisateur
$this->clear();

// Dessine une ligne horizontale
$this->hr();
```

Enfin, vous pouvez mettre à jour la ligne courante du texte sur l'écran en utilisant `_io->overwrite()` :

```
$this->out('Counting down');
$this->out('10', 0);
for ($i = 9; $i > 0; $i--) {
    sleep(1);
    $this->_io->overwrite($i, 0, 2);
}
```

Il est important de se rappeler que vous ne pouvez pas écraser le texte une fois qu'une nouvelle ligne a été affichée.

Niveaux de sortie de la Console

Les Shells ont souvent besoin de différents niveaux de verbosité. Quand vous lancez une tâche cron, la plupart des sorties ne sont pas nécessaires. Et il y a des fois où vous n'êtes pas intéressés dans tout ce qu'un shell a à dire. Vous pouvez utiliser des niveaux de sortie pour signaler la sortie de façon appropriée. L'utilisateur du shell peut ensuite décider pour quel niveau de détail ils sont intéressés en configurant le bon flag quand on appelle le shell. `Cake\Console\Shell::out()` supporte 3 types de sortie par défaut.

- `QUIET` - Seulement des informations importantes doivent être marquées pour une paisible.
- `NORMAL` - Le niveau par défaut, et un usage normal.
- `VERBOSE` - Les messages marqués qui peuvent être trop ennuyeux pour une utilisation quotidienne, mais aide au debugging en `VERBOSE`.

Vous pouvez marquer la sortie comme suit :

```
// apparaitra à tous les niveaux.
$this->out('Quiet message', 1, Shell::QUIET);
$this->quiet('Quiet message');

// n'apparaitra pas quand une sortie quiet est togglé
$this->out('message normal', 1, Shell::NORMAL);
$this->out('message loud', 1, Shell::VERBOSE);
$this->verbose('Verbose output');

// would only appear when verbose output is enabled.
$this->out('extra message', 1, Shell::VERBOSE);
$this->verbose('Verbose output');
```

Vous pouvez contrôler le niveau de sortie des shells, en utilisant les options `--quiet` et `--verbose`. Ces options sont ajoutées par défaut, et vous autorise à contrôler la cohérence du niveau de sortie à l'intérieur de vos shells CakePHP.

Les options de console `--quiet` et `--verbose` contrôlent également les données sont loguées sont envoyées à stdout/stderr. Normalement, info et les message de log de plus haut niveau sont envoyés vers stdout/stderr. Lorsque `--verbose` est utilisé, les logs de debug seront envoyés à stdout. Lorsque `--quiet` est utilisé, seulement les warning et messages plus haut seront envoyés à stderr.

Style de sortie

La Style de sortie est fait en incluant les tags - juste comme le HTML - dans votre sortie.

ConsoleOutput remplacera ces tags avec la bonne séquence de code ansi, ou supprimera les tags si vous êtes sur une console qui ne supporte pas les codes ansi. Il y a plusieurs styles intégrés, et vous pouvez en créer plus. Ceux intégrés sont :

- `success` Message de succès. Texte vert.
- `error` Messages d'Erreur. Texte rouge.
- `warning` Messages d'avertissement. Texte jaune.
- `info` Messages d'informations. Texte cyan.
- `comment` Texte supplémentaire. Texte bleu.
- `question` Texte qui est une question, ajouté automatiquement par shell.

Vous pouvez créer des styles supplémentaires en utilisant `$this->stdout->styles()`. Pour déclarer un nouveau style de sortie, vous pouvez faire :

```
$this->_io->styles('flashy', ['text' => 'magenta', 'blink' => true]);
```

Cela vous permettra d'utiliser un tag `<flashy>` dans la sortie de votre shell, et si les couleurs ansi sont activées, ce qui suit sera rendu en texte magenta clignotant `$this->out('<flashy>Whoooa</flashy> Quelque chose a posé problème');`. Quand vous définissez les styles, vous pouvez utiliser les couleurs suivantes pour les attributs `text` et `background` :

- black
- blue
- cyan
- green
- magenta
- red
- white
- yellow

Vous pouvez aussi utiliser les options suivantes en commutateurs booléens, en les définissant à une valeur `true` qui les active.

- blink
- bold
- reverse
- underline

Ajouter un style le rend aussi disponible pour toutes les instances de ConsoleOutput, donc vous n'avez pas à re-déclarer les styles pour les deux objets stdout et stderr.

Enlever la coloration

Bien que la coloration soit vraiment géniale, il peut y avoir des fois où vous voulez l'arrêter, ou forcer à l'avoir :

```
$this->_io->outputAs(ConsoleOutput::RAW);
```

Ce qui est au-dessus met la sortie objet dans un mode de sortie en ligne. Dans le mode de sortie en ligne, il n'y a aucun style du tout. Il y a trois modes que vous pouvez utiliser :

- `ConsoleOutput::COLOR` - La sortie avec couleur enlève les codes en place.
- `ConsoleOutput::PLAIN` - Sortie en texte plein, les tags de style connus seront enlevés de la sortie.
- `ConsoleOutput::RAW` - Sortie en ligne, aucun style ou format ne sera fait C'est un bon mode à utiliser si vous sortez du XML ou si voulez débugger pourquoi votre style ne marche pas.

Par défaut sur les systèmes *nix, les objets ConsoleOutput ont par défaut de la couleur. Sur les systèmes Windows, la sortie simple est mise par défaut sauf si la variable d'environnement `ANSICON` est présente.

Arrêter l'Exécution d'un Shell

Lorsque vos commandes shell ont atteint une condition où vous souhaitez que l'exécution s'arrête, vous pouvez utiliser `abort()` pour lancer une `StopException` qui arrêtera le process :

```
$user = $this->Users->get($this->args[0]);
if (!$user) {
    // Arrête avec un message et un code d'erreur.
    $this->abort('Utilisateur non trouvé', 128);
}
```

Nouveau dans la version 3.2 : La méthode `abort()` a été ajoutée`dans 3.2. Dans les versions précédentes, vous pouvez utiliser `error()` pour afficher un message et arrêter l'exécution.

Statuts et Codes d'Erreur

Les outils en ligne de commande devraient retourner 0 en cas de succès, ou une valeur différente de zéro en cas d'erreur. Puisque les méthodes PHP retournent généralement `true` ou `false`, la fonction `dispatch` du shell Cake permet de régler cela en convertissant vos valeurs de retour `null` et `true` en 0, et toutes les autres valeurs en 1.

La fonction `dispatch` du shell de Cake attrape aussi les `StopException` et utilise la valeur de son code d'exception en code de sortie de l'exception. Comme décrit ci-dessus, vous pouvez utiliser la méthode `abort()` pour afficher un message et sortir avec un code spécifique, ou lancer la `StopException` directement comme montré dans l'exemple :

```
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()
    {
        throw new StopException("", 2);
    }
}
```

L'exemple ci-dessus va retourner les codes de sortie suivants lorsqu'il est exécuté en ligne de commande :

```
$ bin/cake erroneousshell ; echo $?
0
$ bin/cake erroneousshell itFails ; echo $?
1
$ bin/cake erroneousshell itFailsSpecifically ; echo $?
2
```

Astuce : Éviter les codes de sortie 64 - 78, car ils ont une signification spécifique décrite par `sysexits.h`. Éviter les codes de sortie au-dessus de 127, car ils sont utilisés pour indiquer un processus de sortie par signal, tel que SIGKILL ou SIGSEGV.

Note : Vous pouvez en lire plus sur les conventions des codes de sorties dans la page du manuel sysexit sur la plupart des systèmes Unix (`man sysexits`), ou la page d'aide `System Error Codes` dans Windows.

Méthodes Hook

`Cake\Console\Shell::initialize()`
Initialize le constructeur du shell pour les sous-classes et permet la configuration de tâches avant l'exécution du shell.

`Cake\Console\Shell::startup()`
Démarre le Shell et affiche le message d'accueil. Permet de vérifier et de configurer avant de faire la commande ou l'exécution principale.

Astuce : Redéfinit cette méthode si vous voulez retirer l'information de bienvenue, ou sinon modifier le pre-command flow.

Configurer les options et générer de l'aide

class `Cake\Console\ConsoleOptionParser`

`ConsoleOptionParser` helps provide a more familiar command line option and argument parser.

OptionParsers vous permet d'accomplir deux buts en même temps. Premièrement, il vous permet de définir les options et arguments, séparant la validation basique des entrées et votre code. Deuxièmement, il vous permet de fournir de la documentation, qui est utilisée pour bien générer le fichier d'aide formaté.

La console du framework récupère votre parser d'option du shell en appelant `$this->getOptionParser()`. Surcharger cette méthode vous permet de configurer l'OptionParser pour faire correspondre les entrées attendues de votre shell. Vous pouvez aussi configurer les parsers d'option des sous-commandes, ce qui vous permet d'avoir des parsers d'option différents pour les sous-commandes et les tâches. ConsoleOptionParser implémente une interface courant et inclut les méthodes pour configurer les multiple options/arguments en une fois :

```
public function getOptionParser()
{
    $parser = parent::getOptionParser();
    //configure parser
    return $parser;
}
```

Configurer un option parser avec l'interface chainée

Toutes les méthodes utilisées pour configurer le parser peuvent être chainées, vous permettant de définir l'intégralité des options du parser en une unique série d'appel de méthodes :

```
public function getOptionParser()
{
    $parser = parent::getOptionParser();
    $parser->addArgument('type', [
        'help' => 'Either a full path or type of class.'
    ])->addArgument('className', [
        'help' => 'A CakePHP core class name (e.g: Component, HtmlHelper).'
    ])->addOption('method', [
        'short' => 'm',
        'help' => __('The specific method you want help on.')
    ])->setDescription(__('Lookup doc block comments for classes in CakePHP.'));
```

(suite sur la page suivante)

(suite de la page précédente)

```
    return $parser;
}
```

Les méthodes autorisant le chainage sont :

- addArgument()
- addArguments()
- addOption()
- addOptions()
- addSubcommand()
- addSubcommands()
- setCommand()
- setDescription()
- setEpilog()

Définir la description

`Cake\Console\ConsoleOptionParser::setDescription($text)`

La description s'affiche au dessus des arguments et des options. En passant soit un tableau ou une chaîne de caractère, vous pouvez définir la valeur de la description :

```
// Définit plusieurs lignes en une fois
$parser->setDescription(['line one', 'line two']);
// Avant 3.4
$parser->description(['line one', 'line two']);

// Lit la valeur actuelle
$parser->getDescription()
```

src/Shell/ConsoleShell.php est un bon exemple de la méthode `description()` :

```
/**
 * Display help for this console.
 *
 * @return ConsoleOptionParser
 */
public function getOptionParser()
{
    $parser = new ConsoleOptionParser('console');
    $parser->setDescription(
        'This shell provides a REPL that you can use to interact ' .
        'with your application in an interactive fashion. You can use ' .
        'it to run adhoc queries with your models, or experiment ' .
        'and explore the features of CakePHP and your application.' .
        "\n\n" .
        'You will need to have psysh installed for this Shell to work.'
    );
    return $parser;
}
```

La sortie `description` de la console peut être vue en exécutant la commande suivante :

```
$ bin/cake console --help

Welcome to CakePHP v3.0.13 Console
```

(suite sur la page suivante)

(suite de la page précédente)

```
---------------------------------------------------------------
App : src
Path: /home/user/cakeblog/src/
---------------------------------------------------------------
This shell provides a REPL that you can use to interact with your
application in an interactive fashion. You can use it to run adhoc
queries with your models, or experiment and explore the features of
CakePHP and your application.

You will need to have psysh installed for this Shell to work.

Usage:
cake console [-h] [-v] [-q]

Options:

--help, -h     Display this help.
--verbose, -v  Enable verbose output.
--quiet, -q    Enable quiet output.
```

Set a help alias

`Cake\Console\ConsoleOptionParser::setHelpAlias`(*$alias*)

Si vous souhaitez changer le nom de la commande, vous pouvez utiliser la méthode `setHelpAlias()` :

```
$parser->setHelpAlias('my-shell');
```

Cela changera la phrase de "Usage" pour `my-shell` à la place de la valeur par défaut `cake` :

```
Usage:
my-shell console [-h] [-v] [-q]
```

Nouveau dans la version 3.5.0 : La méthode `setHelpAlias` a été ajoutée dans 3.5.0

Définir un « Epilog »

`Cake\Console\ConsoleOptionParser::setEpilog`(*$text = null*)

Récupère ou définit l'epilog pour le parser d'option. L'epilog est affichée après l'argument et l'information d'option. En passant un tableau ou une chaîne, vous pouvez définir la valeur de epilog. L'appeler avec aucun argument va retourner la valeur actuelle :

```
// Définit plusieurs lignes en une fois
$parser->setEpilog(['line one', 'line two']);
// Avant 3.4
$parser->epilog(['line one', 'line two']);

// Lit la valeur actuelle
$parser->getEpilog()
```

Pour illustrer la méthode `epilog()`, ajoutons un appel à la méthode `getOptionParser()` utilisée ci-dessus dans **src/Shell/ConsoleShell.php** :

```
/**
 * Display help for this console.
 *
 * @return ConsoleOptionParser
 */
public function getOptionParser()
{
    $parser = new ConsoleOptionParser('console');
    $parser->setDescription(
        'This shell provides a REPL that you can use to interact ' .
        'with your application in an interactive fashion. You can use ' .
        'it to run adhoc queries with your models, or experiment ' .
        'and explore the features of CakePHP and your application.' .
        "\n\n" .
        'You will need to have psysh installed for this Shell to work.'
    );
    $parser->setEpilog('Thank you for baking with CakePHP!');
    return $parser;
}
```

Le texte ajouté avec la méthode `setEpilog()` peut être vue dans la sortie avec la commande de console suivante :

```
$ bin/cake console --help

Welcome to CakePHP v3.0.13 Console
---------------------------------------------------------------
App : src
Path: /home/user/cakeblog/src/
---------------------------------------------------------------
This shell provides a REPL that you can use to interact with your
application in an interactive fashion. You can use it to run adhoc
queries with your models, or experiment and explore the features of
CakePHP and your application.

You will need to have psysh installed for this Shell to work.

Usage:
cake console [-h] [-v] [-q]

Options:

--help, -h     Display this help.
--verbose, -v  Enable verbose output.
--quiet, -q    Enable quiet output.

Thank you for baking with CakePHP!
```

Ajouter des Arguments

`Cake\Console\ConsoleOptionParser::addArgument($name, $params = [])`

Les arguments de position sont fréquemment utilisés dans les outils en ligne de commande, et `ConsoleOptionParser` vous permet de définir les arguments de position ainsi que de les rendre obligatoires. Vous pouvez ajouter des arguments un à la fois avec `$parser->addArgument();` ou plusieurs à la fois avec `$parser->addArguments();` :

```
$parser->addArgument('model', ['help' => 'The model to bake']);
```

Vous pouvez utiliser les options suivantes lors de la création d'un argument :

- `help` Le texte d'aide à afficher pour cet argument.
- `required` Si le paramètre est obligatoire.
- `index` L'index pour l'argument, si non défini à gauche, l'argument sera mis à la fin des arguments. Si vous définissez le même index deux fois, la première option sera écrasée.
- `choices` Un tableau de choix valides pour cet argument. Si vide à gauche, toutes les valeurs sont valides. Une exception sera lancée quand parse() rencontre une valeur non valide.

Les arguments qui ont été marqués comme nécessaires vont lancer une exception lors du parsing de la commande s'ils ont été omis. Donc vous n'avez pas à gérer cela dans votre shell.

`Cake\Console\ConsoleOptionParser::addArguments(array $args)`

Si vous avez un tableau avec plusieurs arguments, vous pouvez utiliser `$parser->addArguments()` pour ajouter plusieurs arguments en une fois :

```
$parser->addArguments([
    'node', ['help' => 'The node to create', 'required' => true],
    'parent' => ['help' => 'The parent node', 'required' => true]
]);
```

Comme avec toutes les méthodes de construction avec ConsoleOptionParser, addArguments peuvent être utilisés comme des parties d'une chaîne de méthode courante.

Validation des Arguments

Lors de la création d'arguments de position, vous pouvez utiliser le flag `required`, pour indiquer qu'un argument doit être présent quand un shell est appelé. De plus, vous pouvez utiliser `choices` pour forcer un argument pour qu'il soit une liste de choix valides :

```
$parser->addArgument('type', [
    'help' => 'Le type de nœud avec lequel interagir.',
    'required' => true,
    'choices' => ['aro', 'aco']
]);
```

Ce qui est au-dessus va créer un argument qui est nécessaire et a une validation sur l'entrée. Si l'argument est soit manquant, soit a une valeur incorrecte, une exception sera levée et le shell sera arrêté.

Ajouter des Options

`Cake\Console\ConsoleOptionParser::addOption($name, $options = [])`

Les options ou les flags sont aussi fréquemment utilisés avec les outils de ligne de commande. `ConsoleOptionParser` supporte la création d'options avec les deux verbose et short aliases, fournissant les valeurs par défaut et créant des switches en booléen. Les options sont créées avec soit `$parser->addOption()`, soit `$parser->addOptions()` :

```
$parser->addOption('connection', [
    'short' => 'c'
    'help' => 'connection',
    'default' => 'default'
]);
```

Ce qui est au-dessus vous permet l'utilisation soit de `cake myshell --connection=other`, soit de `cake myshell --connection other`, ou soit de `cake myshell -c other` lors de l'appel au shell. Vous pouvez aussi créer des switches de booléen, ces switches ne consomment pas de valeurs, et leur présence les active juste dans les paramètres parsés :

```
$parser->addOption('no-commit', ['boolean' => true]);
```

Avec cette option, lors de l'appel d'un shell comme `cake myshell --no-commit something` le paramètre no-commit aurait une valeur à true, et "something" serait traité comme un argument de position. Les options intégrées `--help`, `--verbose`, et `--quiet` utilisent cette fonctionnalité.

Lors de la création d'options, vous pouvez utiliser les options suivantes pour définir le comportement de l'option :

- `short` - La variante de la lettre unique pour cette option, laissez à non définie pour n'en avoir aucun.
- `help` - Le texte d'aide pour cette option. Utilisé lors de la génération d'aide pour l'option.
- `default` - La valeur par défaut pour cette option. Si elle n'est pas définie, la valeur par défaut sera `true`.
- `boolean` - L'option n'utilise aucune valeur, c'est juste un switch de booléen. Par défaut à `false`.
- `choices` - Un tableau de choix valides pour cette option. Si elle est vide, toutes les valeurs sont valides. Une exception sera lancée lorsque parse() rencontre une valeur non valide.

`Cake\Console\ConsoleOptionParser::addOptions(array $options)`

Si vous avez un tableau avec plusieurs options, vous pouvez utiliser `$parser->addOptions()` pour ajouter plusieurs options en une fois :

```
$parser->addOptions([
    'node', ['short' => 'n', 'help' => 'The node to create'],
    'parent' => ['short' => 'p', 'help' => 'The parent node']
]);
```

Comme avec toutes les méthodes de construction de ConsoleOptionParser, addOptions peut être utilisée comme une partie de la chaîne de méthode courante.

Les valeurs des options sont stockées dans le tableau `$this->params`. Vous pouvez également utiliser la méthode `$this->param()` pour éviter les erreurs lorsque vous essayez d'accéder à une option qui n'est pas présente.

Validation des Options

Les options peuvent être fournies avec un ensemble de choix un peu comme les arguments de position peuvent l'être. Quand une option a défini les choix, ceux-ci sont les seuls choix valides pour une option. Toutes les autres valeurs vont lancer une `InvalidArgumentException` :

```
$parser->addOption('accept', [
    'help' => 'What version to accept.',
    'choices' => ['working', 'theirs', 'mine']
]);
```

Utiliser les Options Booléennes

Les options peuvent être définies en options booléennes, qui sont utiles quand vous avez besoin de créer des options de flag. Comme les options par défaut, les options booléennes les incluent toujours dans les paramètres parsés. Quand les flags sont présents, ils sont définis à `true`, quand ils sont absents à `false` :

```
$parser->addOption('verbose', [
    'help' => 'Enable verbose output.',
    'boolean' => true
]);
```

L'option suivante fera que `$this->params['verbose']` sera toujours disponible. Cela vous permet d'oublier `empty()` ou `isset()` pour vérifier les flags de booléens :

```
if ($this->params['verbose']) {
    // faire quelque chose
}
```

Puisque les options booléennes sont toujours définies à `true` ou à `false`, vous pouvez omettre les méthodes de vérification supplémentaires lorsque vous utilisez l'accès via le tableau. La méthode `$this->param()` rend ces vérifications inutiles dans tous les cas.

Ajouter des sous-commandes

`Cake\Console\ConsoleOptionParser::addSubcommand`(*$name*, *$options = []*)

Les applications de Console sont souvent faites de sous-commandes, et ces sous-commandes nécessiteront un parsing d'options spéciales et ont leur propre aide. Un exemple parfait de cela est `bake`. Bake est fait de plusieurs tâches séparées qui ont toutes leur propre aide et options. `ConsoleOptionParser` vous permet de définir les sous-commandes et de fournir les parsers d'option spécifiques donc le shell sait comment parser les commandes pour ses tâches :

```
$parser->addSubcommand('model', [
    'help' => 'Bake a model',
    'parser' => $this->Model->getOptionParser()
]);
```

Ce qui est au-dessus est un exemple de la façon dont vous pouvez fournir de l'aide et un parser d'option spécialisé pour une tâche du shell. En appelant le `getOptionParser()` de la tâche, nous n'avons pas à dupliquer la génération du parser d'option, ou mixer les tâches concernées dans notre shell. Ajoutez des sous-commandes de cette façon a deux avantages. Premièrement, cela laisse votre shell documenter ces sous-commandes dans l'aide générée, et cela vous permet aussi un accès facile à l'aide de la sous-commande. Avec la sous-commande créée ci-dessus, vous pouvez appeler `cake myshell --help` et regarder la liste des sous-commandes, et aussi lancer `cake myshell model --help` pour voir l'aide uniquement pour la tâche model.

Note : Une fois que votre Shell définit des sous-commandes, toutes les sous-commandes doivent être explicitement définies.

Quand vous définissez une sous-commande, vous pouvez utiliser les options suivantes :

- `help` - Le texte d'aide pour la sous-commande.
- `parser` - Un ConsoleOptionParser pour la sous-commande. Cela vous permet de créer des méthodes de parsers d'options spécifiques. Quand l'aide est générée pour une sous-commande, si un parser est présent, il sera utilisé. Vous pouvez aussi fournir le parser en tableau qui est compatible avec `Cake\Console\ConsoleOptionParser::buildFromArray()`

Ajouter des sous-commandes peut être fait comme une partie de la chaîne de méthode courante.

Modifié dans la version 3.5.0 : Lorsque vous ajouter des sous-commandes composées de plusieurs mots, vous pouvez maintenant les appeler en `snake_case` en plus de la forme en camelBack.

Construire un ConsoleOptionParser à partir d'un Tableau

`Cake\Console\ConsoleOptionParser::`**`buildFromArray`**(*$spec*)

Comme mentionné précédemment, lors de la création de parsers d'option de la sous-commande, vous pouvez définir le parser spec en tableau pour cette méthode. Ceci peut faciliter la construction de parsers de sous-commande, puisque tout est un tableau :

```
$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')]
        ]
    ]
]);
```

A l'intérieur du parser spec, vous pouvez définir les clés pour *arguments*`, `options`, `description` et `epilog`. Vous ne pouvez pas définir les `sous-commandes` dans un constructeur de type tableau. Les valeurs pour les arguments, et les options, doivent suivre le format que `Cake\Console\ConsoleOptionParser::addArguments()` et `Cake\Console\ConsoleOptionParser::addOptions()` utilisent. Vous pouvez aussi utiliser buildFromArray lui-même, pour construire un parser d'option :

```
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' => [
```

(suite sur la page suivante)

(suite de la page précédente)

```
            'aro' => ['help' => __('ARO to check.'), 'required' => true],
            'aco' => ['help' => __('ACO to check.'), 'required' => true],
            'action' => ['help' => __('Action to check')]
        ]
    ]);
}
```

Fusionner les ConsoleOptionParsers

`Cake\Console\ConsoleOptionParser::merge($spec)`

Lorsque vous construisez un groupe de commandes, vous voudrez peut-être combiner plusieurs parsers :

```
$parser->merge($anotherParser);
```

Notez que l'ordre des arguments de chaque parser doit être identique, et que les options doivent être compatibles pour que cela fonctionne. N'utilisez donc pas les mêmes clés pour des choses différentes.

Obtenir de l'Aide dans les Shells

Avec l'ajout de ConsoleOptionParser, récupérer l'aide des shells est faite d'une façon cohérente et uniforme. En utilisant l'option `--help` ou `-h`, vous pouvez voir l'aide pour tout shell du cœur, et tout shell qui implémente un ConsoleOptionParser :

```
cake bake --help
cake bake -h
```

Les deux généreraient l'aide pour bake. Si le shell supporte les sous-commandes, vous pouvez obtenir de l'aide pour ceux-là d'un façon similaire :

```
cake bake model --help
cake bake model -h
```

Cela vous ramènera l'aide spécifique pour la tâche model de bake.

Obtenir de l'Aide en XML

Lorsque vous réalisez des outils d'automatisation ou de développement qui ont besoin d'interagir avec les shells de CakePHP, il est appréciable d'obtenir de l'aide dans un format parsable par une machine. ConsoleOptionParser peut fournir de l'aide au format XML en définissant un argument supplémentaire :

```
cake bake --help xml
cake bake -h xml
```

Les commandes ci-dessus vont retourner un document XML contenant de l'aide à propos des options, arguments et sous-commandes du shell sélectionné. Voici un exemple de documentation :

```
<?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>
```

(suite sur la page suivante)

(suite de la page précédente)

```
    <epilog>
        Omitting all arguments and options will enter into an interactive
        mode.
    </epilog>
    <subcommands/>
    <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="">
            <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>
```

Renommer des Commandes

Par défaut, CakePHP va automatiquement chercher et mettre à disposition toutes les commandes dans votre application et ses plugins. Il est possible que vous souhaitiez réduire le nombre de commandes exposées lorsque vous construisez une application console indépendante. Pour cela, vous pouvez utiliser le hook `console()` de votre Application pour limiter le nombre de commandes qui sont exposées :

```
namespace App;

use App\Shell\UserShell;
use App\Shell\VersionShell;
```

(suite sur la page suivante)

(suite de la page précédente)

```
use Cake\Http\BaseApplication;

class Application extends BaseApplication
{
    public function console($commands)
    {
        // Ajout par nom de de classe
        $commands->add('user', UserShell::class);

        // Ajout par une instance
        $commands->add('version', new VersionShell());

        return $commands;
    }
}
```

Dans l'exemple ci-dessus, les seules commandes qui seront disponibles seront `help`, `version` and `user`.

Nouveau dans la version 3.5.0 : Le hook `console` a été ajouté.

Routing dans shells / CLI

Dans l'interface en ligne de commande (CLI), spécialement dans vos shells et tasks, `env('HTTP_HOST')` et les autres variables d'environnement spécifiques à votre navigateur ne sont pas définies.

Si vous générez des rapports ou envoyez des emails qui utilisent `Router::url()`, ceux-ci vont contenir l'hôte par défaut `http://localhost/` et cela va entrainer des URLs invalides. Dans ce cas, vous devrez spécifier le domaine manuellement. Vous pouvez faire cela en utilisant la valeur de Configure `App.fullBaseUrl` de votre bootstrap ou config, par exemple.

Pour envoyer des emails, vous devrez fournir à la classe Email l'hôte avec lequel vous souhaitez envoyer l'email en faisant :

```
use Cake\Mailer\Email;

$email = new Email();
// Avant 3.4 utilisez domain()
$email->setDomain('www.example.org');
```

Cela suppose que les ID du message généré sont valides et correspondent au domaine duquel les emails sont envoyés.

Plus de sujets

Shell Helpers

Nouveau dans la version 3.1 : Les Shell Helpers ont été ajoutés dans la version 3.1.0

Les Shell Helpers vous permettent d'empaqueter une logique de sortie complexe d'une manière réutilisable. Les Shell Helpers sont accessibles et utilisables depuis n'importe quel shell ou task :

```
// Affiche des données sous forme de tableau.
$this->helper('Table')->output($data);
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Récupère un helper depuis un plugin.
$this->helper('Plugin.HelperName')->output($data);
```

Vous pouvez également récupérer des instances de helpers et appeler n'importe quelle méthode publique :

```
// Récupère et utilise le Progress Helper.
$progress = $this->helper('Progress');
$progress->increment(10);
$progress->draw();
```

Créer des Helpers

Alors que CakePHP fournit quelques shell helpers, vous pouvez en créer d'autres dans vos applications ou plugins. Par example, nous allons créer un simple helper pour générer des en-têtes. Tout d'abord, créez **src/Shell/Helper/HeadingHelper.php** et mettez-y le contenu suivant :

```
<?php
namespace App\Shell\Helper;

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 ensuite utiliser ce nouvel helper dans une de nos commandes shell en l'appelant :

```
// Avec ### de chaque côté
$this->helper('Heading')->output(['It works!']);

// Avec ~~~~ de chaque côté
$this->helper('Heading')->output(['It works!', '~', 4]);
```

Les helpers implémentent généralement la méthode `output()` qui prend un tableau de paramètres. Cependant, comme les helpers de Console sont des classes vanilla elles peuvent implémenter des méthodes supplémentaires qui prennent n'importe quelle forme d'argument.

Helpers Fournis

Helper Table

Le TableHelper vous aide à créer des tableaux en art ASCII bien formatés. L'utiliser est relativement simple :

```
$data = [
    ['Header 1', 'Header', 'Long Header'],
    ['short', 'Longish thing', 'short'],
    ['Longer thing', 'short', 'Longest Value'],
```

(suite sur la page suivante)

(suite de la page précédente)

```
];
$this->helper('Table')->output($data);

// Génère
+--------------+---------------+---------------+
| Header 1     | Header        | Long Header   |
+--------------+---------------+---------------+
| short        | Longish thing | short         |
| Longer thing | short         | Longest Value |
+--------------+---------------+---------------+
```

Helper Progress

Le ProgessHelper peut être utilisé de deux manières différentes. Le mode simple vous permet de fournir une méthode de rappel qui est invoquée jusqu'à ce que l'avancement soit complété :

```
$this->helper('Progress')->output(['callback' => function ($progress) {
    // Fait quelque chose ici.
    $progress->increment(20);
    $progress->draw();
}]);
```

Vous pouvez mieux contrôler la barre d'avancement en fournissant des options supplémentaires :

- `total` Le nombre total d'items dans la barre d'avancement. 100 par défaut.
- `width` Largeur de la barre d'avancement. 80 par défaut.
- `callback` la méthode de rappel qui sera appelée dans une boucle pour faire avancer la barre.

Un exemple d'utilisation de toutes les options serait :

```
$this->helper('Progress')->output([
    'total' => 10,
    'width' => 20,
    'callback' => function ($progress) {
        $progress->increment(2);
        $progress->draw();
    }
]);
```

Le helper progress peut également être utilisé manuellement pour incrémenter et re-rendre la barre d'avancement si besoin :

```
$progress = $this->helper('Progress');
$progress->init([
    'total' => 10,
    'width' => 20,
]);

$progress->increment(4);
$progress->draw();
```

Console Interactive (REPL)

Le squelette de l'application CakePHP intègre un REPL(Read Eval Print Loop = Lire Evaluer Afficher Boucler) qui facilite 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`.

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 une fois de temps en temps ou pour envoyer des newsletters :

```
  */5  *    *    *    *  cd /full/path/to/root && bin/cake myshell myparam
# *    *    *    *    *  command to execute
# |    |    |    |    |
# |    |    |    |    |
# |    |    |    |    \------ day of week (0 - 6) (0 to 6 are Sunday to Saturday,
# |    |    |    |           or use names)
# |    |    |    \------------ month (1 - 12)
# |    |    \----------------- day of month (1 - 31)
```

(suite sur la page suivante)

(suite de la page précédente)

```
# |     \──────────────────────── hour (0 - 23)
# \──────────────────────────────── min (0 - 59)
```

Vous pouvez avoir plus d'infos ici : http://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 /full/path/to/root && bin/cake myshell myparam` pourrait ne pas fonctionner. Vous pouvez à la place utiliser `php /full/path/to/root/bin/cake.php myshell myparam`.

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 /full/path/to/root/bin/cake.php myshell myparam`.

Shell I18N

Les fonctionnalités i18n de CakePHP utilisent les fichiers po [137] comme source de traduction. Cela les rend facile à intégrer avec des outils tels que Poedit [138] ou d'autres outils habituels de traduction.

Le Shell i18n est une façon rapide et simple 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.

137. http://fr.wikipedia.org/wiki/GNU_gettext
138. http://www.poedit.net/

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
```

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
```

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

    if [[ ${COMP_CWORD} = 2 ]] ; then
        opts=$($(cake} Completion subcommands $prev)
        COMPREPLY=( $(compgen -W "${opts}" -- ${cur}) )
```

(suite sur la page suivante)

(suite de la page précédente)

```
        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 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
```

Nouveau dans la version 3.4.0 : A partir de 3.4.0, l'option `--cli` est supportée.

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

Nouveau dans la version 3.1 : RoutesShell a été ajoutée dans 3.1

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
```

Shell de Mise à Jour

La mise à jour shell va faire quasiment tout le boulot pour mettre à jour vos applications cakePHP de la version 2.x à 3.x.

Cette mise à jour est fournie par le plugin Upgrade [139]. Merci de vous référer au fichier README pour avoir toutes les informations sur la façon de mettre à jour votre application.

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
```

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
```

Nouveau dans la version 3.3.0 : The cache shell was added in 3.3.0

139. https://github.com/cakephp/upgrade

Shell du Cache de l'ORM

Obsolète depuis la version 3.6.0 : OrmCacheShell est remplacé par *Shell du Cache du Schéma*

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`.

Nouveau dans la version 3.3.0 : Utiliser cette méthode retournera la valeur de la variable `$var` passée en paramètre, vous permettant ainsi, par exemple, de l'utiliser sur un `return` :

```
return debug($data); // Retournera $data dans tous les cas
```

stackTrace()

La fonction `stackTrace()` est globalement disponible, et vous permet d'afficher une stack trace quelque soit la fonction appelée.

breakpoint()

Nouveau dans la version 3.1.

Si vous avez installé Psysh [140] vous pouvez utiliser cette fonction dans les environnements CLI pour ouvrir une console interactive avec le scope local courant :

140. http://psysh.org/

```
// 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.

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',
]);
```

Nouveau dans la version 3.4.0 : Les masques d'affichage ont été ajoutés dans la version 3.4.0

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
```

(suite sur la page suivante)

(suite de la page précédente)

```
(
    [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 terminée, ou même avant que vous souhaitiez la déployer, il y a certains points à vérifier.

Déplacer les Fichiers

Nous vous incitons à créer un git commit et de faire un pull ou un clone du commit ou du répertoire sur votre serveur et de lancer `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 version de dev.

Modifier le fichier config/app.php

Mettre à jour app.php, spécialement la valeur de `debug` est extrêmement important. 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.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 component *Cross Site Request Forgery* 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*.

CHAPITRE 18

Email

Avertissement : Avant la version 3.1, les classes `Email` et `Transport` étaient sous le namespace `Cake\Network\Email` au lieu du namespace `Cake\Mailer`.

class `Cake\Mailer\`**`Email`** (*mixed $profile = null*)

`Email` est une nouvelle classe 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\Email;
```

Après avoir chargé `Email`, vous pouvez envoyer un email avec ce qui suit :

```
$email = new Email('default');
$email->from(['me@example.com' => 'My Site'])
    ->to('you@example.com')
    ->subject('About')
    ->send('My message');
```

Puisque les méthodes de setter d'`Email` retournent l'instance de la classe, vous pouvez définir ses propriétés avec le chaînage des méthodes.

`Email` comporte plusieurs méthodes pour définir les destinataires - `to()`, `cc()`, `bcc()`, `addTo()`, `addCc()` et `addBcc()`. La principale différence est que les trois premières méthodes vont réinitialiser ce qui était déjà défini et les suivantes vont ajouter plus de destinataires dans leur champs respectifs :

```
$email = new Email();
$email->to('to@example.com', 'To Example');
$email->addTo('to2@example.com', 'To2 Example');
// Les destinaitres de l'email sont: to@example.com et to2@example.com
$email->to('test@example.com', 'ToTest Example');
// Le destinataire de l'email est: test@example.com
```

Obsolète depuis la version 3.4.0 : Utilisez `setFrom()`, `setTo()`, `setCc()` , `setBcc()` et `setSubject()` à la place.

Choisir l'émetteur

Quand on envoie des emails de la part d'autre personne, c'est souvent une bonne idée de définir l'émetteur original en utilisant le header Sender. Vous pouvez faire ceci en utilisant `sender()` :

```
$email = new Email();
$email->sender('app@example.com', 'MyApp emailer');
```

Note : C'est aussi une bonne idée de définir l'envelope de l'émetteur quand on envoie un mail de la part d'une autre personne. Cela les empêche d'obtenir tout message sur la délivrance.

Obsolète depuis la version 3.4.0 : Utilisez plutôt `setSender()`.

Configuration

La Configuration par défaut pour `Email` est créée en utilisant `config()` et `configTransport()`. Vous devrez mettre vos préconfigurations d'email dans le fichier **config/app.php**. Le fichier **config/app.default.php** est un exemple de ce fichier. Il n'est pas nécessaire de définir de configuration d'email dans **config/app.php**. `Email` peut être utilisé sans cela et utilise les méthodes séparément pour définir toutes les configurations ou charger un tableau de configs.

En définissant des profiles et des transports, vous pouvez garder le code de votre application sans données de configuration, et éviter de dupliquer, ce qui rend la maintenance et le déploiement moins compliqués.

Pour charger une configuration prédéfinie, vous pouvez utiliser la méthode `profile()` ou la passer au constructeur d'"`Email` :

```
$email = new Email();
$email->profile('default');

//ou dans le constructeur::
$email = new Email('default');
```

Plutôt que de passer une chaîne avec le bon nom de configuration prédéfini, vous pouvez aussi juste charger un tableau d'options :

```
$email = new Email();
$email->profile(['from' => 'me@example.org', 'transport' => 'my_custom']);

//or dans le constructeur::
$email = new Email(['from' => 'me@example.org', 'transport' => 'my_custom']);
```

Modifié dans la version 3.1 : Le profil d'email `default` est automatiquement défini quand une instance `Email` est créée.

Obsolète depuis la version 3.4.0 : Utilisez `setProfile()` à la place de `profile()`.

Configurer les Transports

static `Cake\Mailer\Email::configTransport`(*$key*, *$config = null*)

Les messages d'Email sont délivrés par les transports. Différents transports vous permettent d'envoyer les messages par la fonction `mail()` de PHP, les serveurs SMTP, ou aucun d'eux ce qui peut être utile pour débugger. La configuration des transports vous permet de garder les données de configuration en dehors du code de votre application et rend le déploiement plus simple puisque vous pouvez simplement changer les données de configuration. Un exemple de configuration des transports ressemblerai à ceci :

```
use Cake\Mailer\Email;

// Exemple de configuration de Mail
Email::configTransport('default', [
    'className' => 'Mail'
]);

// Exemple de configuration SMTP.
Email::configTransport('gmail', [
    'host' => 'ssl://smtp.gmail.com',
    'port' => 465,
    'username' => 'my@gmail.com',
    'password' => 'secret',
    'className' => 'Smtp'
]);
```

Vous pouvez configurer les serveurs SSL SMTP, comme Gmail. pour faire ceci, mettez le prefix `ssl://` dans l'hôte et configurez le port avec la bonne valeur. Vous pouvez aussi activer TLS SMTP en utilisant l'option `tls` :

```
use Cake\Mailer\Email;

Email::configTransport('gmail', [
    'host' => 'smtp.gmail.com',
    'port' => 587,
    'username' => 'my@gmail.com',
    'password' => 'secret',
    'className' => 'Smtp',
    'tls' => true
]);
```

La configuration ci-dessus va activer la communication TLS pour tous les messages d'email.

Avertissement : Vous devrez avoir l'accès aux applications moins sécurisées activé dans votre compte Google pour que cela fonctionne : Autoriser les applications moins sécurisées à accéder à votre compte [141].

Note : Pour utiliser SSL + SMTP, vous devrez avoir SSL configuré dans votre installation PHP.

141. https://support.google.com/accounts/answer/6010255

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* :

```
Email::configTransport('default', [
    'url' => 'smtp://my@gmail.com:secret@smtp.gmail.com:465?tls=true',
]);
```

Lorsque vous utilisez une chaine DSN, vous pouvez définir des paramètres/options supplémentaires en tant qu'arguments de query string.

Obsolète depuis la version 3.4.0 : Utilisez `setConfigTransport()` à la place de `configTransport()`.

static `Cake\Mailer\Email::`**`dropTransport`**(*$key*)

Une fois configuré, les transports ne peuvent pas être modifiés. Afin de modifier un transport, vous devez d'abord le supprimer et le reconfigurer.

Profiles de Configurations

Définir des profiles de délivrance vous permet d'ajouter les configurations habituelles d'email dans des profiles réutilisables. Votre application peut avoir autant de profiles que nécessaire. Les clés de configuration suivantes sont utilisées :

- `'from'` : Email ou un tableau d'emmeteur. Regardez `Email::from()`.
- `'sender'` : Email ou un tableau d'émetteur réel. Regardez `Email::sender()`.
- `'to'` : Email ou un tableau de destination. Regardez `Email::to()`.
- `'cc'` : Email ou un tableau de copy carbon. Regardez `Email::cc()`.
- `'bcc'` : Email ou un tableau de copy carbon blind. Regardez `Email::bcc()`.
- `'replyTo'` : Email ou un tableau de répondre à cet e-mail. Regardez `Email::replyTo()`.
- `'readReceipt'` : Adresse Email ou un tableau d'adresses pour recevoir un récepissé de lecture. Regardez `Email::readReceipt()`.
- `'returnPath'` : Adresse Email ou un tableau des adresses à retourner si vous avez une erreur. Regardez `Email::returnPath()`.
- `'messageId'` : ID du Message de l'e-mail. Regardez `Email::messageId()`.
- `'subject'` : Sujet du message. Regardez `Email::subject()`.
- `'message'` : Contenu du message. Ne définissez pas ce champ si vous utilisez un contenu rendu.
- `'priority'` : Priorité de l'email, exprimée avec un nombre (généralement de 1 à 5, 1 étant la priorité la plus haute).
- `'headers'` : Headers à inclure. Regardez `Email::headers()`.
- `'viewRender'` : Si vous utilisez un contenu rendu, définissez le nom de classe de la vue. Regardez `Email::viewRenderer()`.
- `'template'` : Si vous utilisez un contenu rendu, définissez le nom du template. Regardez `Email::template()`.
- `'theme'` : Theme utilisé pour le rendu du template. Voir `Email::theme()`.
- `'layout'` : Si vous utilisez un contenu rendu, définissez le layout à rendre. Si vous voulez rendre un template sans layout, définissez ce champ à null. Regardez `Email::template()`.
- `'viewVars'` : Si vous utilisez un contenu rendu, définissez le tableau avec les variables devant être rendus dans la vue. Regardez `Email::viewVars()`.
- `'attachments'` : Liste des fichiers à attacher. Regardez `Email::attachments()`.
- `'emailFormat'` : Format de l'email (html, text ou both). Regardez `Email::emailFormat()`.
- `'transport'` : Nom du Transport. Regardez *`Mailer\Email::configTransport()`*.
- `'log'` : Niveau de Log pour connecter les headers de l'email headers et le message. `true` va utiliser LOG_DEBUG. Regardez aussi *Utiliser les Niveaux*.
- `'helpers'` : Tableau de helpers utilisés dans le template email. `Email::helpers()`.

Toutes ces configurations sont optionnelles, excepté `'from'`.

Note : Les valeurs des clés ci-dessus utilisant Email ou un tableau, comme from, to, cc etc... seront passées en

premier paramètre des méthodes correspondantes. L'equivalent pour `Email::from('my@example.com', 'My Site')` sera défini comme `'from' => ['my@example.com' => 'My Site']` dans votre config.

Définir les Headers

Dans `Email`, vous êtes libre de définir les headers que vous souhaitez. Si vous migrez pour utiliser Email, n'oubliez pas de mettre le préfixe `X-` dans vos headers.

Regardez `Email::headers()` et `Email::addHeaders()`

Obsolète depuis la version 3.4.0 : Utilisez `setHeaders()` à la place de `headers()`.

Envoyer les Emails Templatés

Les Emails sont souvent bien plus que de simples message textes. Afin de faciliter cela, CakePHP fournit une façon d'envoyer les emails en utilisant la *view layer* de CakePHP.

Les templates pour les emails se placent dans un dossier spécial appelé `Email` dans le répertoire `Template` de votre application. Les templates des emails peuvent aussi utiliser les layouts et éléments tout comme les templates normales :

```
$email = new Email();
$email
    ->template('welcome', 'fancy')
    ->emailFormat('html')
    ->to('bob@example.com')
    ->from('app@domain.com')
    ->send();
```

Ce qui est au-dessus utilise **src/Template/Email/html/welcome.ctp** pour la vue, et **src/Template/Layout/Email/html/fancy.ctp** pour le layout. Vous pouvez aussi envoyer des messages email templaté multipart :

```
$email = new Email();
$email
    ->template('welcome', 'fancy')
    ->emailFormat('both')
    ->to('bob@example.com')
    ->from('app@domain.com')
    ->send();
```

Ceci utiliserait les fichiers de template suivants :

- **src/Template/Email/text/welcome.ctp**
- **src/Template/Layout/Email/text/fancy.ctp**
- **src/Template/Email/html/welcome.ctp**
- **src/Template/Layout/Email/html/fancy.ctp**

Quand on envoie les emails templatés, vous avez la possibilité d'envoyer soit `text`, `html` soit `both`.

Vous pouvez définir des variables de vue avec `Email::viewVars()` :

```
$email = new Email('templated');
$email->viewVars(['value' => 12345]);
```

Dans votre email template, vous pouvez utiliser ceux-ci avec :

```
<p>Ici est votre valeur: <b><?= $value; ?></b></p>
```

Vous pouvez aussi utiliser les helpers dans les emails, un peu comme vous pouvez dans des fichiers de template normaux. Par défaut, seul `HtmlHelper` est chargé. Vous pouvez chargez des helpers supplémentaires en utilisant la méthode `helpers()` :

```
$email->helpers(['Html', 'Custom', 'Text']);
```

Quand vous définissez les helpers, assurez vous d'inclure "Html" ou il sera retiré des helpers chargés dans votre template d'email.

Si vous voulez envoyer un email en utilisant templates dans un plugin, vous pouvez utiliser la *syntaxe de plugin* familière pour le faire :

```
$email = new Email();
$email->template('Blog.new_comment', 'Blog.auto_message');
```

Ce qui est au-dessus utiliserait les templates à partir d'un plugin de Blog par exemple.

Dans certains cas, vous devez remplacer le template par défaut fourni par les plugins. Vous pouvez faire ceci en utilisant les themes en disant à Email d'utiliser le bon theme en utilisant la méthode `Email::theme()` :

```
$email = new Email();
$email->template('Blog.new_comment', 'Blog.auto_message');
$email->theme('TestTheme');
```

Ceci vous permet de remplacer le template *new_comment* dans votre theme sans modifier le plugin Blog. Le fichier de template devra être créé dans le chemin suivant : **src/Template/Plugin/TestTheme/Plugin/Blog/Email/text/new_comment.ctp**.

Obsolète depuis la version 3.4.0 : Utilisez `setTemplate()` à la place de `template()`. Utilisez `setLayout()` à la place de l'argument layout de `template()`. Utilisez ``setTheme()` à la place de `theme()`.

Envoyer les pièces jointes

`Cake\Mailer\Email::`**`attachments`**(*$attachments = null*)

Vous pouvez aussi attacher des fichiers aux messages d'email. Il y a quelques formats différents qui dépendent de quel type de fichier vous avez, et comment vous voulez que les noms de fichier apparaissent dans le mail de réception du client :

1. Chaîne de caractères : `$email->attachments('/full/file/path/file.png')` va attacher ce fichier avec le nom file.png.
2. Tableau : `$email->attachments(['/full/file/path/file.png'])` aura le même comportement qu'en utilisant une chaîne de caractères.
3. Tableau avec clé : `$email->attachments(['photo.png' => '/full/some_hash.png'])` va attacher some_hash.png avec le nom photo.png. Le récipiendaire va voir photo.png, pas some_hash.png.
4. Tableaux imbriqués :

```
$email->attachments([
    'photo.png' => [
        'file' => '/full/some_hash.png',
        'mimetype' => 'image/png',
        'contentId' => 'my-unique-id'
    ]
]);
```

Ce qui est au-dessus va attacher le fichier avec différent mimetype et avec un content ID personnalisé (Quand vous définissez le content ID, la pièce jointe est transformée en inline). Le mimetype et contentId sont optionnels dans ce formulaire.

4.1. Quand vous utilisez `contentId`, vous pouvez utiliser le fichier dans corps HTML comme `<img src="cid:my-content-id">`.

4.2. 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.

4.3 Au lieu de l'option `file`, vous pouvez fournir les contenus de fichier en chaîne en utilisant l'option `data`. Cela vous permet d'attacher les fichiers sans avoir besoin de chemins de fichier vers eux.

Obsolète depuis la version 3.4.0 : Utilisez `setAttachments()` à la place de `attachments()`.

Utiliser les Transports

Les Transports sont des classes destinées à envoyer l'email selon certain protocoles ou méthodes. CakePHP supporte les transports Mail (par défaut), Debug et SMTP.

Pour configurer votre méthode, vous devez utiliser la méthode `Cake\Mailer\Email::transport()` ou avoir le transport dans votre configuration :

```
$email = new Email();

// Use a named transport already configured using Email::setConfigTransport()
$email->transport('gmail');

// Use a constructed object.
$transport = new DebugTransport();
$email->transport($transport);
```

Obsolète depuis la version 3.4.0 : Utilisez `setTransport()` à la place de `transport()`

Créer des Transports Personnalisés

Vous pouvez créer vos transports personnalisés pour intégrer avec d'autres systèmes email (comme SwiftMailer). Pour créer votre transport, créez tout d'abord le fichier **src/Mailer/Transport/ExampleTransport.php** (où Exemple est le nom de votre transport). Pour commencer, votre fichier devrait ressembler à cela :

```
namespace App\Mailer\Transport;

use Cake\Mailer\AbstractTransport;
use Cake\Mailer\Email;

class ExampleTransport extends AbstractTransport
{
    public function send(Email $email)
    {
        // Logique d'exécution
    }
}
```

Vous devez intégrer la méthode `send(Email $email)` avec votre logique personnalisée. En option, vous pouvez intégrer la méthode `config($config)`. `config()` est appelée avant send() et vous permet d'accepter les configurations de l'utilisateur. Par défaut, cette méthode met la configuration dans l'attribut protégé `$_config`.

Si vous avez besoin d'appeler des méthodes supplémentaires sur le transport avant l'envoi, vous pouvez utiliser `Cake\Mailer\Email::transportClass()` pour obtenir une instance du transport. Exemple :

```
$yourInstance = $email->transport('your')->transportClass();
$yourInstance->myCustomMethod();
$email->send();
```

Faciliter les Règles de Validation des Adresses

`Cake\Mailer\Email::`**`emailPattern`**(*$pattern = null*)

Si vous avez des problèmes de validation lors de l'envoi vers des adresses non conformes, vous pouvez faciliter le patron utilisé pour valider les adresses email. C'est parfois nécessaire quand il s'agit de certains ISP Japonais :

```
$email = new Email('default');

// Relax le patron d'email, ainsi vous pouvez envoyer
// vers des adresses non conformes
$email->emailPattern($newPattern);
```

Obsolète depuis la version 3.4.0 : Utilisez `setEmailPattern()` à la place de `emailPattern()`.

Envoyer des Messages Rapidement

Parfois vous avez besoin d'une façon rapide d'envoyer un email, et vous n'avez pas particulièrement envie en même temps de définir un tas de configuration. `Cake\Mailer\Email::deliver()` est présent pour ce cas.

Vous pouvez créer votre configuration dans `Cake\Mailer\Email::config()`, ou utiliser un tableau avec toutes les options dont vous aurez besoin et utiliser la méthode statique `Email::deliver()`. Exemple :

```
Email::deliver('you@example.com', 'Subject', 'Message', ['from' => 'me@example.com']);
```

Cette méthode va envoyer un email à you@example.com, à partir de me@example.com avec le sujet Subject et le contenu Message.

Le retour de `deliver()` est une instance de *`Cake\Mailer\Email`* avec l'ensemble des configurations. Si vous ne voulez pas envoyer l'email maintenant, et souhaitez configurer quelques trucs avant d'envoyer, vous pouvez passer le 5ème paramètre à `false`.

Le 3ème paramètre est le contenu du message ou un tableau avec les variables (quand on utilise le contenu rendu).

Le 4ème paramètre peut être un tableau avec les configurations ou une chaîne de caractères avec le nom de configuration dans `Configure`.

Si vous voulez, vous pouvez passer les to, subject et message à null et faire toutes les configurations dans le 4ème paramètre (en tableau ou en utilisant `Configure`). Vérifiez la liste des *configurations* pour voir toutes les configs acceptées.

Envoyer des Emails depuis CLI

Quand vous envoyez des emails à travers un script CLI (Shells, Tasks, ...), vous devez définir manuellement le nom de domaine que Email doit utiliser. Il sera utilisé comme nom d'hôte pour l'id du message (puisque il n'y a pas de nom d'hôte dans un environnement CLI) :

```
$email->domain('www.example.org');
// Resulte en ids de message comme ``<UUID@www.example.org>`` (valid)
// au lieu de `<UUID@>`` (invalid)
```

Un id de message valide peut permettre à ce message de ne pas finir dans un dossier de spam.

Obsolète depuis la version 3.4.0 : Utilisez `setDomain()` à la place de `domain()`.

Créer des emails réutilisables

Nouveau dans la version 3.1.0.

Les `Mailers` vous permettent de créer des emails réutilisables pour votre application. Ils peuvent aussi servir à contenir plusieurs configurations d'emails en un seul et même endroit. Cela vous permet de garder votre code DRY ainsi que la configuration d'emails en dehors des autres parties constituant votre application.

Dans cet exemple, vous allez créer un `Mailer` qui contient des emails liés aux 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)
    {
        $this
            ->to($user->email)
            ->subject(sprintf('Welcome %s', $user->name))
            ->template('welcome_mail', 'custom'); // Par défaut le template avec le␣
↪même nom que le nom de la méthode est utilisé.
    }

    public function resetPassword($user)
    {
        $this
            ->to($user->email)
            ->subject('Reset password')
            ->set(['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 prend une `Entity User` et utilise ses propriétés pour configurer chacun des emails.

Vous pouvez maintenant utiliser votre `UserMailer` pour envoyer tous les emails liés aux utilisateurs depuis n'importe où dans l'application. Par exemple, si vous souhaitez 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 = $this->Users->newEntity();
        if ($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 complétement séparer l'envoi de l'email de bienvenue du code de l'application, vous pouvez utiliser votre `UserMailer` via l'évènement `Model.afterSave`. En utilisant un évènement, vous pouvez complètement séparer la logique d'envoi d'emails du reste de votre logique « utilisateurs ». Vous pourriez par exemple ajouter ce qui suit à votre `UserMailer` :

```
public function implementedEvents()
{
    return [
        'Model.afterSave' => 'onRegistration'
    ];
}

public function onRegistration(Event $event, EntityInterface $entity, ArrayObject
↪$options)
{
    if ($entity->isNew()) {
        $this->send('welcome', [$entity]);
    }
}
```

L'objet mailer serait ainsi enregistré en tant qu'écouteur (listeners) d'événement et la méthode `onRegistration()` serait appellée à chaque fois que l'événement `Model.afterSave` serait déclenché. Plus d'information sur comment enregistrer des objets écouteurs d'événements sont disponibles dans la documentation *Enregistrer les Listeners*.

CHAPITRE 19

Gestion des Erreurs & Exceptions

Un grand nombre de méthodes internes à PHP utilisent les erreurs pour communiquer les échecs. Ces erreurs doivent être récupérées et traitées. CakePHP fournit un récupérateur d'erreurs qui les affiche et/ou les écrit dans des fichiers de log par défaut lorsqu'elles se produisent. Ce gestionnaire d'erreurs est utilisé pour capturer les exceptions non interceptées par les controllers et par les autres parties de votre application.

Configuration des Erreurs et des Exceptions

La configuration des Erreurs est faite à l'intérieur du fichier **config/app.php** de votre application. Par défaut CakePHP utilise la classe `ErrorHandler` ou `ConsoleErrorHandler` pour capturer et afficher/mettre les erreurs dans des fichiers de log. Vous pouvez remplacer ce comportement en changeant le gestionnaire d'erreurs par défaut. Le gestionnaire d'erreurs par défaut gère également les exceptions non interceptées.

La gestion des erreurs accepte quelques options qui vous permettent de personnaliser la gestion des erreurs pour votre application :

- `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.
- `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 la journalisation (logging) ainsi que la gestion d'erreur.

ErrorHandler affiche par défaut les erreurs quand `debug` est `true` et les erreurs de logs quand `debug` est `false`. Le type d'erreurs capté dans les deux cas est contrôlé par `errorLevel`. 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.

Créer vos Propres Gestionnaires d'Erreurs

Vous pouvez créer un gestionnaire d'erreurs à partir de n'importe quel type de callback. Par exemple, vous pouvez utiliser une classe appelée `AppError` pour gérer vos erreurs. En étendant `BaseErrorHandler`, vous pouvez fournir une logique de gestion des erreurs personnalisée. Un exemple serait :

```
// Dans config/bootstrap.php
use App\Error\AppError;

$errorHandler = new AppError();
$errorHandler->register();

// Dans src/Error/AppError.php
namespace App\Error;

use Cake\Error\BaseErrorHandler;

class AppError extends BaseErrorHandler
{
    public function _displayError($error, $debug)
    {
        echo 'Il y a eu une erreur!';
    }
    public function _displayException($exception)
    {
        echo 'Il y a eu un exception';
    }
}
```

`BaseErrorHandler` définit deux méthodes abstraites. `_displayError()` est utilisée lorsque les erreurs sont déclenchées. La méthode `_displayException()` est appelée lorsqu'il y a une exception non interceptée.

Changer le Comportement des Erreurs Fatales

Le gestionnaire d'erreurs par défaut convertit les erreurs fatales en exceptions et réutilise la logique de traitement d'exception pour afficher une page d'erreur. Si vous ne voulez pas montrer la page d'erreur standard, vous pouvez la surcharger comme ceci :

```
// Dans config/bootstrap.php
use App\Error\AppError;

$errorHandler = new AppError();
$errorHandler->register();
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Dans src/Error/AppError.php
namespace App\Error;

use Cake\Error\BaseErrorHandler;

class AppError extends BaseErrorHandler
{
    // Autre méthode.

    public function handleFatalError($code, $description, $file, $line)
    {
        echo 'Une erreur fatale est survenue';
    }
}
```

Classes des Exceptions

Il y a de nombreuses classes d'exception dans CakePHP. Le gestionnaire d'exception intégré va capturer les exceptions levées et rendre une page utile. Les exceptions qui n'utilisent pas spécialement un code dans la plage 400 seront traitées comme une erreur interne au serveur.

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.

Nouveau dans la version 3.1 : InvalidCsrfTokenException a été ajoutée.

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.

Nouveau dans la version 3.1.7 : NotAcceptableException a été ajoutée.

exception Cake\Http\Exception**ConflictException**
Utilisée pour faire une erreur 409 Conflict.

Nouveau dans la version 3.1.7 : ConflictException a été ajoutée.

exception Cake\Http\Exception**GoneException**
Utilisée pour faire une erreur 410 Gone.

Nouveau dans la version 3.1.7 : GoneException a été ajoutée.

Pour plus de détails sur les codes de statut d'erreur HTTP 4xx, regardez **RFC 2616#section-10.4** [142].

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.

Nouveau dans la version 3.1.7 : Service Unavailable a été ajoutée.

Pour plus de détails sur les codes de statut d'erreur HTTP 5xx, regardez **RFC 2616#section-10.5** [143].

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 :

```
// Prior to 3.6 use Cake\Network\Exception\NotFoundException
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->set('_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.

De plus, les exceptions de couche du framework suivantes sont disponibles, et seront lancées à partir de certains components du cœur de CakePHP :

Autres Exceptions Intégrées

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.

142. https://tools.ietf.org/html/rfc2616.html#section-10.4
143. https://tools.ietf.org/html/rfc2616.html#section-10.5

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()`*

Nouveau dans la version 3.4.1 : PersistenceFailedException a été ajoutée.

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". Toutes les Exceptions standards que CakePHP va lancer étendent également Exception.

`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."
```

Utiliser les Exceptions HTTP dans vos Controllers

Vous pouvez envoyer n'importe quelle exception HTTP liée à partir des actions de votre controller pour indiquer les états d'échec. Par exemple :

```
// Prior to 3.6 use Cake\Http\Exception\NotFoundException
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->set('_serialize', ['article']);
}
```

Ce qui précède va faire que le gestionnaire d'exception attrape et traite la `NotFoundException`. Par défaut, cela va créer une page d'erreur et enregistrer l'exception.

Exception Renderer

class `Cake\Core\Exception\`**`ExceptionRenderer`**(*Exception $exception*)

La classe ExceptionRenderer avec l'aide de `ErrorController` s'occupe du rendu des pages d'erreur pour toutes les exceptions lancées par votre application.

Les vues de la page d'erreur sont localisées dans **src/Template/Error/**. Pour toutes les erreurs 4xx et 5xx, les fichiers de template **error400.ctp** et **error500.ctp** sont utilisées respectivement. Vous pouvez les personnaliser selon vos besoins. Par défaut, votre **src/Template/Layout/error.ctp** est également utilisé pour les pages d'erreur. Si par exemple, vous voulez utiliser un autre layout **src/Template/Layout/my_error.ctp** pour vos pages d'erreur, modifiez simplement les vues d'erreur et ajoutez la ligne `$this->layout = 'my_error';` dans **error400.ctp** et **error500.ctp**.

Chaque exception au niveau du framework a son propre fichier de vue localisé dans les templates du cœur mais vous n'avez pas besoin de les personnaliser puisqu'ils sont utilisés seulement pendant le développement. Avec debug éteint, toutes les exceptions au niveau du framework sont converties en `InternalErrorException`.

Créer vos Propres Exceptions dans votre Application

Vous pouvez créer vos propres exceptions d'application en utilisant toute exception SPL [144] intégrée, `Exception` lui-même ou :`Cake\Core\Exception\Exception`.

Si votre application contenait l'exception suivante :

```
use Cake\Core\Exception\Exception;

class MissingWidgetException extends Exception
{};
```

Vous pourriez fournir de jolies erreurs de développement, en créant **src/Template/Error/missing_widget.ctp**. Quand on est en mode production, l'erreur du dessus serait traitée comme une erreur 500. Le constructeur pour `Cake\Core\Exception\Exception` a été étendu, vous autorisant à lui passer des données hashées. Ces hashs sont interpolés dans le messageTemplate, ainsi que dans la vue qui est utilisée pour représenter l'erreur dans le mode développement. Cela vous permet de créer des exceptions riches en données, en fournissant plus de contexte pour vos erreurs. Vous pouvez aussi fournir un template de message qui permet aux méthodes natives `__toString()` de fonctionner normalement :

```
use Cake\Core\Exception\Exception;

class MissingWidgetException extends Exception
{
    protected $_messageTemplate = 'Il semblerait que %s soit manquant.';
}

throw new MissingWidgetException(['widget' => 'Pointy']);
```

Lorsque le gestionnaire d'exception intégré attrapera l'exception, vous obtiendriez une variable `$widget` dans votre template de vue d'erreur. De plus, si vous attrapez l'exception en chaîne ou utilisez sa méthode `getMessage()`, vous aurez `Il semblerait que Pointy soit manquant.`. Cela vous permet de créer rapidement vos propres erreurs de développement riches, exactement comme CakePHP le fait en interne.

Créer des Codes de Statut Personnalisés

Vous pouvez créer des codes de statut HTTP personnalisés en changeant le code utilisé quand vous créez une exception :

```
throw new MissingWidgetHelperException('Widget manquant', 501);
```

Va créer un code de réponse 501, vous pouvez utiliser le code de statut HTTP que vous souhaitez. En développement, si votre exception n'a pas de template spécifique, et que vous utilisez un code supérieur ou égal à `500`, vous verrez le template **error500.ctp**. Pour tout autre code d'erreur, vous aurez le template **error400.ctp**. Si vous avez défini un template d'erreur pour votre exception personnalisée, ce template sera utilisé en mode développement. Si vous souhaitez votre propre logique de gestionnaire d'exception même en production, regardez la section suivante.

144. http://php.net/manual/fr/spl.exceptions.php

Etendre et Implémenter vos Propres Gestionnaires d'Exceptions

Vous pouvez implémenter un gestionnaire d'exception spécifique pour votre application de plusieurs façons. Chaque approche vous donne différents niveaux de contrôle sur le processus de gestion d'exception.

— Créer et enregistrer votre propre gestionnaire d'erreurs.
— Etendre le `BaseErrorHandler` fourni par CakePHP.
— Configurer l'option `exceptionRenderer` dans le gestionnaire d'erreurs par défaut.

Dans les prochaines sections, nous allons détailler les différentes approches et les bénéfices de chacune.

Créer votre Propre Gestionnaire d'Exceptions

Créer votre propre gestionnaire d'exception vous donne le contrôle total sur le processus de gestion des exceptions. Dans ce cas, vous devrez vous-même appeler `set_exception_handler`.

Etendre le BaseErrorHandler

La section *Configurer les erreurs* comporte un exemple.

Utiliser l'Option exceptionRenderer dans le Gestionnaire par Défaut

Si vous ne voulez pas prendre le contrôle sur le gestionnaire d'exception, mais que vous voulez changer la manière dont les exceptions sont rendues, vous pouvez utiliser l'option `exceptionRenderer` dans **config`/app.php** pour choisir la classe qui affichera les pages d'exception. Par défaut *`Cake\Core\Exception\ExceptionRenderer`* est utilisée. Votre gestionnaire d'exceptions doit être placé dans **src/Error**. Dans une classe de rendu personnalisé d'exception vous pouvez fournir un traitement particulier pour les erreurs spécifique à votre application :

```
// Dans src/Error/AppExceptionRenderer.php
namespace App\Error;

use Cake\Error\ExceptionRenderer;

class AppExceptionRenderer extends ExceptionRenderer
{
    public function missingWidget($error)
    {
        return 'Oups ce widget est manquant!';
    }
}

// Dans config/app.php
'Error' => [
    'exceptionRenderer' => 'App\Error\AppExceptionRenderer',
    // ...
],
// ...
```

Le code ci-dessus gérerait toutes les exceptions de type `MissingWidgetException`, et vous permettrait un affichage et/ou une logique de gestion personnalisée pour ces exceptions de l'application. Les méthodes de gestion d'exceptions obtiennent l'exception étant traitée en argument. Votre gestionnaire de rendu personnalisé peut retourner une chaîne ou un objet `Response`. Retourner une `Response` vous donnera le contrôle total de la réponse.

Note : Votre gestionnaire de rendu doit attendre une exception dans son constructeur et implémenter une méthode de rendu. Ne pas le faire entraînera des erreurs supplémentaires.

Si vous utilisez un gestionnaire d'exception personnalisé, configurer le moteur de rendu n'aura aucun effet. A moins que vous le référenciez à l'intérieur de votre implémentation.

Créer un Controller Personnalisé pour Gérer les Exceptions

Par convention CakePHP utilisera `App\Controller\ErrorController` s'il existe. Implémenter cette classe vous offrira une voie pour personnaliser les pages d'erreur sans aucune configuration supplémentaire.

Si vous utilisez un moteur de rendu d'exceptions personnalisé, vous pouvez utiliser la méthode `_getController()` pour rendre un controller personnalisé. En implémentant `_getController()` dans votre moteur de rendu d'exceptions, vous pouvez utiliser n'importe quel controller de votre choix :

```
// Dans src/Error/AppExceptionRenderer
namespace App\Error;

use App\Controller\SuperCustomErrorController;
use Cake\Error\ExceptionRenderer;

class AppExceptionRenderer extends ExceptionRenderer
{
    protected function _getController($exception)
    {
        return new SuperCustomErrorController();
    }
}

// Dans config/app.php
'Error' => [
    'exceptionRenderer' => 'App\Error\AppExceptionRenderer',
    // ...
],
// ...
```

Le controller d'erreur, qu'il soit conventionnel ou personnalisé, est utilisé pour rendre la vue de page d'erreurs et reçoit tous les événements standards du cycle de vie des requêtes.

Logger les Exceptions

En Utilisant la gestion d'exception intégrée, vous pouvez logger toutes les exceptions qui sont gérées par ErrorHandler en configurant l'option `log` à `true` dans votre **config/app.php**. Activer cela va logger chaque exception vers *`Cake\Log\Log`* et les loggers configurés.

Note : Si vous utilisez un gestionnaire personnalisé, cette configuration n'aura aucun effet. A moins que vous ne le référenciez à l'intérieur de votre implémentation.

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
            ]);
            $this->eventManager()->dispatch($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)`

Nouveau dans la version 3.2.11 : Le tracking d'événements et `Cake\Event\EventList` ont été ajoutés.

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 [
            '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;

$this->Orders->eventManager()->on('Model.Orders.afterPlace', function ($event) {
    Log::write(
        'info',
        'A new order was placed with id: ' . $event->getSubject()->id
    );
});
```

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 = $this->eventManager()->matchingListeners('Verification');
if (!empty($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.

Nouveau dans la version 3.2.3 : La méthode `matchingListeners` retourne un tableau d'événements qui matchent un patron de recherche.

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()
    {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        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 `Event\EventManager::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 `Event\EventManager::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)
{
    // ...
    $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.
```

(suite sur la page suivante)

(suite de la page précédente)

```
$myFunction = function ($event) { ... };
$this->eventManager()->on('My.event', $myFunction);

// Détacher la fonction anonyme
$this->eventManager()->off('My.event', $myFunction);

// Attacher un EventListener
$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 [145]. 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 [146] 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, par exemple `fr_FR`, `es_AR`, `da_DK`, qui contient en même temps la langue et le pays où elle est parlée.

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"
```

145. http://en.wikipedia.org/wiki/Gettext
146. https://github.com/cakephp/localized

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 [147] 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 :

147. http://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 [148] pour obtenir une vue d'ensemble complète des alias que vous pouvez utiliser pour chaque langue.

148. http://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 [149] 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)

149. 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

// Avant 3.5, utilisez translator()
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.
});
```

Nouveau dans la version 3.4.0 : Replacing the `_fallback` loader was added in 3.4.0

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');

$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 *Filtres du Dispatcher*, vous pouvez configurer les types Date, Time, et DateTime pour parser les formats localisés :

```
use Cake\Database\Type;

// Permet de parser avec le format de locale par défaut.
Type::build('datetime')->useLocaleParser();

// Configure un parser personnalisé du format de datetime.
Type::build('datetime')->useLocaleParser()->setLocaleFormat('dd-M-y');

// Vous pouvez aussi utiliser les constantes IntlDateFormatter.
Type::build('datetime')->useLocaleParser()
    ->setLocaleFormat([IntlDateFormatter::SHORT, -1]);
```

Le parsing du format par défaut est le même que le format de chaîne par défaut.

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.

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 est facile - la fonction log() est fourni 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\Log;

// Nom de classe court
Log::config('debug', [
    'className' => 'File',
    'path' => LOGS,
    'levels' => ['notice', 'info', 'debug'],
    'file' => 'debug',
]);

// Nom avec le namespace complet.
```

(suite sur la page suivante)

(suite de la page précédente)

```
Log::config('error', [
    'className' => 'Cake\Log\Engine\FileLog',
    'path' => LOGS,
    'levels' => ['warning', 'error', 'critical', 'alert', 'emergency'],
    'file' => 'error',
]);
```

Ce qui est au-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 facile 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::config()*.

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::config('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::config('error', [
    'url' => 'file:///?levels[]=warning&levels[]=error&file=error',
]);
```

Note : Les loggers sont nécessaires pour intégrer l'interface `Psr\Log\LoggerInterface`.

Créer des Adaptateurs de Log

Les gestionnaires de flux de log peuvent faire partie de votre application, ou parti d'un plugin. Si par exemple vous avez un enregistreur de logs de base de données appelé `DatabaseLog`. Comme faisant partie de votre application il devrait être placé dans **src/Log/Engine/DatabaseLog.php**. Comme faisant partie d'un plugin il devrait être placé dans **plugins/LoggingPack/src/Log/Engine/DatabaseLog.php**. Pour configurer des flux de logs, vous devez utiliser *Cake\Log\Log::config()*. Par example, la configuration de notre `DatabaseLog` pourrait ressembler à ceci :

```
// Pour src/Log
Log::config('otherFile', [
    'className' => 'Database',
    'model' => 'LogEntry',
    // ...
]);

// Pour un plugin appelé LoggingPack
Log::config('otherFile', [
    'className' => 'LoggingPack.Database',
    'model' => 'LogEntry',
    // ...
]);
```

Lorsque vous configurez le flux d'un 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 des flux de log comme un 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 adaptateurs de logging intègrent `Psr\Log\LoggerInterface`. La classe `CakeLogEngineBaseLog` est un moyen facile de satisfaire l'interface puisqu'elle nécessite seulement que vous intégriez la méthode `log()`.

Le moteur de `FileLog` a quelques nouvelles configurations :

- `size` Utilisé pour implémenter la rotation de fichier de journal basic. Si la taille d'un fichier de log atteint la taille spécifiée, le fichier existant est renommé en ajoutant le timestamp au nom du fichier et un nouveau fichier de log est créé. Peut être une valeur de bytes en entier ou des valeurs de chaînes lisible par l'humain comme "10MB", "100KB" etc. Par défaut à 10MB.
- `rotate` Les fichiers de log font une rotation à un temps spécifié avant d'être retiré. Si la valeur est 0, les versions anciennes seront retirées plutôt que mises en rotation. Par défaut à 10.
- `mask` Définit les permissions du fichier pour les fichiers créés. Si laissé vide, les permissions par défaut sont utilisées.

Avertissement : Les moteurs ont le suffixe `Log`. Vous devrez éviter les noms de classe comme `SomeLogLog` qui inclut le suffixe deux fois à la fin.

Note : Vous devrez configurer les loggers pendant le bootstrapping. **config/app.php** est l'endroit par convention pour configurer les adaptateurs de log.

En mode debug, les répertoires manquants vont maintenant être automatiquement créés pour éviter le lancement des erreurs non nécessaires lors de l'utilisation de FileEngine.

Journalisation des Erreurs et des Exception

Les erreurs et les exception peuvent elles aussi être journalisées. En configurant les valeurs correspondantes dans votre fichier app.php. Les erreurs seront affichées quand debug est à `true` et loguées quand debug est à `false`. Définir l'option `log` à `true` pour logger les exceptions non capturées. Voir *Configuration* pour plus d'information.

Interagir avec les Flux de Log

Vous pouvez interroger le flux configurés avec `Cake\Log\Log::configured()`. Le retour de `configured()` est un tableau de tous les flux actuellement configurés. Vous pouvez rejeter des flux en utilisant `Cake\Log\Log::drop()`. Une fois que le flux d'un log à été rejeté il ne recevra plus de messages.

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` :

```
// Execute cela dans une classe CakePHP
$this->log("Quelque chose ne fonctionne pas!");

// Aboutit à ce que cela soit ajouté à tmp/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/'
]);
```

Avertissement : Si vous ne configurez pas d'adaptateur de logging, les logs ne seront pas stockés.

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 que ceux écrits et sera fait (presque) d'une manière non-blocking 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 *bootstrap.php* :

```
Log::config('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 sur le serveur ou la procédure dans le message de log. Par exemple : `%s - Web Server 1 - %s` va ressembler à `error - Web Server 1 - An error occurred in this request` après avoir remplacé les placeholders.

— *prefix* : Une chaine qui va être préfixée à tous les messages de log.
— *flag* : Un drapeau entier utilisé pour l'ouverture de la connexion à logger, par défaut *LOG_ODELAY* sera utilisée. 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.

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 qui 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 CakePHP:
$this->log("Quelque chose qui ne fonctionne pas!", 'debug');
```

Tous les flux de log configurés sont écrits séquentiellement à chaque fois que *Cake\Log\Log::write()* est appelée. Vous n'avez pas besoin de configurer un flux pour utiliser la journalisation. Si vous n'avez pas configuré d'adaptateurs de log, `log()` va retourner false et aucun message de log ne sera écrit.

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 :

```
// Configurez logs/shops.log pour recevoir tous les types (niveaux de log),
// mais seulement ceux avec les scope `orders` et `payments`
Log::config('shops', [
    'className' => 'File',
    'path' => LOGS,
    'levels' => [],
    'scopes' => ['orders', 'payments'],
    'file' => 'shops.log',
]);

// configurez logs/payments.log pour recevoir tous les types, mais seulement
// ceux qui ont un scope `payments`
Log::config('payments', [
    'className' => 'File',
    'path' => LOGS,
    'levels' => [],
    'scopes' => ['payments'],
    'file' => 'payments.log',
]);

Log::warning('this gets written only to shops.log', ['scope' => ['orders']]);
Log::warning('this gets written to both shops.log and payments.log', ['scope' => [
↪'payments']]);
Log::warning('this gets written to both shops.log and payments.log', ['scope' => [
↪'unknown']]);
```

Les scopes peuvent aussi être passées en une chaine unique ou un tableau numériquement indexé. Notez que l'utilisation de ce formulaire va limiter la capacité de passer plus de données en contexte :

```
Log::warning('This is a warning', ['orders']);
Log::warning('This is a warning', 'payments');
```

Note : Quand l'option `scopes` est 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.

l'API de Log

class Cake\Log**Log**

Une simple classe pour écrire dans les logs (journaux).

static Cake\Log\Log::**config**(*$key*, *$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**

Appelle 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::**debug**(*$message*, *$scope = []*)

static Cake\Log\Log::**info**(*$message*, *$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. Si `$msg` n'est pas une chaîne, elle sera convertie avec `print_r` avant d'être écrite.

Utiliser Monolog

Monolog est un logger populaire pour PHP. Puisqu'il intègre les mêmes interfaces que les loggers de CakePHP, il est facile de l'utiliser dans votre application comme logger par défaut.

Après avoir installé Monolog en utilisant composer, configurez le logger en utilisant la méthode `Log::config()` :

```
// config/bootstrap.php

use Monolog\Logger;
use Monolog\Handler\StreamHandler;

Log::config('default', function () {
    $log = new Logger('app');
    $log->pushHandler(new StreamHandler('path/to/your/combined.log'));
    return $log;
});

// 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::config('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);
    }
}
```

Dans l'exemple ci-dessus, nous utilisons la méthode `execute()` pour lancer la méthode `_execute()` de notre formulaire seulement lorsque les données sont valides, et définissons un message flash en conséquence. Nous aurions aussi pu utiliser la méthode `validate()` pour valider uniquement les données de requête :

```
$isValid = $form->validate($this->request->getData());
```

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);
    }
}
```

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']
```

(suite sur la page suivante)

(suite de la page précédente)

```
]
*/
```

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');
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.

Dans CakePHP 3.0 chaque plugin définit 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 [150] 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.

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`. En supposant que vous voulez installer un plugin appelé `ContactManager`,

150. http://packagist.org

vous auriez un dossier dans `plugins` appelé `ContactManager`. Dans ce répertoire se trouvent les View, Model, Controller, webroot, et tous les autres répertoires du plugin.

Plugin Map File

Lorsque vous installez des plugins via Composer, vous pouvez voir que **vendor/cakephp-plugins.php** est créé. Ce fichier de configuration contient une carte des noms de plugin et leur chemin dans le système de fichiers. Cela ouvre la possibilité pour un plugin d'être installé dans le dossier vendor standard qui est à l'extérieur des dossiers de recherche standards. La classe `Plugin` utilisera ce fichier pour localiser les plugins lorsqu'ils sont chargés avec `load()` ou `loadAll()`. Généralement vous n'aurez pas à éditer ce fichier à la main car Composer et le package `plugin-installer` le feront pour vous.

Charger un Plugin

Après avoir installé un plugin et mis à jour l'autoloader, vous devrez charger le plugin. Vous pouvez charger les plugins un par un, ou tous d'un coup avec une méthode unique :

```
// Dans config/bootstrap.php
// Ou dans Application::bootstrap()

// Charge un Plugin unique
Plugin::load('ContactManager');

// Charge un plugin unique, avec un namespace personnalisé.
Plugin::load('AcmeCorp/ContactManager');

// Charge tous les plugins d'un coup
Plugin::loadAll();
```

`loadAll()` charge tous les plugins disponibles, vous permettant de définir certaines configurations pour des plugins spécifiques. `load()` fonctionne de la même manière, mais charge seulement les plugins que vous avez spécifié explicitement.

Note : `Plugin::loadAll()` ne va pas charger les plugins se trouvant dans vendor qui ne sont pas définis dans **vendor/cakephp-plugins.php**.

Il existe aussi une commande shell très pratique qui va activer le plugin. Exécutez la ligne suivante :

```
bin/cake plugin load ContactManager
```

Ceci va vous ajouter le bout de code `Plugin::load('ContactManager');` dans le fichier bootstrap.

Autochargement des Classes du Plugin

Quand on utilise `bake` pour la création d'un plugin ou quand on en installe un en utilisant Composer, vous n'avez typiquement pas besoin de faire des changements dans votre application afin que CakePHP reconnaisse les classes qui se trouvent dedans.

Dans tous les autres cas, vous avez peut-être besoin de modifier le fichier composer.json de votre application pour contenir les informations suivantes :

```
"psr-4": {
    (...)
    "MyPlugin\\": "./plugins/MyPlugin/src",
    "MyPlugin\\Test\\": "./plugins/MyPlugin/tests"
}
```

Si vous utilisez des namespaces pour vos plugins, le mapping des namespaces vers les dossiers doit ressembler à ceci :

```
"psr-4": {
    (...)
    "AcmeCorp\\Users\\": "./plugins/AcmeCorp/Users/src",
    "AcmeCorp\\Users\\Test\\": "./plugins/AcmeCorp/Users/tests"
}
```

De plus, vous aurez besoin de dire à Composer de refraichir le cache de l'autochargement :

```
$ php composer.phar dumpautoload
```

Si vous ne pouvez pas utiliser Composer pour toute raison, vous pouvez aussi utiliser un autochargement fallback pour votre plugin :

```
Plugin::load('ContactManager', ['autoload' => true]);
```

Configuration du Plugin

Les méthodes `load` et `loadAll` peuvent vous aider pour la configuration et le routing d'un plugin. Peut-être souhaiterez vous charger tous les plugins automatiquement, en spécifiant des routes et des fichiers de bootstrap pour certains plugins :

```
// dans config/bootstrap.php
// Ou dans Application::bootstrap()

// En utilisant loadAll()
Plugin::loadAll([
    'Blog' => ['routes' => true],
    'ContactManager' => ['bootstrap' => true],
    'WebmasterTools' => ['bootstrap' => true, 'routes' => true],
]);
```

Ou vous pouvez charger les plugins individuellement :

```
// Charge seulement le blog et inclut les routes
Plugin::load('Blog', ['routes' => true]);

// Inclut le fichier de démarrage pour la configuration/initialisation.
Plugin::load('ContactManager', ['bootstrap' => true]);
```

Avec ces deux approches, vous n'avez plus à faire manuellement un `include()` ou un `require()` du fichier de configuration ou du fichier de routes du plugin – cela arrive automatiquement au bon moment et au bon endroit.

Vous pouvez spécifier un ensemble de valeurs par défaut pour `loadAll()` qui vont s'appliquer à chaque plugin qui n'a pas de configuration spécifique.

L'exemple suivant va charger le fichier de bootstrap de tous les plugins, et aussi les routes du plugin Blog :

```
Plugin::loadAll([
    ['bootstrap' => true],
    'Blog' => ['routes' => true]
]);
```

Notez que tous les fichiers spécifiés doivent réellement exister dans le(s) plugin(s) configurés ou PHP vous donnera des avertissements pour chaque fichier qu'il ne peut pas charger. Vous pouvez éviter les avertissements potentiels en utilisant l'option `ignoreMissing` :

```
Plugin::loadAll([
    ['ignoreMissing' => true, 'bootstrap' => true],
    'Blog' => ['routes' => true]
]);
```

Par défaut le namespace du Plugin doit correspondre au nom du plugin. Par exemple si vous avez un plugin avec un namespace de haut niveau `Users`, vous le chargeriez en utilisant :

```
Plugin::load('User');
```

Si vous préférez avoir votre nom de vendor en haut niveau et avoir un namespace comme `AcmeCorp/Users`, alors vous devrez charger le plugin comme suit :

```
Plugin::load('AcmeCorp/Users');
```

Cela va assurer que les noms de classe sont résolus correctement lors de l'utilisation de la *syntaxe de plugin*.

La plupart des plugins vont indiquer la procédure correcte pour les configurer et configurer la base de données dans leur documentation. Certains plugins nécessitent plus de configurations que les autres.

Utiliser un Plugin

Vous pouvez référencer les controllers, models, components, behaviors et helpers du plugin en préfixant le nom du plugin avant le nom de classe.

Par exemple, disons que vous voulez utiliser le ContactInfoHelper du plugin ContactManager pour sortir de bonnes informations de contact dans une de vos vues. Dans votre controller, le tableau `$helpers` pourrait ressembler à ceci :

```
public $helpers = ['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);
```

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
            /Controller
                /Component
            /Model
                /Table
                /Entity
                /Behavior
            /View
                /Helper
            /Template
                /Layout
        /tests
            /TestCase
            /Fixture
        /webroot
```

Notez que 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 certains cas, ils peuvent carrément ne pas avoir de répertoire "Template".

Un plugin peut aussi avoir tous les autres répertoires que votre application a, comme Config, Console, Lib, 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 un plugin, utilisez la commande suivante :

```
bin/cake bake plugin ContactManager
```

Maintenant vous pouvez cuisiner en utilisant les mêmes conventions qui s'appliquent au reste de votre app. Par exemple - baking controllers :

```
bin/cake bake controller --plugin ContactManager Contacts
```

Référez-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 une fois que vous avez créé votre plugin :

```
php composer.phar dumpautoload
```

Routes de Plugins

Les plugins peuvent contenir des fichiers de routes contenant leurs propres routes. Chaque plugin contient 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;
use Cake\Routing\Router;

Router::plugin(
    'ContactManager',
    ['path' => '/contact-manager'],
    function ($routes) {
        $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 pouvez personnaliser ce fichier avec plus de routes plus tard.

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 **config/bootstrap.php**, ajoutez la ligne suivante :

```
Plugin::load('ContactManager', ['routes' => true]);
```

Vous pouvez également charger les routes du plugin dans la liste des routes de votre application. Le faire de cette manière vous permet d'avoir plus de contrôle sur la manière dont les routes de plugin sont chargées et vous permet d'englober les routes du plugin dans des préfixes et des "scopes" spécifiques :

```
Router::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`.

Nouveau dans la version 3.5.0 : `RouteBuilder::loadPlugin()` a été ajoutée dans 3.5.0

Controllers du Plugin

Les controllers pour notre plugin ContactManager seront stockés dans **plugins/ContactManager/src/Controller/**. Puisque la principale chose que nous souhaitons faire est la gestion des contacts, nous aurons besoin de créer 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 qu'on a fait avant, visitez `/contact-manager/contacts`. Vous aurez une erreur « Missing Model » parce que nous n'avons pas de model Contact encore défini.

Si votre application inclut le routage par défaut que CakePHP fournit, vous serez capable d'accéder aux controllers de votre plugin en utilisant les 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 va aussi connecter 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, ceci peut être fait comme expliqué dans la documentation *Autochargement des Classes du 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, séparé 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)
    {
        $this->hasMany('ContactManager.AltName');
    }
}
```

Si vous préférez que les clés du tableau pour l'association n'aient pas le préfix 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
{
    public function initialize(array $config)
    {
        $this->hasMany('AltName', [
            'className' => 'ContactManager.AltName',
        ]);
    }
}
```

Vous pouvez utiliser `TableRegistry` pour charger les tables de votre plugin en utilisant l'habituelle *syntaxe de*

plugin :

```
use Cake\ORM\TableRegistry;

// Prior to 3.6 use TableRegistry::get('ContactManager.Contacts')
$contacts = TableRegistry::getTableLocator()->get('ContactManager.Contacts');
```

Si vous êtes dans un Controller, vous pouvez aussi utiliser :

```
$this->loadModel('ContactsMangager.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]/Template/`. Pour notre plugin ContactManager, nous aurons besoin d'une vue pour notre action `ContactsController::index()`, ainsi incluons ceci aussi :

```
// plugins/ContactManager/src/Template/Contacts/index.ctp:
<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. Ajoutez des layouts de plugin, dans `plugins/[PluginName]/src/Template/Layout`. Pour utiliser le layout d'un plugin dans votre controller, vous pouvez faire ce qui suit :

```
public $layout = 'ContactManager.admin';
```

Si le préfix de plugin n'est pas mis, le fichier de vue/layout sera localisé normalement.

Note : Pour des informations sur la façon d'utiliser les elements à partir d'un plugin, regardez *Elements*.

Redéfinition des Template 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 de l'application plus spécifique, en créant des fichiers en utilisant le template suivant **src/Template/Plugin/[Plugin]/[Controller]/[view].ctp**. Pour le controller Contacts, vous pouvez faire le fichier suivant :

```
src/Template/Plugin/ContactManager/Contacts/index.ctp
```

Créer ce fichier vous permettra de redéfinir **plugins/ContactManager/src/Template/Contacts/index.ctp**.

Si votre plugin est dans une dépendence de Composer (ex : "LeVendor/LePlugin), le chemin vers la vue "index" du controlleur Custom sera

src/Template/Plugin/LeVendor/LePlugin/Custom/index.ctp

Créer ce fichier vous permettra de redéfinir **vendor/levendor/leplugin/src/Template/Custom/index.ctp**.

Si le plugin implémente un préfixe de routing, vous devez inclure ce préfixe dans la surcharge de template de votre application.

Si le plugin "ContactManager" implémente un préfixe "admin", le chemin de la redéfinition sera :

```
src/Template/Plugin/ContactManager/Admin/ContactManager/index.ctp
```

Assets de Plugin

Les assets web du plugin (mais pas les fichiers de PHP) peuvent être servis à travers le répertoire `webroot` du plugin, juste comme les assets de l'application principale :

```
/plugins/ContactManager/webroot/
                                 css/
                                 js/
                                 img/
                                 flash/
                                 pdf/
```

Vous pouvez mettre tout type de fichier dans tout répertoire, juste comme un webroot habituel.

Avertissement : La gestion des assets static, comme les fichiers images, Javascript et CSS, à travers le Dispatcher est très inéfficace. Regardez *Améliorer les Performances de votre Application* pour plus d'informations.

Lier aux plugins

Vous pouvez utiliser la *syntaxe de plugin* pour lier les assets de plugin en utilisant les méthodes script, image ou css de `View\Helper\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 assets de Plugin sont servis en utilisant le filtre du dispatcheur `AssetFilter` par défaut. C'est seulement recommandé pour le développement. En production vous devrez *symlinker vos assets* pour améliorer la performance.

Si vous n'utilisez pas les helpers, vous pouvez préfixer /plugin_name/ au début de l'URL pour servir un asset du plugin . Lier avec "/contact_manager/js/some_file.js" servirait l'asset **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 tout projet.

Construire ces components est exactement la même chose que de les construire à l'intérieur d'une application habituelle, avec aucune convention spéciale de nommage.

Faire référence avec votre component, depuis l'intérieur ou l'extérieur de votre plugin nécessite seulement que vous préfixiez le nom du plugin avant le nom du component. 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()
{
    parent::initialize();
    $this->loadComponent('ContactManager.Example');
}
```

La même technique s'applique aux Helpers et aux Behaviors.

Etendez votre Plugin

Cet exemple est un bon début pour un plugin, mais il y a beaucoup plus à faire. En règle générale, tout ce que vous pouvez faire avec votre application, vous pouvez le faire à l'intérieur d'un plugin à la place.

Continuez, incluez certaines librairies tierces dans "vendor", ajoutez de nouveaux shells à la console de cake, et n'oubliez pas de créer des cas de test ainsi les utilisateurs de votre plugin peuvent automatiquement tester les fonctionnalités de votre plugin !

Dans notre exemple ContactManager, nous pourrions créer des actions add/remove/edit/delete dans le ContactsController, intégrer la validation dans le model Contact, et intégrer la fonctionnalité à laquelle on pourrait s'attendre quand on gère ses contacts. A vous de décider ce qu'il faut intégrer dans vos plugins. N'oubliez juste pas de partager votre code avec la communauté afin que tout le monde puisse bénéficier de votre component génial et réutilisable !

Publiez votre Plugin

Vous pouvez ajouter votre plugin sur plugins.cakephp.org [151]. De cette façon, il peut être facilement utilisé avec Composer. Vous pouvez aussi proposer votre plugin à la liste awesome-cakephp [152]

Aussi, vous pouvez créer un fichier composer.json et publier votre plugin sur packagist.org [153].

Choisissez un nom de package avec une sémantique qui a du sens. Il devra idéalement être préfixé avec la dépendance, dans ce cas « cakephp » comme le framework. Le nom de vendor sera habituellement votre nom d'utilisateur sous GitHub. **N'utilisez pas** le namespace CakePHP (cakephp) puisqu'il est reservé aux plugins appartenant à CakePHP. La convention est d'utiliser les lettres en minuscule et les tirets en séparateur.

Donc si vous créez un plugin « Logging » avec votre compte GitHub « FooBar », un bon nom serait *foo-bar/cakephp-logging*. Et le plugin « Localized » appartenant à CakePHP peut être trouvé dans *cakephp/localized*.

151. https://plugins.cakephp.org
152. https://github.com/FriendsOfCake/awesome-cakephp
153. https://packagist.org/

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 facilement, 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.

Alors que d'autres solutions existent, REST est un bon moyen de fournir facilement un accès à la logique que vous avez créée dans votre application. C'est simple, habituellement basé sur XML (nous parlons de XML simple, rien de semblable à une enveloppe SOAP), et dépend des headers HTTP pour la direction. Exposer une API via REST dans CakePHP est simple.

Mise en place Simple

Le moyen le plus rapide pour démarrer avec REST est d'ajouter quelques lignes pour configurer *resource routes* dans votre fichier config/routes.php.

Une fois que le router a été configuré pour mapper les requêtes REST vers certaines actions de controller, nous pouvons continuer et créer la logique dans nos actions de controller. Un controller basique pourrait ressembler à ceci :

```
// src/Controller/RecipesController.php
class RecipesController extends AppController
{

    public function initialize()
    {
        parent::initialize();
        $this->loadComponent('RequestHandler');
    }

    public function index()
    {
        $recipes = $this->Recipes->find('all');
```

(suite sur la page suivante)

(suite de la page précédente)

```
        $this->set([
            'recipes' => $recipes,
            '_serialize' => ['recipes']
        ]);
    }

    public function view($id)
    {
        $recipe = $this->Recipes->get($id);
        $this->set([
            'recipe' => $recipe,
            '_serialize' => ['recipe']
        ]);
    }

    public function add()
    {
        $recipe = $this->Recipes->newEntity($this->request->getData());
        if ($this->Recipes->save($recipe)) {
            $message = 'Saved';
        } else {
            $message = 'Error';
        }
        $this->set([
            'message' => $message,
            'recipe' => $recipe,
            '_serialize' => ['message', 'recipe']
        ]);
    }

    public function edit($id)
    {
        $recipe = $this->Recipes->get($id);
        if ($this->request->is(['post', 'put'])) {
            $recipe = $this->Recipes->patchEntity($recipe, $this->request->getData());
            if ($this->Recipes->save($recipe)) {
                $message = 'Saved';
            } else {
                $message = 'Error';
            }
        }
        $this->set([
            'message' => $message,
            '_serialize' => ['message']
        ]);
    }

    public function delete($id)
    {
        $recipe = $this->Recipes->get($id);
        $message = 'Deleted';
        if (!$this->Recipes->delete($recipe)) {
            $message = 'Error';
        }
        $this->set([
            'message' => $message,
            '_serialize' => ['message']
```

(suite sur la page suivante)

(suite de la page précédente)

```
        ]);
    }
}
```

Les controllers RESTful utilisent souvent les extensions parsées pour servir différentes views basées sur différents types de requête. Puisque nous gérons les requêtes REST, nous ferons des views XML. Vous pouvez aussi faire des views JSON en utilisant les *Vues JSON et XML* intégrées à CakePHP. En utilisant `XmlView` intégré, nous pouvons définir une variable de vue `_serialize`. Cette variable de vue spéciale est utilisée pour définir les variables de vue que `XmlView` doit sérialiser en XML.

Si nous voulons modifier les données avant qu'elles soient converties en XML, nous ne devons pas définir la variable de vue `_serialize`, et à la place utiliser les fichiers de template. Nous plaçons les vues REST pour notre RecipesController à l'intérieur de **src/Template/Recipes/xml**. Nous pouvons aussi utiliser `Xml` pour une sortie XML facile et rapide dans ces vues. Voici ce que notre vue index pourrait ressembler à :

```
// src/Template/Recipes/xml/index.ctp
// Faire du formatage et de la manipulation sur le tableau
// $recipes.
$xml = Xml::fromArray(['response' => $recipes]);
echo $xml->asXML();
```

Quand vous servez le type de contenu spécifique en utilisant parseExtensions(), CakePHP recherche automatiquement un helper de view qui matche le type. Puisque nous utilisons le XML en type de contenu, il n'y a pas de helper intégré cependant si vous en créez un, il va être automatiquement chargé pour notre utilisation dans ces vues.

Le XML rendu va finir par ressembler à ceci :

```
<recipes>
    <recipe>
        <id>234</id>
        <created>2008-06-13</created>
        <modified>2008-06-14</modified>
        <author>
            <id>23423</id>
            <first_name>Billy</first_name>
            <last_name>Bob</last_name>
        </author>
        <comment>
            <id>245</id>
            <body>Yummy yummmy</body>
        </comment>
    </recipe>
    ...
</recipes>
```

Créer la logique pour l'action edit est un tout petit peu plus compliqué. Puisque vous fournissez une API qui sort du XML, c'est un choix naturel de recevoir le XML en input. Ne vous inquiétez pas, les classes `Cake\Controller\Component\RequestHandler` et `Cake\Routing\Router` vous facilitent les choses. Si une requête POST ou PUT a un type de contenu XML, alors l'input est lancé à travers la classe `Xml` de CakePHP, et la representation en tableau des données est assigné à *$this->request->data*. Avec cette fonctionnalité, la gestion de XML et les données POST en parallèle est seamless : aucun changement n'est nécessaire pour le code du controller ou du model. Tout ce dont vous avez besoin devrait se trouver dans `$this->request->getData()`.

Accepter l'Input dans d'Autres Formats

Typiquement les applications REST ne sortent pas seulement du contenu dans des formats de données alternatifs, elles acceptent aussi des données dans des formats différents. Dans CakePHP, `RequestHandlerComponent` facilite ceci. Par défaut, elle va décoder toute donnée d'input JSON/XML entrante pour des requêtes POST/PUT et fournir la version du tableau de ces données dans *$this->request->data*. Vous pouvez aussi connecter avec des deserialisers supplémentaires dans des formats alternatifs si vous avez besoin d'eux en utilisant `RequestHandler::addInputType()`

RESTful Routing

Le Router de CakePHP facilite la connexion des routes pour les ressources RESTful. Consultez la section *Créer des Routes RESTful* pour plus d'informations.

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 [154] 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 [155] soit mcrypt [156] selon ce qui est disponible sur votre système. Les données cryptées dans une implémentation sont portables vers les autres implémentations.

Avertissement : L'extension mcrypt [157] a été dépréciée dans PHP7.1

154. https://api.cakephp.org/3.x/class-Cake.Utility.Security.html
155. http://php.net/openssl
156. http://php.net/mcrypt
157. http://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 *Utility\Security::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.

Nouveau dans la version 3.2.3 : The randomBytes method was added in 3.2.3.

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',
    '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
```

(suite sur la page suivante)

(suite de la page précédente)

```
  `data` blob DEFAULT NULL, -- for PostgreSQL use bytea instead of blob
  `expires` int(10) unsigned DEFAULT NULL,
  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
```

(suite sur la page suivante)

(suite de la page précédente)

```
        'session.gc_divisor' => 1000,
        'session.cookie_httponly' => true
    ]
]);
```

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, $this->cacheKey);
        return parent::destroy($id);
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    // 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. Utiliser le gestionnaire de session est aussi facile. Dans votre **config/app.php**, imitez le block de session qui suit :

```
'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->session();
$name = $session->read('User.name');
```

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 *FlashComponent* et *Flash*

CHAPITRE 28

Testing

CakePHP fournit un support de test intégré compréhensible. CakePHP permet l'intégration de PHPUnit [158]. 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 commencer avec le Test Unitaire, et comment vous pouvez utiliser les extensions que CakePHP offre.

Installer PHPUnit

CakePHP utilise PHPUnit comme framework de test sous-jacent. PHPUnit est le standard de-facto pour le test unitaire dans PHP. Il offre un ensemble de fonctionnalités profondes et puissantes pour s'assurer que votre code fait ce que vous pensez qu'il doit faire. PHPUnit peut être installé avec le PHAR package [159] ou avec Composer [160].

Installer PHPUnit avec Composer

Pour installer PHPUnit avec Composer :

```
$ php composer.phar require --dev phpunit/phpunit:"^5.7|^6.0"

// Avant CakePHP 3.4.1
$ php composer.phar require --dev phpunit/phpunit:"<6.0"
```

Ceci va ajouter la dépendance à la section `require-dev` de votre `composer.json`, et ensuite installer PHPUnit avec vos autres dépendances.

Vous pouvez maintenant lancer PHPUnit en utilisant :

158. http://phpunit.de
159. http://phpunit.de/#download
160. http://getcomposer.org

```
$ vendor/bin/phpunit
```

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 [161].

Tester la Configuration de la Base de Données

Souvenez-vous qu'il faut avoir debug activé dans votre fichier **config/app.php** avant de lancer des tests. Vous devrez aussi vous assurer d'ajouter une configuration de base de données `test` dans **config/app.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 toute erreur embarrassante pouvant arriver plus tard.

161. http://phpunit.de/manual/current/en/installation.html#installation.phar.windows

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
```

Ce qui est au-dessus 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 au test en paramètre de 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** plutôt que juste **.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.

Nouveau dans la version 3.4.1 : Le support de PHPUnit 6 a été ajouté. Si vous utilisez une version de PHPUnit inférieure à 5.7.0, vos classes de tests devront soit *extends* les classes de CakePHP, soit `PHPUnit_Framework_TestCase`.

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 sera le formatage d'une barre de progression HTML. Notre helper ressemblera à cela :

```
namespace App\View\Helper;

use Cake\View\Helper;
```

(suite sur la page suivante)

(suite de la page précédente)

```
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 vous pouvez 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 ce qui suit :

```
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()
    {

    }

    public function testBar()
    {

    }
}
```

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. Les méthodes de configuration devraient initialiser les objets souhaités pour le test, et faire toute configuration souhaitée. Dans notre configuration nous ajouterons ce qui suit :

```
public function setUp()
{
    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 fabriquer les valeurs dans *Core\Configure* et stocker les chemins dans *Core\App*.

Ensuite, nous allons remplir les méthodes de test. Nous utiliserons quelques assertions pour nous assurer que notre code crée la sortie que nous attendons :

```
public function testBar()
{
    $result = $this->Progress->bar(90);
```

(suite sur la page suivante)

(suite de la page précédente)

```
    $this->assertContains('width: 90%', $result);
    $this->assertContains('progress-bar', $result);

    $result = $this->Progress->bar(33.3333333);
    $this->assertContains('width: 33%', $result);
}
```

Le test ci-dessus est simple mais montre le potentiel bénéfique de l'utilisation des cas de test. Nous utilisons `assertContains()` 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 aide à être plus confiant sur le code que vous écrivez puisque vous pouvez vérifier que le code que vous écrivez remplit les attentes et les assertions que vos tests font. De plus, puisque les tests sont du code, ils peuvent être re-lancés dès que vous faîtes un changement. Cela évite la création de nouveaux bugs.

Lancer les Tests

Une fois que vous avez installé PHPUnit et que quelques cas de tests sont écrits, vous pouvez lancer les cas de test très fréquemment. C'est une bonne idée de lancer les tests avant de committer tout changement pour aider à s'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 :

```
# avec l'installation de composer
$ vendor/bin/phpunit

# avec le fichier phar
php phpunit.phar
```

Si vous avez cloné la source de CakePHP à partir de GitHub [162] 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 parti de la source de votre application, d'abord faîtes la commande `cd` vers le répertoire du plugin, ensuite utilisez la commande `phpunit` qui correspond à la façon dont vous avez installé phpunit :

```
cd plugins

# En utilisant phpunit installé avec composer
../vendor/bin/phpunit

# En utilisant le fichier phar
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 :

162. https://github.com/cakephp/cakephp

```
git clone git://github.com/cakephp/debug_kit.git
cd debug_kit
php ~/composer.phar install
php ~/phpunit.phar
```

Filtrer les Cas de Test (TestCase)

Quand vous avez des cas de test plus larges, vous pouvez lancer un sous-ensemble de méthodes de test quand vous essayez de travailler sur un cas unique d'échec. Avec l'exécuteur CLI vous pouvez utiliser une option pour filtrer les méthodes de test :

```
$ 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 en une ligne de commande en utilisant les outils de couverture de code intégrés à PHPUnit. PHPUnit va générer un ensemble de fichiers en HTML statique contenant les résultats de la couverture. Vous pouvez générer une couverture pour un cas de test en faisant ce qui suit :

```
$ phpunit --coverage-html webroot/coverage tests/TestCase/Model/Table/
↪ArticlesTableTest
```

Cela mettra la couverture des résultats dans le répertoire webroot de votre application. Vous pourrez voir les résultats en allant à `http://localhost/votre_app/coverage`.

Si vous utilisez PHP 5.6.0 (ou supérieur), vous pouvez utilisez phpdbg [163] 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 composé de plusieurs plugins. Dans ces situations, il peut être assez fastidieux d'effectuer des tests pour chaque plugin. Vous pouvez faire des tests pour chaque plugin qui compose votre application en ajoutant une section `<testsuite>` supplémentaire au fichier `phpunit.xml.dist` de votre application :

```
<testsuites>
    <testsuite name="App Test Suite">
        <directory>./tests/TestCase</directory>
    </testsuite>

    <!-- Ajouter vos plugins -->
    <testsuite name="Forum plugin">
        <directory>./plugins/Forum/tests/TestCase</directory>
    </testsuite>
</testsuites>
```

163. http://phpdbg.com/

Les tests supplémentaires ajoutés à l'élément `<testsuites>` seront exécutés automatiquement quand quand vous utiliserez `phpunit`.

Si vous utilisez `<testsuites>` pour utiliser les fixtures à partir des plugins que vous avez installé avec composer, le fichier `composer.json` du plugin doit ajouter le namespace de la fixture à la section autoload. Exemple :

```
"autoload": {
    "psr-4": {
        "PluginName\\Test\\Fixture\\": "tests\\Fixture"
    }
},
```

Les Callbacks du Cycle de Vie des Cas de Test

Les cas de Test ont un certain nombre de callbacks de cycle de vie que vous pouvez utiliser quand vous faîtes les tests :

- `setUp` est appelé avant chaque méthode de test. Doit être utilisé pour créer les objets qui vont être testés, et initialiser toute donnée pour le test. Toujours se rappeler d'appeler `parent::setUp()`.
- `tearDown` est appelé après chaque méthode de test. Devrait être utilisé pour nettoyer une fois que le test est terminé. Toujours se rappeler d'appeler `parent::tearDown()`.
- `setupBeforeClass` est appelé une fois avant que les méthodes de test aient commencées dans un cas. Cette méthode doit être *statique*.
- `tearDownAfterClass` est appelé une fois après que les méthodes de test ont commencé dans un cas. Cette méthode doit être *statique*.

Fixtures

Quand on teste du code qui dépend de models et d'une base de données, on peut utiliser les **fixtures** comme une façon de générer temporairement des tables de données chargées avec des données d'exemple qui peuvent être utilisées par le test. Le bénéfice de l'utilisation de fixtures est que votre test n'a aucune chance d'abîmer les données de l'application qui tourne. De plus, vous pouvez commencer à tester votre code avant de développer réellement en live le contenu pour une application.

CakePHP utilise la connexion nommée `test` dans votre fichier de configuration **config/app.php**. Si la connexion n'est pas utilisable, une exception sera levée et vous ne pourrez pas utiliser les fixtures de la base de données.

CakePHP effectue ce qui suit pendant le déroulement d'une fixture basée sur un cas de test :

1. Crée les tables pour chacune des fixtures nécessaires.
2. Remplit les tables avec les données, si les données sont fournies dans la fixture.
3. Lance les méthodes de test.
4. Vide les tables de fixture.
5. Retire les tables de fixture de la base de données.

Connexions de Test

Par défaut, CakePHP va faire un alias pour chaque connexion de votre application. Chaque connexion définie dans le bootstrap de votre application qui ne commence pas par `test_`, va avoir un alias avec le prefix `test_` créé. Les alias de connexion assurent que vous n'utiliserez pas accidentellement la mauvaise connexion dans les cas de test. Les alias de connexion sont transparents pour le reste de votre application. Par exemple, si vous utilisez la connexion "default", à la place, vous obtiendrez la connexion `test` dans les cas de test. Si vous utilisez la connexion "replica", la suite de tests va tenter d'utiliser "test_replica".

Créer les Fixtures

A la création d'une fixture, vous pouvez définir principalement deux choses : comment la table est créée (quels champs font partie de la table), et quels enregistrements seront remplis initialement dans la table. 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';

      public $fields = [
          'id' => ['type' => 'integer'],
          'title' => ['type' => 'string', 'length' => 255, 'null' => false],
          'body' => 'text',
          'published' => ['type' => 'integer', 'default' => '0', 'null' => false],
          'created' => 'datetime',
          'modified' => 'datetime',
          '_constraints' => [
            'primary' => ['type' => 'primary', 'columns' => ['id']]
          ]
      ];
      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',
```

(suite sur la page suivante)

(suite de la page précédente)

```
            '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 les valeurs aux colonnes qui s'incrémentent automatiquement 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, mais préfixé avec `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 vont utiliser la source de données `test` par défaut. Les sources de données de fixture doivent être préfixées par `test` pour réduire la possibilité de trucher accidentellement toutes les données de votre application quand vous lancez des tests.

Nous utilisons `$fields` pour spécifier les champs qui feront parti de cette table, et comment ils sont définis. Le format utilisé pour définir ces champs est le même qu'utilisé avec `CakeSchema`. Les clés disponibles pour la définition de la table sont :

type Type de données interne à CakePHP. Actuellement supportés : - `string` : redirige vers `VARCHAR`. - `uuid` : redirige vers `UUID` - `text` : redirige vers `TEXT`. - `integer` : redirige vers `INT`. - `biginteger` : redirige vers `BIGINTEGER` - `decimal` : redirige vers `DECIMAL` - `float` : redirige vers `FLOAT`. - `datetime` : redirige vers `DATETIME`. - `timestamp` : redirige vers `TIMESTAMP`. - `time` : redirige vers `TIME`. - `date` : redirige vers `DATE`. - `binary` : redirige vers `BLOB`.

fixed Utilisé avec les types `string` pour créer des colonnes de type `CHAR` dans les plates-formes qui les supportent.

length Défini à la longueur spécifique que le champ doit prendre.

precision Défini le nombre de décimales utilisées sur les champs `float` et `decimal`.

null Défini soit à `true` (pour permettre les NULLs) soit à `false` (pour ne pas permettre les NULLs).

default Valeur par défaut que le champ prend.

Nos pouvons définir un ensemble d'enregistrements qui seront remplis après que la table de fixture est créée. Le format est assez simple, `$records` est un tableau d'enregistrements. Chaque item dans `$records` doit être un enregistrement (une seule ligne). A 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 chaque enregistrement dans le tableau $records doit avoir une clé pour **chaque** champ spécifié dans le tableau `$fields`. Si un champ pour un enregistrement particulier a besoin d'avoir une valeur `null`, spécifiez juste la valeur de cette clé à `null`.

Les Données Dynamiques et les Fixtures

Depuis que les enregistrements pour une fixture sont déclarés en propriété de classe, vous ne pouvez pas utiliser les fonctions ou autres données dynamiques pour définir les fixtures. Pour résoudre ce problème, vous pouvez définir `$records` dans la fonction `init()` de votre fixture. Par exemple, si vous voulez que tous les timestamps soient créés et mis à jours pour refléter la date d'aujourd'hui, vous pouvez faire ce qui suit :

```
namespace App\Test\Fixture;

use Cake\TestSuite\Fixture\TestFixture;
```

(suite sur la page suivante)

(suite de la page précédente)

```
class ArticlesFixture extends TestFixture
{

    public $fields = [
        'id' => ['type' => 'integer'],
        'title' => ['type' => 'string', 'length' => 255, 'null' => false],
        'body' => 'text',
        'published' => ['type' => 'integer', 'default' => '0', 'null' => false],
        'created' => 'datetime',
        'modified' => 'datetime',
        '_constraints' => [
            'primary' => ['type' => 'primary', 'columns' => ['id']],
        ]
    ];

    public function init()
    {
        $this->records = [
            [
                'title' => 'First Article',
                'body' => 'First Article Body',
                'published' => '1',
                'created' => date('Y-m-d H:i:s'),
                'modified' => date('Y-m-d H:i:s'),
            ],
        ];
        parent::init();
    }
}
```

Quand vous surchargez `init()`, rappelez-vous juste de toujours appeler `parent::init()`.

Importer les Informations de Table

Définir le schema des fixtures peut être vraiment pratique lorsque vous créez des plugins, des librairies ou si vous créez un application qui doit être portable. La redéfinition du schéma dans les fixtures peut devenir difficile à maintenir pour les applications de grandes échelles. A cause de cela, CakePHP fournit la possibilité d'importer le schema depuis une connexion existante et utilise une définition de la table réfléchie pour créer la définition de la table utilisée par la suite de tests.

Commençons par un exemple. Imaginons que vous ayez un model nommé articles disponible dans votre application (qui est lié avec une table nommée articles), on changerait la fixture donnée dans la section précédente (**tests/Fixture/ArticlesFixture.php**) en ce qui suit :

```
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'];
}
```

Nouveau dans la version 3.1.7.

En général vous avez une classe Table avec votre fixture. Vous pouvez aussi utiliser ceci pour récupérer le nom de la table :

```
class ArticlesFixture extends TestFixture
{
    public $import = ['model' => 'Articles'];
}
```

Puisqu'on utilise `TableRegistry::getTableLocator()->get()`, on peut aussi utiliser la syntaxe de plugin.

Vous pouvez naturellement importer la définition de votre table à partir d'un model/d'une table existante, mais vous avez vos enregistrements directement définis dans le fixture comme il a été montré dans la section précédente. 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'
        ]
    ];
}
```

Vous pouvez ne pas charger/créer schéma dans une fixture. Ceci est utile si vous aviez déjà une configuration de base de données de test, avec toutes les tables vides créées. En ne définissant ni `$fields` ni `$import`, une fixture va seulement insérer les enregistrements et tronquer les enregistrements sur chaque méthode de test.

Charger les Fixtures dans vos Tests (TestCase)

Après avoir créé vos fixtures, vous pouvez les utiliser dans vos cas de test. Dans chaque cas de test vous devriez charger les fixtures dont vous aurez besoin. Vous devriez charger une fixture pour chaque model qui aura une requête lancée contre elle. Pour charger les fixtures, vous définissez la propriété `$fixtures` dans votre model :

```
class ArticleTest extends TestCase
{
    public $fixtures = ['app.articles', 'app.comments'];
}
```

Ce qui est au-dessus va charger les fixtures d'Article et de Comment à partir du répertoire de fixture de l'application. Vous pouvez aussi charger les fixtures à partir du cœur de CakePHP ou des plugins :

```
class ArticlesTest extends TestCase
{
    public $fixtures = ['plugin.DebugKit.articles', 'plugin.MyVendorName/MyPlugin.
↪messages', 'core.comments'];
}
```

Utiliser le préfixe `core` va charger les fixtures à partir de CakePHP, et utiliser un nom de plugin en préfixe chargera la fixture à partir d'un plugin nommé.

Vous pouvez contrôler quand vos fixtures sont chargées en configurant `Cake\TestSuite\TestCase::$autoFixtures` à `false` et plus tard les charger en utilisant `Cake\TestSuite\TestCase::loadFixtures()` :

```
class ArticlesTest extends TestCase
{
    public $fixtures = ['app.articles', 'app.comments'];
    public $autoFixtures = false;

    public function testMyFunction()
    {
        $this->loadFixtures('Articles', 'Comments');
    }
}
```

Vous pouvez charger les fixtures dans les sous-répertoires. Utiliser plusieurs répertoires peut faciliter l'organisation de vos fixtures si vous avez une application plus grande. Pour charger les fixtures dans les sous-répertoires, incluez simplement le nom du sous-répertoire dans le nom de la fixture :

```
class ArticlesTableTest extends CakeTestCase
{
    public $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 croissantes avec la taille votre application. Il est possible qu'à un certain point, vous ne soyez plus en mesure les maintenir.

Le **`fixture factories plugin<https ://github.com/vierge-noire/cakephp-fixture-factories>`_** propose une alternative efficace pour des applications de taille moyenne et plus.

Le plugin utilise son propre phpunit listener [164], qui effectue les actions suivantes :

1. Faire tourner les migrations (description ici) [165].
2. Tronquer les tables utilisées au préalable avant chaque test.
3. Lancer les tests.

La commande bake suivante vous assistera pour créer vos factories :

```
bin/cake bake fixture_factory -h
```

Une fois vos factories mises en place [166], vous voilà équipés pour créer vos fixtures de test à vitesse folle.

Les intéractions non nécessaires avec la base de donnée ralentissent 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'intéragissent pas avec la base de donnée :

```
$article = ArticleFactory::make()->getEntity();
```

Pour insérer dans la base de donnée :

```
$article = ArticleFactory::make()->persist();
```

En supposant que les articles appartiennent à plusieurs auteurs, 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 la création, ni la déclaration de fixtures, elles sont parfaitement compatibles avec ces dernières. Pour plus de détails, rendez-vous ici [167].

Tester les Classes Table

Disons que nous avons déjà notre table Articles définie dans **src/Model/Table/ArticlesTable.php**, qui 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)
```

(suite sur la page suivante)

164. https://github.com/vierge-noire/cakephp-test-suite-light
165. https://github.com/vierge-noire/cakephp-test-migrator
166. https://github.com/vierge-noire/cakephp-fixture-factories/blob/master/docs/factories.md
167. https://github.com/vierge-noire/cakephp-fixture-factories

(suite de la page précédente)

```
    {
        $query->where([
            $this->alias() . '.published' => 1
        ]);
        return $query;
    }
}
```

Nous voulons maintenant configurer un test qui va tester ce model tout en utilisant les Fixtures pour garder nos Tests isolés. 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\ORM\TableRegistry;
use Cake\TestSuite\TestCase;

class ArticlesTableTest extends TestCase
{
    public $fixtures = ['app.articles'];
}
```

Dans notre variable de cas de test `$fixtures`, 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\ORM\TableRegistry;
use Cake\TestSuite\TestCase;

class ArticlesTableTest extends TestCase
{
    public $fixtures = ['app.articles'];

    public function setUp()
    {
        parent::setUp();

        // Prior to 3.6 use TableRegistry::get('Articles')
        $this->Articles = TableRegistry::getTableLocator()->get('Articles');
    }

    public function testFindPublished()
    {
        $query = $this->Articles->find('published');
        $this->assertInstanceOf('Cake\ORM\Query', $query);
        $result = $query->hydrate(false)->toArray();
        $expected = [
```

(suite sur la page suivante)

(suite de la page précédente)

```
            ['id' => 1, 'title' => 'First Article'],
            ['id' => 2, 'title' => 'Second Article'],
            ['id' => 3, 'title' => 'Third Article']
        ];

        $this->assertEquals($expected, $result);
    }
}
```

Vous pouvez voir que nous avons ajouté une méthode appelée `testFindPublished()`. Nous commençons par créer une instance de notre model `Article`, et lançons ensuite notre méthode `published()`. Dans `$expected`, nous définissons ce que nous en attendons, ce qui devrait être le résultat approprié (que nous connaissons depuis que nous avons défini les enregistrements qui sont remplis initialement dans la table articles.). Nous testons que les résultats correspondent à nos attentes en utilisant la méthode `assertEquals()`. Regardez 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 présente ainsi :

```
namespace App\Test\TestCase\Model\Table;

use App\Model\Table\ArticlesTable;
use App\Test\Factory\ArticleFactory;
use Cake\TestSuite\TestCase;

class ArticlesTableTest extends TestCase
{
    public function setUp(): void
    { ... }

    public function testFindPublished(): void
    {
        // Insérer 3 articles publiés
        $articles = ArticleFactory::make(['published' => 1], 3)->persist();
        // Insérer 2 articles non publiés
        ArticleFactory::make(['published' => 0], 2)->persist();

        $result = $this->Articles->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'est déclarée. Les 5 articles créés n'existeront que pour ce test.

Méthodes de Mocking des Models

Il y aura des fois où vous voudrez mocker les méthodes sur les models quand vous les testez. Vous devrez utiliser `getMockForModel` pour créer les mocks de test des models. Cela évite des problèmes avec les propriétés réfléchies que les mocks normaux ont :

```
public function testSendingEmails()
{
    $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 :

```
TableRegistry::clear();
```

Test d'Intégrations 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 cette classe en tant que classe de base pour les cas de test de votre controller vous permet de mettre en place des tests d'intégration pour vos controllers.

Si vous n'êtes pas familier avec les tests d'intégrations, il s'agit d'une approche de test qui facilite le test de plusieurs éléments en même temps. Les fonctionnalités de test d'intégration dans CakePHP simulent une requête HTTP à traiter par votre application. Par exemple, tester vos controllers impactera les Models, Components et Helpers qui auraient été invoqués suite à une requête HTTP. Cela vous permet d'écrire des tests au plus haut niveau de votre application en ayant un impact sur chacun de ses travaux.

Disons que vous avez 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 $helpers = ['Form', 'Html'];

    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->Article->find('all', [
                'fields' => ['id', 'title']
```

(suite sur la page suivante)

(suite de la page précédente)

```
            ]);
        } else {
            $result = $this->Article->find();
        }

        $this->set([
            'title' => 'Articles',
            'articles' => $result
        ]);
    }
}
```

Créez un fichier nommé **ArticlesControllerTest.php** dans votre répertoire **tests/TestCase/Controller** et mettez ce qui suit à l'intérieur :

```
namespace App\Test\TestCase\Controller;

use Cake\ORM\TableRegistry;
use Cake\TestSuite\IntegrationTestCase;

class ArticlesControllerTest extends IntegrationTestCase
{
    public $fixtures = ['app.articles'];

    public function testIndex()
    {
        $this->get('/articles');

        $this->assertResponseOk();
        // D'autres asserts.
    }

    public function testIndexQueryData()
    {
        $this->get('/articles?page=1');

        $this->assertResponseOk();
        // D'autres asserts.
    }

    public function testIndexShort()
    {
        $this->get('/articles/index/short');

        $this->assertResponseOk();
        $this->assertResponseContains('Articles');
        // D'autres asserts.
    }

    public function testIndexPostData()
    {
        $data = [
            'user_id' => 1,
            'published' => 1,
            'slug' => 'new-article',
            'title' => 'New Article',
```

(suite sur la page suivante)

(suite de la page précédente)

```
            'body' => 'New Body'
        ];
        $this->post('/articles', $data);
        $this->assertResponseSuccess();

        // Prior to 3.6 use TableRegistry::get('Articles')
        $articles = TableRegistry::getTableLocator()->get('Articles');
        $query = $articles->find()->where(['title' => $data['title']]);
        $this->assertEquals(1, $query->count());
    }
}
```

Cet exemple montre quelques méthodes d'envoi de requête et quelques assertions qu'intègre `IntegrationTestCase`. Avant de pouvoir utiliser les assertions, vous aurez besoin de simuler une requête. Vous pouvez utiliser l'une des méthodes suivantes pour simuler 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 de saisir le corps d'une requête. Après avoir émis une requête, vous pouvez utiliser les différentes assertions que fournit `IntegrationTestCase` ou PHPUnit afin de vous assurer que votre requête possède de correctes effets secondaires.

Nouveau dans la version 3.5.0 : `options()` et `head()` ont été ajoutées dans 3.5.0.

Configurer la Requête

La classe `IntegrationTestCase` intègre de nombreux helpers pour faciliter la configuration des requêtes que vous allez envoyer à votre controller :

```
// Définit des cookies
$this->cookie('name', 'Uncle Bob');

// Définit des données de session
$this->session(['Auth.User.id', 1]);

// Configure les en-têtes
$this->configRequest([
    'headers' => ['Accept' => 'application/json']
]);
```

Les états de ces helpers définis par ces méthodes est remis à zéro dans la méthode `tearDown()`.

Tester des Actions Protégées par AuthComponent

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 de helper fournies par `IntegrationTestCase`. 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()
{
    // Pas de données de session définies.
    $this->get('/articles/add');

    $this->assertRedirect(['controller' => 'Users', 'action' => 'login']);
}

public function testAddAuthenticated()
{
    // 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 l'authentification stateless, vous pouvez, comme pour l'authentification basic, configurer la demande de manière à ce qu'elle injecte des variables d'environnement et des headers (en-têtes), ce qui permettra de simuler les en-têtes d'une demande d'authentification réelle.

Lorsque vous testez l'authentification simple (Basic) ou de type « Digest », vous pouvez ajouter les variables d'environnement que PHP crée *<http ://php.net/manual/fr/features.http-auth.php> `_ automatiquement. Ces variables d'environnement utilisées dans l'adaptateur d'authentification sont décrites dans : ref : `basic-authentication*

```
public function testBasicAuthentication()
{
    $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()
{
    $this->configRequest([
        'headers' => [
            'authorization' => 'Bearer: oauth-token'
        ]
    ]);

    $this->get('/api/posts');
    $this->assertResponseOk();
}
```

La clé des en-têtes dans `configRequest()` peut être utilisée pour configurer tout en-tête HTTP supplémentaires nécessaires à une action.

Tester les Actions Protégées par CsrfComponent ou SecurityComponent

Quand vous testez les actions protégées par SecurityComponent ou CsrfComponent, vous pouvez activer la génération automatique de token pour vous assurer que vos tests ne vont pas être en échec à cause d'un token non présent :

```
public function testAdd()
{
    $this->enableCsrfToken();
    $this->enableSecurityToken();
    $this->post('/posts/add', ['title' => 'News excitante!']);
}
```

Il est aussi important d'activer debug dans les tests qui utilisent les tokens pour éviter que le SecurityComponent pense que le token debug est utilisé dans un environnement non-debug. Quand vous testez 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']
]);
```

Nouveau dans la version 3.1.2 : Les méthodes `enableCsrfToken()` et `enableSecurityToken()` ont été ajoutées dans la version 3.1.2.

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, `IntegrationTestCase` détectera automatiquement la présence d'une classe `App\Application` et activera automatiquement les tests d'intégration sur votre Application. Vous pouvez activer / désactiver ce comportement avec la méthode `useHttpServer()` :

```
public function setUp()
{
    // Active les tests d'intégration PSR-7
    $this->useHttpServer(true);
```

(suite sur la page suivante)

(suite de la page précédente)

```
    // Désactive les tests d'intégration PSR-7
    $this->useHttpServer(false);
}
```

Vous pouvez personnaliser le nom de la classe Application utilisé ainsi que les arguments du contructeur en utilisant la méthode `configApplication()` :

```
public function setUp()
{
    $this->configApplication('App\App', [CONFIG]);
}
```

Après avoir activé le mode PSR-7 (et avoir peut-être configuré la classe d'Application), vous pouvez utiliser le reste des fonctionnalités de `IntegrationTestCase` normalement.

Vous devriez également faire en sorte d'utiliser *Application : :bootstrap()* pour charger les plugins qui contiennent des événements et des routes. De cette manière, vous vous assurez que les événements et les routes seront connectés pour chacun de vos « test case ».

Nouveau dans la version 3.3.0 : Les Middleware PSR-7 et la méthode `useHttpServer()` ont été ajoutée avec 3.3.0.

Tester avec des cookies chiffrés

Si vous utilisez le *`Cake\Controller\Component\CookieComponent`* dans vos controllers, vos cookies sont probablement chiffrés. Depuis 3.1.7, CakePHP met à votre disposition des méthodes pour intéragir avec les cookies chiffrés dans vos « test cases » :

```
// Définit un cookie en utilisant AES et la clé par défaut.
$this->cookieEncrypted('my_cookie', 'Some secret values');

// Partons du principe que cette requête modifie le cookie.
$this->get('/bookmarks/index');

$this->assertCookieEncrypted('An updated value', 'my_cookie');
```

Nouveau dans la version 3.1.7 : `assertCookieEncrypted` et `cookieEncrypted` ont été ajoutées dans 3.1.7.

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 pour que vous puissez écrire vos assertions :

```
$this->enableRetainFlashMessages();
$this->get('/bookmarks/delete/9999');

$this->assertSession('That bookmark does not exist', 'Flash.flash.0.message');
```

Nouveau dans la version 3.4.7 : `enableRetainFlashMessages()` a été ajoutée dans 3.4.7

Tester un controller retournant du JSON

JSON est un format commun à utiliser lors de la conception de web service. Tester les points de terminaisons (end-points) de votre web service est très simple avec CakePHP. Commençons avec un simple exemple de controller qui renvoie du JSON :

```
class MarkersController extends AppController
{
    public function initialize()
    {
        parent::initialize();
        $this->loadComponent('RequestHandler');
    }

    public function view($id)
    {
        $marker = $this->Markers->get($id);
        $this->set([
            '_serialize' => ['marker'],
            'marker' => $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()
    {
        $this->configRequest([
            'headers' => ['Accept' => 'application/json']
        ]);
        $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, $this->_response->body());
    }
}
```

Nous utilisons l'option `JSON_PRETTY_PRINT` car la vue qui retourne la représentation JSON intégrée à CakePHP (JsonView) utilise cette option quand le mode `debug` est activé.

Désactiver le Middleware de Gestion d'Erreurs dans les Tests

Quand vous debuggez des tests qui échouent car l'application a rencontré des erreurs, il peut être utile de désactiver temporairement le middleware de gestion des erreurs pour permettre aux erreurs de remonter. Vous pouvez utiliser la méthode `disableErrorHandlerMiddleware()` pour permettre ce comportement :

```
public function testGetMissing()
{
    $this->disableErrorHandlerMiddleware();
    $this->get('/markers/not-there');
    $this->assertResponseCode(404);
}
```

Dans l'exemple ci-dessus, le test échouera et le message d'exception et le stack-trace seront affichés à la place de la page d'erreur de l'application.

Nouveau dans la version 3.5.0.

Méthodes d'Assertion

La classe `IntegrationTestCase` vous fournis de nombreuses méthodes d'assertions afin de tester plus simplement les réponses. 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 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('Yeah!');
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Vérifie un contenu partiel de la réponse
$this->assertResponseContains('You won!');
$this->assertResponseNotContains('You lost!');

// Vérifie le layout
$this->assertLayout('default');

// Vérifie quel Template a été rendu.
$this->assertTemplate('index');

// Vérifie les données de la session
$this->assertSession(1, 'Auth.User.id');

// Vérifie l'entête de la réponse.
$this->assertHeader('Content-Type', 'application/json');

// 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 [168] et celles de PHPUnit [169].

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 la sortie rendue d'une view. `StringCompareTrait` ajoute une méthode d'assertion simple pour cela.

Pour l'utiliser, vous devez inclure un Trait, définir le chemin de base de comparaison et appeler `assertSameAsFile`:

```
use Cake\TestSuite\StringCompareTrait;
use Cake\TestSuite\TestCase;

class SomeTest extends TestCase
{
    use StringCompareTrait;

    public function setUp()
    {
        $this->_compareBasePath = APP . 'tests' . DS . 'comparisons' . DS;
        parent::setUp();
    }

    public function testExample()
    {
```

(suite sur la page suivante)

168. https://api.cakephp.org/3.x/class-Cake.TestSuite.TestCase.html
169. https://phpunit.de/manual/current/en/appendixes.assertions.html

(suite de la page précédente)

```
        $result = ...;
        $this->assertSameAsFile('example.php', $result);
    }
}
```

L'exemple ci-dessus va comparer `$result` au contenu du fichier `APP/tests/comparisons/example.php`.

Un mécanisme est fourni 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 au fur et à mesure où ils sont rendus :

```
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
```

Tester avec des Cookies Chiffrés

Si vous utilisez *Cake\Controller\Component\CookieComponent* dans vos controllers, vos cookies sont probablement chiffrés. Depuis 3.1.7, CakePHP fournit des méthodes pour intéragir avec les cookies chiffrés dans vos cas de test :

```
// Définit un cookie en utilisant aes et la clé par défaut.
$this->cookieEncrypted('my_cookie', 'Some secret values');

// En supposant que cette action modifie le cookie.
$this->get('/bookmarks/index');

$this->assertCookieEncrypted('Une valeur mise à jour', 'my_cookie');
```

Tester un Controller dont la Réponse est au format JSON

JSON est un format sympa et courant à utiliser quand on construit un service web. Tester les endpoints de votre service web est très simple avec CakePHP. Commençons par un exemple de controller simple qui répond en JSON :

```
class MarkersController extends AppController
{
    public $components = ['RequestHandler'];

    public function view($id)
    {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        $marker = $this->Markers->get($id);
        $this->set([
            '_serialize' => ['marker'],
            'marker' => $marker,
        ]);
    }
}
```

Maintenant créons un fichier **tests/TestCase/Controller/MarkersControllerTest.php** et assurons-nous que notre service web retourne une réponse appropriée :

```
class MarkersControllerTest extends IntegrationTestCase
{

    public function testGet()
    {
        $this->configRequest([
            'headers' => ['Accept' => 'application/json']
        ]);
        $result = $this->get('/markers/view/1.json');

        // Vérifie que le code de réponse est 200
        $this->assertResponseOk();

        $expected = [
            ['id' => 1, 'lng' => 66, 'lat' => 45],
        ];
        $expected = json_encode($expected, JSON_PRETTY_PRINT);
        $this->assertEquals($expected, $this->_response->body());
    }
}
```

Nous utilisons l'option `JSON_PRETTY_PRINT` comme le fait CakePHP à partir de la classe JsonView. Ce dernier utilise cette option quand le mode `debug` est activé. Vous pouvez utiliser ceci afin que votre test marche dans les deux cas :

```
json_encode($data, Configure::read('debug') ? JSON_PRETTY_PRINT : 0);
```

Tests d'Intégration de la Console

Pour faciliter les tests de vos applications console, CakePHP est doté d'une classe `ConsoleIntegrationTestCase` qui peut être utilisée pour tester vos applications consoles et faire des assertions sur leurs résultats.

Nouveau dans la version 3.5.0 : `ConsoleIntegrationTestCase` a été ajoutée dans 3.5.0.

Pour commencer à tester votre application console, créez un « test case » qui étend `Cake\TestSuite\ConsoleIntegrationTestCase`. Cette classe contient une méthode `exec()` qui est utilisée pour exécuter votre commande. Vous pouvez passer la même chaîne que vous passeriez au CLI dans cette méthode.

Commençons par créer un shell très simple, stocké dans **src/Shell/MyConsoleShell.php** :

```
namespace App\Shell;
```

(suite sur la page suivante)

(suite de la page précédente)

```
use Cake\Console\ConsoleOptionParser;
use Cake\Console\Shell;

class MyConsoleShell extends Shell
{
    public function getOptionParser()
    {
        $parser = new ConsoleOptionParser();
        $parser->setDescription('My cool console app');

        return $parser;
    }
}
```

Pour écrire un test d'intégration pour ce shell, on va créer un « test case » dans **tests/TestCase/Shell/MyConsoleShellTest.php** qui *extends* `Cake\TestSuite\ConsoleIntegrationTestCase`. Ce shell ne fait pas grand chose pour le moment, mais testons que la description de notre shell est affichée dans `stdout` :

```
namespace App\Test\TestCase\Shell;

use Cake\TestSuite\ConsoleIntegrationTestCase;

class MyConsoleShellTest extends ConsoleIntegrationTestCase
{
    public function testDescriptionOutput()
    {
        $this->exec('my_console');
        $this->assertOutputContains('My cool console app');
    }
}
```

Les tests passent ! Même si c'est un exemple très simple, cela prouve que construire un test d'intégration pour une application console est facile. Continuons en ajoutant des sous-commandes et des options à notre shell :

```
namespace App\Shell;

use Cake\Console\ConsoleOptionParser;
use Cake\I18n\FrozenTime;

class MyConsoleShell extends Shell
{
    public function getOptionParser()
    {
        $parser = new ConsoleOptionParser();

        $updateModifiedParser = new ConsoleOptionParser();
        $updateModifiedParser->addArgument('table', [
            'help' => 'Table to update',
            'required' => true
        ]);

        $parser
            ->setDescription('My cool console app')
            ->addSubcommand('updateModified', [
                'parser' => $updateModifiedParser
```

(suite sur la page suivante)

(suite de la page précédente)

```
            ]);

        return $parser;
    }

    public function updateModified()
    {
        $table = $this->args[0];
        $this->loadModel($table);
        $this->{$table}->query()
            ->update()
            ->set([
                'modified' => new FrozenTime()
            ])
            ->execute();
    }
}
```

C'est maintenant un shell plus complexe avec une sous-commande et son propre parser. Testons la sous-commande `updateModified`. Modifier votre « test case » avec le morceau de code suivant :

```
namespace Cake\Test\TestCase\Shell;

use Cake\Console\Shell;
use Cake\I18n\FrozenTime;
use Cake\ORM\TableRegistry;
use Cake\TestSuite\ConsoleIntegrationTestCase;

class MyConsoleShellTest extends ConsoleIntegrationTestCase
{

    public $fixtures = [
        // assumes you have a UsersFixture
        'app.users'
    ];

    public function testDescriptionOutput()
    {
        $this->exec('my_console');
        $this->assertOutputContains('My cool console app');
    }

    public function testUpdateModified()
    {
        $now = new FrozenTime('2017-01-01 00:00:00');
        FrozenTime::setTestNow($now);

        $this->loadFixtures('Users');

        $this->exec('my_console update_modified Users');
        $this->assertExitCode(Shell::CODE_SUCCESS);

        // Prior to 3.6 use TableRegistry::get('Users')
        $user = TableRegistry::getTableLocator()->get('Users')->get(1);
        $this->assertSame($user->modified->timestamp, $now->timestamp);
```

(suite sur la page suivante)

(suite de la page précédente)

```
        FrozenTime::setTestNow(null);
    }
}
```

Comme vous pouvez le voir via la méthode `testUpdateModified`, nous testons que la sous-commande `update_modified` met à jour la table que nous passons comme premier argument. Premièrement, nous faisons l'assertion que le shell a terminé de s'exécuter avec le bon code de statut, `0`. Ensuite, nous testons que notre sous-commande a fait son travail, c'est-à-dire qu'elle a correctement mis à jour la colonne `modified` de la table que nous avons passée en argument.

Gardez bien en mémoire que la méthode `exec()` accepte la même chaîne que ce que vous tapez dans votre CLI, donc vous pouvez ajouter des options et des arguments pour tester un maximum de cas.

Tester les Shells Interactifs

Les applications console sont souvent interactives. Tester les shells interactifs avec la classe `Cake\TestSuite\ConsoleIntegrationTestCase` va seulement nécessiter que vous passiez les données attendues comme second paramètre de la méthode `exec()`. Ces données doivent être passées sous forme de tableau, dans l'ordre dans lequel ces données sont attendues.

En continuant avec notre shell d'exemple, ajoutons une sous-commande interactive. Mettez à jour la classe de shell avec le code suivant :

```
namespace App\Shell;

use Cake\Console\ConsoleOptionParser;
use Cake\Console\Shell;
use Cake\I18n\FrozenTime;

class MyConsoleShell extends Shell
{
    public function getOptionParser()
    {
        $parser = new ConsoleOptionParser();

        $updateModifiedParser = new ConsoleOptionParser();
        $updateModifiedParser->addArgument('table', [
            'help' => 'Table to update',
            'required' => true
        ]);

        $parser
            ->setDescription('My cool console app')
            ->addSubcommand('updateModified', [
                'parser' => $updateModifiedParser
            ])
            // add a new subcommand
            ->addSubcommand('bestFramework');

        return $parser;
    }

    public function updateModified()
    {
        $table = $this->args[0];
```

(suite sur la page suivante)

(suite de la page précédente)

```
        $this->loadModel($table);
        $this->{$table}->query()
            ->update()
            ->set([
                'modified' => new FrozenTime()
            ])
            ->execute();
    }

    // create this interactive subcommand
    public function bestFramework()
    {
        $this->out('Hi there!');

        $framework = $this->in('What is the best PHP framework?');
        if ($framework !== 'CakePHP') {
            $this->err("I disagree that '$framework' is the best.");
            $this->_stop(Shell::CODE_ERROR);
        }

        $this->out('I agree!');
    }
}
```

Maintenant que nous avons une sous-commande interactive, nous pouvons ajouter un test qui va permettre de vérifier que nous recevons la réponse attendue et un test où nous passerons une réponse que nous savons incorrecte. Ajoutez les méthodes suivantes dans **tests/TestCase/Shell/MyConsoleShellTest.php** :

```
public function testBestFramework()
{
    $this->exec('my_console best_framework', [
        'CakePHP'
    ]);
    $this->assertExitCode(Shell::CODE_SUCCESS);
    $this->assertOutputContains('I agree!');
}

public function testBestFrameworkWrongAnswer()
{
    $this->exec('my_console best_framework', [
        'my homemade framework'
    ]);
    $this->assertExitCode(Shell::CODE_ERROR);
    $this->assertErrorRegExp("/I disagree that \'(.+)\' is the best\./");
}
```

Comme vous pouvez le voir dans `testBestFramework`, il répond à la première valeur passée « CakePHP ». Puisque que c'est la réponse attendue par notre sous-commande, le shell termine avec un succès après avoir retourné une réponse.

Le second test, `testBestFrameworkWrongAnswer`, passe une réponse invalide ce qui fait que notre shell échoue et retourne le code `1`. Nous faisons également l'assertion que `stderr` a bien reçu notre erreur et que le retour contient bien la valeur incorrecte passée en paramètre.

Tester le CommandRunner

Pour tester les shells qui sont « dispatchées » via la classe `CommandRunner`, activer le dans votre « test case » avec la méthode suivante :

```
$this->useCommandRunner();
```

Nouveau dans la version 3.5.0 : La classe `CommandRunner` a été ajoutée dans 3.5.0.

Méthodes d'assertions

La classe `Cake\TestSuite\ConsoleIntegrationTestCase` met à disposition plusieurs méthodes qui facilitent les assertions des sorties de console :

```
// s'assure que le shell a quitté avec le code attendu
$this->assertExitCode($expected);

// s'assure que stdout contient une chaîne
$this->assertOutputContains($expected);

// s'assure que stderr contient une chaîne
$this->assertErrorContains($expected);

// s'assure que stdout "match" une regex
$this->assertOutputRegExp($expected);

// s'assure que stderr "match" une regex
$this->assertErrorRegExp($expected);
```

Tester les Views

Généralement, la plupart des applications ne va pas directement tester leur code HTML. Faire ça donne souvent des résultats fragiles, il est difficile de maintenir les suites de test qui sont sujet à se casser. En écrivant des tests fonctionnels en utilisant `ControllerTestCase`, vous pouvez inspecter le contenu de la vue rendue en configurant l'option `return` à "view". Alors qu'il est possible de tester le contenu de la vue en utilisant ControllerTestCase, un test d'intégration/vue plus robuste et maintenable peut être effectué en utilisant des outils comme Selenium webdriver [170].

Tester les Components

Imaginons que nous avons un component appelé PagematronComponent dans notre application. Ce component nous aide à paginer la valeur limite à travers tous les controllers qui l'utilisent. Voici notre exemple de component localisé dans **src/Controller/Component/PagematronComponent.php** :

```
class PagematronComponent extends Component
{
    public $controller = null;

    public function setController($controller)
    {
```

(suite sur la page suivante)

170. http://seleniumhq.org

(suite de la page précédente)

```
        $this->controller = $controller;
        // Assurez-vous que le controller utilise la pagination.
        if (!isset($this->controller->paginate)) {
            $this->controller->paginate = [];
        }
    }

    public function startup(Event $event)
    {
        $this->setController($event->getSubject());
    }

    public function adjust($length = 'short')
    {
        switch ($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;

class PagematronComponentTest extends TestCase
{

    public $component = null;
    public $controller = null;

    public function setUp()
    {
        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])
```

(suite sur la page suivante)

(suite de la page précédente)

```
            ->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()
    {
        // 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()
    {
        parent::tearDown();
        // Nettoie les variables quand les tests sont finis.
        unset($this->component, $this->controller);
    }
}
```

Tester les Helpers

Puisqu'un bon nombre de logique se situe dans les classes Helper, il est important de s'assurer que ces classes sont couvertes par des cas de test.

Tout d'abord, nous créons un exemple de helper à tester. `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($amount)
    {
        return 'USD ' . number_format($amount, 2, '.', ',');
    }
}
```

Ici nous définissons la décimale à 2 après la virgule, le séparateur de décimal, le séparateur des centaines avec une virgule, et le nombre formaté avec la chaîne "USD" en préfixe.

Maintenant nous 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;

    // Nous instancions notre helper
    public function setUp()
    {
        parent::setUp();
        $View = new View();
        $this->helper = new CurrencyRendererHelper($View);
    }

    // Test de la fonction usd()
    public function testUsd()
    {
        $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)
        );
    }
}
```

Ici nous appelons `usd()` avec des paramètres différents et disons à test suite de vérifier si les valeurs retournées sont égales à ce que nous en attendons.

Sauvegardons cela et exécutons le test. Vous devriez voir une barre verte et un message indiquant 1 passé et 4 assertions.

Lorsque vous testez un Helper qui utilise d'autres Helpers, assurez-vous de créer un mock de 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 nous les testons, nous avons tendance à tester les événements dans les cas de test qui éxecutent 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, disons que nous avons les tables suivantes :

```
class OrdersTable extends Table
{

    public function place($order)
    {
        if ($this->save($order)) {
            // moved cart removal to CartsTable
            $event = new Event('Model.Order.afterPlace', $this, [
                'order' => $order
            ]);
            $this->eventManager()->dispatch($event);
            return true;
        }
        return false;
    }
}

class CartsTable extends Table
{

    public function implementedEvents()
    {
        return [
            'Model.Order.afterPlace' => 'removeFromCart'
        ];
    }

    public function removeFromCart(Event $event)
    {
        $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 pour lequel vous souhaitez faire des asserts.

Pour tester le `OrdersTable` du dessus, vous devez activer le tracking dans la méthode `setUp()` puis vérifier par exemple 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\ORM\TableRegistry;
use Cake\TestSuite\TestCase;
```

(suite sur la page suivante)

(suite de la page précédente)

```
class OrdersTableTest extends TestCase
{

    public $fixtures = ['app.orders'];

    public function setUp()
    {
        parent::setUp();

        // Prior to 3.6 use TableRegistry::get('Orders')
        $this->Orders = TableRegistry::getTableLocator()->get('Orders');

        // enable event tracking
        $this->Orders->eventManager()->setEventList(new EventList());
    }

    public function testPlace()
    {
        $order = new Order([
            'user_id' => 1,
            'item' => 'Cake',
            'quantity' => 42,
        ]);

        $this->assertTrue($this->Orders->place($order));

        $this->assertEventFired('Model.Order.afterPlace', $this->Orders->
↪eventManager());
        $this->assertEventFiredWith('Model.Order.afterPlace', 'order', $order, $this->
↪Orders->eventManager());
    }
}
```

Par défaut, l'`EventManager` global est utilisé pour les assertions, donc tester les événements globaux ne nécessitent pas de passer le gestionnaire d'événements :

```
$this->assertEventFired('My.Global.Event');
$this->assertEventFiredWith('My.Global.Event', 'user', 1);
```

Nouveau dans la version 3.2.11 : Le tracking d'événement, `assertEventFired()`, et `assertEventFiredWith` ont été ajoutés.

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>
```

(suite sur la page suivante)

(suite de la page précédente)

```
  </testsuite>
</testsuites>
```

Créer des Tests pour les Plugins

Les Tests pour les plugins sont créés dans leur propre répertoire à l'intérieur du 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` à partir du chapitre des plugins de ce manuel. Une différence par rapport aux autres test est dans la première ligne où "Blog.BlogPost" est importé. Vous devrez aussi préfixer les fixtures de votre plugin avec `plugin.blog.blog_posts` :

```
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/
    public $fixtures = ['plugin.blog.blog_posts'];

    public function testSomething()
    {
        // Teste quelque chose.
    }
}
```

Si vous voulez utiliser les fixtures de plugin dans les app tests, vous pouvez y faire référence en utilisant la syntaxe `plugin.pluginName.fixtureName` dans le tableau `$fixtures`.

Avant d'utiliser des fixtures assurez-vous que votre `phpunit.xml` contienne un listener (écouteur) pour les fixtures :

```
<!-- Configure un listener pour les fixtures -->
<listeners>
    <listener
    class="\Cake\TestSuite\Fixture\FixtureInjector"
    file="./vendor/cakephp/cakephp/src/TestSuite/Fixture/FixtureInjector.php">
        <arguments>
            <object class="\Cake\TestSuite\Fixture\FixtureManager" />
        </arguments>
    </listener>
</listeners>
```

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": {
        "MyPlugin\\Test\\": "./plugins/MyPlugin/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 le 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. Cell

`<name>` doit être le nom de l'objet dont vous voulez générer le squelette de tests.

Intégration avec Jenkins

Jenkins [171] est un serveur d'intégration continu, qui peut vous aider à automatiser l'exécution de vos cas de test. Cela aide à s'assurer que tous les tests passent et que votre application est déjà prête.

Intégrer une application CakePHP avec Jenkins est assez simple. Ce qui suit suppose que vous avez déjà installé Jenkins sur un système *nix, et que vous êtes capable de l'administrer. Vous savez aussi comment créer des jobs, et lancer des builds. Si vous n'êtes pas sur de tout cela, référez vous à la documentation de Jenkins [172].

171. http://jenkins-ci.org
172. http://jenkins-ci.org/

Créer un Job

Commençons par créer un job pour votre application, et connectons votre répertoire afin que jenkins puisse accéder à votre code.

Ajouter une Config de Base de Données de Test

Utiliser une base de données séparée juste pour Jenkins est généralement une bonne idée, puisque cela évite au sang de couler et évite un certain nombre de problèmes basiques. Une fois que vous avez créé une nouvelle base de données dans un serveur de base de données auquel jenkins peut accéder (habituellement localhost). Ajoutez une *étape de script shell* au build qui contient ce qui suit :

```
cat > config/app_local.php <<'CONFIG'
<?php
return [
    'Datasources' => [
        'test' => [
            'datasource' => 'Database/Mysql',
            'host'       => 'localhost',
            'database'   => 'jenkins_test',
            'username'   => 'jenkins',
            'password'   => 'cakephp_jenkins',
            'encoding'   => 'utf8'
        ]
    ]
];
CONFIG
```

Ensuite, décommentez la ligne suivante dans votre fichier **config/bootstrap.php** :

```
//Configure::load('app_local', 'default');
```

En créant un fichier **app_local.php**, vous avez un moyen facile de définir une configuration spécifique pour Jenkins. Vous pouvez utiliser ce même fichier de configuration pour remplacer tous les autres fichiers de configuration dont vous avez besoin sur Jenkins.

C'est souvent une bonne idée de supprimer et re-créer la base de données avant chaque build aussi. Cela vous évite des echecs de chaînes, où un build cassé entraîne l'echec des autres. Ajoutez une autre *étape de script shell* au build qui contient ce qui suit :

```
mysql -u jenkins -pcakephp_jenkins -e 'DROP DATABASE IF EXISTS jenkins_test; CREATE␣
↪DATABASE jenkins_test';
```

Ajouter vos Tests

Ajoutez une autre *étape de script shell* à votre build. Dans cette étape, lancez les tests pour votre application. Créer un fichier de log junit, ou clover coverage est souvent un bonus sympa, puisqu'il vous donne une vue graphique sympa des résultats de votre test :

```
# Télécharger Composer s'il est manquant.
test -f 'composer.phar' || curl -sS https://getcomposer.org/installer | php
# Installer les dépendances.
php composer.phar install
vendor/bin/phpunit --log-junit junit.xml --coverage-clover clover.xml
```

Si vous utilisez le clover coverage, ou les résultats junit, assurez-vous de les configurer aussi dans Jenkins. Ne pas configurer ces étapes signifiera que vous ne verrez pas les résultats.

Lancer un Build

Vous devriez être capable de lancer un build maintenant. Vérifiez la sortie de la console et faites tous les changements nécessaires pour obtenir le build précédent.

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 [173].

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')
    ->notEmpty('title', 'Please fill this field')
    ->add('title', [
        'length' => [
            'rule' => ['minLength', 10],
            'message' => 'Titles need to be at least 10 characters long',
        ]
    ])
    ->allowEmpty('published')
```

(suite sur la page suivante)

173. https://api.cakephp.org/3.x/class-Cake.Validation.Validation.html

(suite de la page précédente)

```
    ->add('published', 'boolean', [
        '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');
```

If you have multiple fields that are required, you can define them as a list :

```
// Define multiple fields for create
$validator->requirePresence(['author_id', 'title'], 'create');

// Define multiple fields for mixed modes
$validator->requirePresence([
    'author_id' => [
        'mode' => 'create',
        'message' => 'An author is required.',
    ],
    'published' => [
        'mode' => 'update',
        'message' => 'The published state is required.',
    ]
]);
```

Nouveau dans la version 3.3.0 : `requirePresence()` accepts an array of fields as of 3.3.0

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' => 'Comments must have a substantial body.'
        ],
        'maxLength' => [
            'rule' => ['maxLength', 250],
            'message' => 'Comments cannot be too long.'
        ]
    ]);
```

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\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.

Nouveau dans la version 3.5.0.

Vous pouvez utiliser le plugin Localized [174] 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');
        // 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 [175] :

```
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' => 'The title is not valid'
]);

// Utilise un tableau appelable qui n'est pas un provider
$validator->add('title', 'custom', [
    'rule' => [$this, 'method'],
    'message' => 'The title is not valid'
]);

// Utilise une closure
$extra = 'Some additional value needed inside the closure';
$validator->add('title', 'custom', [
```

(suite sur la page suivante)

174. https://github.com/cakephp/localized
175. https://github.com/cakephp/localized/blob/master/src/Validation/ValidationInterface.php

(suite de la page précédente)

```
    'rule' => function ($value, $context) use ($extra) {
        // Logique personnalisée qui retourne true/false
    },
    'message' => 'The title is not valid'
]);

// Utilisez une règle à partir d'un provider personnalisé
$validator->add('title', 'custom', [
    'rule' => 'customRule',
    'provider' => 'custom',
    'message' => 'The title is not unique enough'
]);
```

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.

Nouveau dans la version 3.1.1 : La possibilité de faire un callable pour `requirePresence()` a été ajoutée dans 3.1.1.

Imbriquer des Validators

Nouveau dans la version 3.0.5.

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' => '']
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ]
];
```

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();
```

(suite sur la page suivante)

(suite de la page précédente)

```
$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 `ORM\Table::newEntity()`, `ORM\Table::newEntities()`, `ORM\Table::patchEntity()`, `ORM\Table::patchEntities()` or *`ORM\Table::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 [176] 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.

176. https://api.cakephp.org/3.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

static `Cake\Core\Plugin::path`(*string $plugin*)

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",
        "": "./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))`.

`Collection\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`**(*callable $c*)

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`**(*callable $c*)

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($matcher)`

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'],
        ]
    ]
];

$numbers = (new Collection($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);
```

(suite sur la page suivante)

(suite de la page précédente)

```
$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 $c)`

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]]
]
```

Nouveau dans la version 3.4.0 : `chunkWithKeys()` a été ajoutée dans la version 3.4.0

Filtrer

`Cake\Collection\Collection::filter`(*callable $c*)

Les collections permettent de filtrer et de créer facilement 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(callable $c)`

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(callable $c)`

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(array $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(array $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(callable $c)`

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|callable $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(string|callable $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(string|callable $callback)`

Pour finir, la méthode `sumOf()` va retourner la somme d'une propriété de tous les éléments :

```
$collection = new Collection($people);
$sumOfAges =  $collection->sumOf('age');
```

(suite sur la page suivante)

(suite de la page précédente)

```
$sumOfChildrenAges = $collection->sumOf(function ($person) {
    return $person->child->age;
});

$sumOfDadAges = $collection->sumOf('dad.age');
```

`Cake\Collection\Collection::avg`(*$matcher = 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');
```

Nouveau dans la version 3.5.0.

`Cake\Collection\Collection::median`(*$matcher = 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');
```

Nouveau dans la version 3.5.0.

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($elements)`

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)`

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. Créer une structure imbriquée où les enfants sont groupés selon une propriété identifier parente est facile avec la méthode `nest()`.

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'],
```

(suite sur la page suivante)

(suite de la page précédente)

```
]);

$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`(*$dir = 'desc'*, *$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
[
    ['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();

$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'],
]
```

Nouveau dans la version 3.3.0 : `Collection::transpose()` a été ajoutée dans la version 3.3.0.

Retrait d'Eléments

`Cake\Collection\Collection::sample(int $size)`

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(int $size, int $from)`

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`(*int $positions*)

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` (*string $path*, *array|Traversable $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');
            }

            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`(*callable $c*)

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)
```

(suite sur la page suivante)

(suite de la page précédente)

```
        ->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;
```

(suite sur la page suivante)

(suite de la page précédente)

```
});

$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.

Nouveau dans la version 3.5.0 : Les Collections initialisées avec un tableau ne peuvent plus etre itérées de façon Lazy afin d'améliorer les performances.

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()
{
    ...
    foreach ($transientElements as $e) {
        yield $e;
    }
}
$rewindable = (new Collection(results()))->buffered();
```

Clonage de Collection

`Cake\Collection\Collection::compile`(*bool $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.

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 *.ctp* à l'intérieur de ce dossier en utilisant les regex :

```
$files = $dir->find('.*\.ctp');
```

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.');
    // $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 selon name.

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('/a/path/for', 'testing');
// $path égal /a/path/for/testing
```

$element peut aussi être un tableau :

```
$path = Folder::addPathElement('/a/path/for', ['testing', 'another']);
// $path égal à /a/path/for/testing/another
```

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('/non-existent-folder');
```

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('/path/to/folder');
$folder->copy([
    'to' => '/path/to/new/folder',
    'from' => '/path/to/copy/from', // Will cause a cd() to occur
    'mode' => 0755,
```

(suite sur la page suivante)

(suite de la page précédente)

```
    'skip' => ['skip-me.php', '.git'],
    'scheme' => Folder::SKIP  // Ne fait pas les répertoires/fichiers qui␣
↪existent déjà.
]);
```

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'))
↪{
    // Successfully created the nested folders
}
```

`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 bytes 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::normalizePath`(*string $path*)
Retourne un ensemble correct de slashes pour un $path donné. ("\" pour les chemins Windows et "/" pour les autres chemins.)

`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 le dernier temps d'accès.

`Cake\Filesystem\File::`**`lastChange`**()
Retourne le dernier temps modifié ou `false` en cas d'erreur.

`Cake\Filesystem\File::`**`md5`**(*integer|boolean $maxsize = 5*)
Récupère la MD5 Checksum du fichier avec la vérification précédente du Filesize 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 lignes de fin en un terminator 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 caractère 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*)
Rend le nom de fichier bon pour la sauvegarde.

`Cake\Filesystem\File::size`()
Retourne le Filesize en bytes.

`Cake\Filesystem\File::writable`()

Type renvoyé boolean

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] => [
```

(suite sur la page suivante)

(suite de la page précédente)

```
            [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]');
```

(suite sur la page suivante)

(suite de la page précédente)

```
/* $result ressemble maintenant à:
    [
        ['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
        ]
    ]
*/
```

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',
            'state' => 'TN',
            'something' => '{0}'
        ]
    ],
    [
        'Person' => [
            'first_name' => 'Garrett',
            'last_name' => 'Woodworth',
            'city' => 'Venice Beach',
            'state' => 'CA',
```

(suite sur la page suivante)

(suite de la page précédente)

```
            '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);
// 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');
```

(suite sur la page suivante)

(suite de la page précédente)

```
// $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);

/* $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 = [
```

(suite sur la page suivante)

(suite de la page précédente)

```
    [
        '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,
    '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*[, *array $n*])

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
    ]
    [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']];
```

(suite sur la page suivante)

(suite de la page précédente)

```
$result = Hash::dimensions($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,
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ]
*/
```

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']]
];
$result = Hash::sort($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']];
$res = Hash::mergeDiff($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] => [
```

(suite sur la page suivante)

(suite de la page précédente)

```
                        [0] => image_1_id
                        [1] => image_2_id
                ]
        ]
    ]
*/

$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]);
/* $result ressemble maintenant à:
    [
        (int) 0 => [
```

(suite sur la page suivante)

(suite de la page précédente)

```
            '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 mais puissant qui peut être utilisé pour faire des requêtes. C'est un bon moyen de communiquer avec des services webs et des APIs distantes.

Modifié dans la version 3.3.0 : Avant 3.3.0, vous devez utiliser `Cake\Network\Http\Client`.

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 également simple :

```
// Envoi d'une requête POST avec des données encodées 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 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(...);
```

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.

Avertissement : Pour des raisons de compatibilité, les chaînes commençant par `@` seront considérées comme locales ou des chemins de fichier d'un dépôt.

Cette fonctionnalité est dépréciée depuis CakePHP 3.0.5 et sera retirée dans une version future. Avant que cela n'arrive, les données d'utilisateur passées au Client Http devront être nettoyées comme suit :

```
$response = $http->post('http://example.com/api', [
    'search' => ltrim($this->request->getData('search'), '@'),
]);
```

S'il est nécessaire de garder les caractères du début `@` dans les chaînes de la requête, vous pouvez passer une chaîne de requête pré-encodée avec `http_build_query()` :

```
$response = $http->post('http://example.com/api', http_build_query([
    'search' => $this->request->getData('search'),
]));
```

Construire des Corps de Requête Multipart à la Main

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);
// Définit le type de contenu.
$xml->type('application/xml');
$data->add($xml);

// Création d'un fichier upload 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 REST API, 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 faîtes une requête GET qui a besoin des deux paramètres querystring et d'un corps de requête, vous pouvez faire comme ce qui suit :

```
// 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 l'Auth Basic 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 quand lors des vérifications des certificats SSL (non recommandé).
- `ssl_verify_depth` - par défaut à 5. Depth to traverse in the CA chain.
- `ssl_verify_host` - par défaut à `true`. Valide le certificat SSL pour un nom d'hôte.
- `ssl_cafile` - par défaut pour construire dans cafile. Ecrasez-le pour utiliser des bundles CA personnalisés.
- `timeout` - Durée d'attente avant le timing out en secondes.
- `type` - Envoi 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 faîtes des requêtes GET.
- `redirect` - Nombre de redirections à suivre. `false` par défaut.

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` intègre plusieurs systèmes d'authentification. Les différentes stratégies d'authentification peuvent être ajoutées par les développeurs. Les stratégies d'Authentification sont appelées avant que la requête ne soit envoyée, et permettent aux headers d'être ajoutés 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' => 'mark', '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' => 'mark',
    'password' => 'secret',
```

(suite sur la page suivante)

(suite de la page précédente)

```
    'realm' => 'myrealm',
    'nonce' => 'onetimevalue',
    'qop' => 1,
    'opaque' => 'someval'
  ]
]);
```

En configurant la clé "type" à "digest", vous dîtes 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 inclue 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' => 'bigkey',
    '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 entête. A 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 un 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' => 'mark',
    '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() [177].

Créer des 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 être moins pénible, vous pouvez créer des clients scoped :

```
// Création d'un client scoped.
$http = new Client([
  'host' => 'api.example.com',
  'scheme' => 'https',
  'auth' => ['username' => 'mark', 'password' => 'testing']
]);

// Faire une requête vers api.example.com
$response = $http->get('/test.php');
```

Les informations suivantes peuvent être utilisées lors de la création d'un client scoped :

- host
- scheme
- proxy
- auth
- port
- cookies
- timeout
- ssl_verify_peer
- ssl_verify_depth
- ssl_verify_host

Chacune de ces options peut être remplacées en les spécifiant quand vous faîtes des requêtes. host, scheme, proxy, port sont remplacées dans l'URL de la requête :

```
// Utilisation du client scoped que nous avons créé précédemment.
$response = $http->get('http://foo.com/test.php');
```

Ce qui est au-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 scoped 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 responses. Toute response avec des cookies, les verra stockés dans l'instance d'origine de Http\Client. Les cookies stockés dans une instance Client sont automatiquement inclus dans les futures requêtes vers le domaine + combinaisons de chemin qui correspondent :

```
$http = new Client([
    'host' => 'cakephp.org'
]);
```

(suite sur la page suivante)

177. http://php.net/manual/en/function.stream-context-create.php

(suite de la page précédente)

```
// Création d'une requête qui définit des cookies
$response = $http->get('/');

// Cookies à partir 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 :

```
// Personalisation d'un cookie existant.
$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'));
```

Nouveau dans la version 3.5.0 : `addCookie()` a été ajoutée dans 3.5.0

Objets Response

class `Cake\Http\Client\`**`Response`**

Les objets Response ont un certain nombre de méthodes pour parcourir les données de réponse.

Modifié dans la version 3.3.0 : Depuis la version 3.3.0 `Cake\Http\Client\Response` implémente PSR-7 ResponseInterface [178].

Lire des 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->body();

// En propriété
$response->body;
```

Vous pouvez aussi accéder à l'objet stream de la réponse et utilisez ses méthodes :

```
// Récupère une Psr\Http\Message\StreamInterface contenant le corps de la réponse
$stream = $response->getBody();

// Lit un stream de 100 bytes en une fois.
while (!$stream->eof()) {
```

(suite sur la page suivante)

178. http://www.php-fig.org/psr/psr-7/#3-3-psr-http-message-responseinterface

(suite de la page précédente)

```
    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 une utilisation facile d'accéder à la lecture des données décodées. Les données JSON dans un tableau, alors que les données XML sont décodées dans un arbre `SimpleXMLElement` :

```
// Récupération du XML.
$http = new Client();
$response = $http->get('http://example.com/test.xml');
$xml = $response->xml;

// Récupération du JSON.
$http = new Client();
$response = $http->get('http://example.com/test.json');
$json = $response->json;
```

Les données de réponse décodées sont stockées dans l'objet response, donc y accéder de nombreuses 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 avec des valeurs sensibles à la casse quand vous y accédez avec les méthodes :

```
// Récupère les en-têtes sous la forme d'un tableau associatif array.
$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 la réponse encodée
$response->getEncoding();

// Récupère un tableau de key=>value pour tous les en-têtes
$response->headers;
```

Accéder aux données de Cookie

Vous pouvez lire les cookies avec différentes méthodes selon le nombre de données que vous souhaitez sur les cookies :

```
// Récupère tous les cookies (toutes les données)
$response->getCookies();

// Récupère une valeur d'une unique cookie.
$response->getCookie('session_id');
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Récupère les données complètes pour un unique cookie
// includes value, expires, path, httponly, secure keys.
$response->getCookieData('session_id');

// Accède aux données complètes pour tous les cookies.
$response->cookies;
```

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();

// helper __get()
$response->code;
```

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 [179].

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 :

179. 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');
```

Créer des Chaînes d'URL Safe

static `Cake\Utility\Inflector::slug`(*$word*, *$replacement* = '-')

Slug convertit les caractères spéciaux en version latins et convertit les caractères ne correspondant pas et les espaces en tirets. La méthode slug s'attend à un encodage UTF-8 :

```
// apple-puree
Inflector::slug('apple purée');
```

Note : `Inflector::slug()` est dépréciée depuis la version 3.2.7. Utilisez `Text::slug()` à la place.

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()
    {
        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
echo $this->Number->toReadableSize(1321205.76); // 1.26 MB
```

(suite sur la page suivante)

(suite de la page précédente)

```
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'
]);
// Sortie '123 456,79 !'

// Appelé avec Number
```

(suite sur la page suivante)

(suite de la page précédente)

```
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.
$this->eventManager()->off($this->Auth);

// Re-active Auth pour les callbacks.
$this->eventManager()->on($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()
    {
        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])
            ));
        }
    }
}
```

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 [180] :

```
// 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' => '.']);
```

180. 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** [181]. 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
    ],
```

(suite sur la page suivante)

181. https://tools.ietf.org/html/rfc4122.html

(suite de la page précédente)

```
    'before' => '',
    '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,
```

(suite sur la page suivante)

(suite de la page précédente)

```
    '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\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,
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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\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);
```

(suite sur la page suivante)

(suite de la page précédente)

```
// 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**Time**

Si vous avez besoin de fonctionnalités `TimeHelper` en-dehors d'une `View`, utilisez la classe `Time` :

```
use Cake\I18n\Time;

class UsersController extends AppController
{

    public function initialize()
    {
        parent::initialize();
        $this->loadComponent('Auth');
    }

    public function afterLogin()
    {
        $time = new Time($this->Auth->user('date_of_birth'));
        if ($time->isToday()) {
            // accueillir l'utilisateur avec un message de bon anniversaire
            $this->Flash->success(__('Bon anniversaire à toi...'));
        }
    }
}
```

En interne, CakePHP utilise Chronos [182] pour faire fonctionner l'utilitaire `Time`. Tout ce que vous pouvez faire avec `Chronos` et `DateTime`, vous pouvez le faire avec `Time` et `Date`.

Note : Avant 3.2.0, CakePHP utilisait Carbon [183].

182. https://github.com/cakephp/chronos
183. https://github.com/briannesbitt/Carbon

Pour plus d'informations sur Chronos, rendez-vous sur la documentation de l'API [184].

Créer des Instances Time

Il y a plusieurs façons de créer des instances `Time` :

```
use Cake\I18n\Time;

// Crée à partir d'une chaîne datetime.
$time = Time::createFromFormat(
    'Y-m-d H:i:s',
    $datetime,
    'America/New_York'
);

// Crée à partir d'un timestamp
$time = Time::createFromTimestamp($ts);

// Récupère le temps actuel.
$time = Time::now();

// Ou utilise juste 'new'
$time = new Time('2014-01-10 11:11', 'America/New_York');

$time = new Time('2 hours ago');
```

Le constructeur de la classe `Time` peut prendre les mêmes paramètres que la classe PHP interne `DateTime`. 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.
$now = new Time('2014-04-12 12:22:30');
Time::setTestNow($now);

// Retourne '2014-04-12 12:22:30'
$now = Time::now();

// Retourne '2014-04-12 12:22:30'
$now = Time::parse('now');
```

Manipulation

Une fois créées, vous pouvez manipuler les instances `Time` en utilisant les méthodes setter :

```
$now = Time::now();
$now->year(2013)
    ->month(10)
    ->day(31);
```

Vous pouvez aussi utiliser les méthodes fournies nativement par la classe PHP `DateTime` :

184. https://api.cakephp.org/chronos/1.0/

```
$now->setDate(2013, 10, 31);
```

Les dates peuvent être modifiées à travers la soustraction et l'addition de leurs composantes :

```
$now = Time::now();
$now->subDays(5);
$now->addMonth(1);

// Utilisation des chaînes strtotime.
$now->modify('+5 days');
```

Vous pouvez obtenir des composantes internes d'une date en accédant à ses propriétés :

```
$now = Time::now();
echo $now->year; // 2014
echo $now->month; // 5
echo $now->day; // 10
echo $now->timezone; // America/New_York
```

Il est aussi permis d'assigner directement ces propriétés pour modifier la date :

```
$time->year = 2015;
$time->timezone = 'Europe/Paris';
```

Formatage

static Cake\I18n\Time::**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');  // For any mutable DateTime
FrozenTime::setJsonEncodeFormat('yyyy-MM-dd HH:mm:ss');  // For any immutable DateTime
Date::setJsonEncodeFormat('yyyy-MM-dd HH:mm:ss');  // For any mutable Date
FrozenDate::setJsonEncodeFormat('yyyy-MM-dd HH:mm:ss');  // For any immutable Date
```

Note : Cette méthode doit être appelée statiquement.

Cake\I18n\Time::**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 :

```
$now = Time::parse('2014-10-31');

// Affiche un stamp datetime localisé.
echo $now;

// Affiche '10/31/14, 12:00 AM' pour la locale en-US
$now->i18nFormat();

// Utilise la date complète et le format time
$now->i18nFormat(\IntlDateFormatter::FULL);

// Utilise la date complète mais un format court de temps
```

(suite sur la page suivante)

(suite de la page précédente)

```
$now->i18nFormat([\IntlDateFormatter::FULL, \IntlDateFormatter::SHORT]);

// affiche '2014-10-31 00:00:00'
$now->i18nFormat('yyyy-MM-dd HH:mm:ss');
```

Il est possible de spécifier le format d'affichage désiré. Vous pouvez soit passer une constante IntlDateFormatter [185] ou une chaine 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 :

```
// Affiche 'Friday, Aban 9, 1393 AP at 12:00:00 AM GMT'
$result = $now->i18nFormat(\IntlDateFormatter::FULL, null, 'en-IR@calendar=persian');
```

Les types de calendrier suivants sont supportés :

- japanese
- buddhist
- chinese
- persian
- indian
- islamic
- hebrew
- coptic
- ethiopic

Nouveau dans la version 3.1 : Le support des calendriers non-grégoriens a été ajouté dans 3.1

Note : Pour les chaînes constantes, par exemple pour IntlDateFormatter : :FULL, Intl utilise la librairie ICU qui alimente ses données à partir de CLDR (http ://cldr.unicode.org/) dont la version peut varier selon l'installation PHP et donner des résultats différents.

`Cake\I18n\Time::nice()`

Affiche un format prédéfini "nice" :

```
$now = Time::parse('2014-10-31');

// Affiche 'Oct 31, 2014 12:32pm' en en-US
echo $now->nice();
```

Vous pouvez modifier le timezone avec lequel la date est affichée sans modifier l'objet `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 :

```
$now->i18nFormat(\IntlDateFormatter::FULL, 'Europe/Paris');
```

Laisser le premier paramètre à `null` va utiliser la chaine de formatage par défaut :

```
$now->i18nFormat(null, 'Europe/Paris');
```

Enfin, il est possible d'utiliser une locale différente pour l'affichage d'une date :

185. http://www.php.net/manual/en/class.intldateformatter.php

```
echo $now->i18nFormat(\IntlDateFormatter::FULL, 'Europe/Paris', 'fr-FR');

echo $now->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 [186]. Vous pouvez cependant modifier ceci par défaut à la volée :

```
Time::setDefaultLocale('es-ES'); // For any mutable DateTime
FrozenTime::setDefaultLocale('es-ES'); // For any immutable DateTime
Date::setDefaultLocale('es-ES'); // For any mutable Date
FrozenDate::setDefaultLocale('es-ES'); // For any immutable Date
```

A partir de maintenant, les datetimes vont s'afficher avec un format de préférence 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); // For any mutable DateTime
FrozenTime::setToStringFormat(\IntlDateFormatter::SHORT); // For any immutable␣
↪DateTime
Date::setToStringFormat(\IntlDateFormatter::SHORT); // For any mutable Date
FrozenDate::setToStringFormat(\IntlDateFormatter::SHORT); // For any immutable Date

// La même méthode existe pour les Date, FrozenDate, et FrozenTime
Time::setToStringFormat([
    \IntlDateFormatter::FULL,
    \IntlDateFormatter::SHORT
]);

// La même méthode existe pour les Date, FrozenDate, et FrozenTime
Time::setToStringFormat('yyyy-MM-dd HH:mm:ss');
```

Il est recommandé de toujours utiliser les constantes plutôt que de directement passer une date en format chaîne de caractère.

Formater les Temps Relatifs

`Cake\I18n\Time::`**`timeAgoInWords`**(*array $options = []*)

Souvent, il est utile d'afficher les temps liés au présent :

```
$now = new Time('Aug 22, 2011');
echo $now->timeAgoInWords(
    ['format' => 'MMM d, YYY', 'end' => '+1 year']
);
// On Nov 10th, 2011 this would display: 2 months, 2 weeks, 6 days ago
```

L'option `end` vous laisse définir à partir de quel point les temps relatifs doivent être formatés en utilisant l'option `format`. L'option `accuracy` nous laisse contrôler le niveau de détail qui devra être utilisé pour chaque intervalle :

186. http://www.php.net/manual/en/intl.configuration.php#ini.intl.default-locale

```
// Si $timestamp est 1 month, 1 week, 5 days et 6 hours ago
echo $timestamp->timeAgoInWords([
    'accuracy' => ['month' => 'month'],
    'end' => '1 year'
]);
// Affiche '1 month ago'
```

En configurant `accuracy` en une chaîne, vous pouvez spécifier le niveau maximum de détail que vous souhaitez afficher :

```
$time = new Time('+23 hours');
// Affiche 'in about a day'
$result = $time->timeAgoInWords([
    'accuracy' => 'day'
]);
```

Conversion

`Cake\I18n\Time::toQuarter()`

Une fois créées, vous pouvez convertir les instances `Time` en timestamps ou valeurs quarter :

```
$time = new Time('2014-06-15');
$time->toQuarter();
$time->toUnixString();
```

Comparer Avec le Present

`Cake\I18n\Time::isYesterday()`

`Cake\I18n\Time::isThisWeek()`

`Cake\I18n\Time::isThisMonth()`

`Cake\I18n\Time::isThisYear()`

Vous pouvez comparer une instance `Time` avec le temps présent de plusieurs façons :

```
$time = new Time('2014-06-15');

echo $time->isYesterday();
echo $time->isThisWeek();
echo $time->isThisMonth();
echo $time->isThisYear();
```

Chacune des méthodes ci-dessus va retourner `true`/`false` selon si oui ou non l'instance `Time` correspond au temps présent.

Comparer Avec les Intervals

`Cake\I18n\Time::isWithinNext($interval)`

Vous pouvez regarder si une instance `Time` tombe dans un intervalle en utilisant `wasWithinLast()` et `isWithinNext()` :

```
$time = new Time('2014-06-15');

// A moins de 2 jours.
echo $time->isWithinNext(2);

// A moins de 2 semaines.
echo $time->isWithinNext('2 weeks');
```

`Cake\I18n\Time::wasWithinLast($interval)`

Vous pouvez aussi comparer une instance `Time` dans un intervalle dans le passé :

```
// Dans les 2 derniers jours.
echo $time->wasWithinLast(2);

// Dans les 2 dernières semaines.
echo $time->wasWithinLast('2 weeks');
```

Dates

Nouveau dans la version 3.2.

La classe `Date` dans CakePHP implémente les mêmes API et méthodes que *`Cake\I18n\Time`*. La différence principale entre `Time` et `Date` est que `Date` ne suit pas les composants liés à l'heure et est toujours en UTC. Par exemple :

```
use Cake\I18n\Date;
$date = new Date('2015-06-15');

$date->modify('+2 hours');
// Affiche 2015-06-15 00:00:00
echo $date->format('Y-m-d H:i:s');

$date->modify('+36 hours');
// Affiche 2015-06-15 00:00:00
echo $date->format('Y-m-d H:i:s');
```

Les tentatives de modification de timezone sur une instance de `Date` seront toujours ignorées :

```
use Cake\I18n\Date;
$date = new Date('2015-06-15');
$date->setTimezone(new \DateTimeZone('America/New_York'));

// Affiche UTC
echo $date->format('e');
```

Dates et Heures Immutables

class Cake\I18n**FrozenTime**

class Cake\I18n**FrozenDate**

CakePHP offre des classes de date et d'heure immutables qui implémentent la même interface que leurs équivalents mutables. Les objets immutables sont utiles pour éviter les modifications accidentelles de données, ou lorsque vous voulez éviter les problèmes liés à l'ordre de dépendances. 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);

// La sorie ici est inconnue.
echo $time->format('Y-m-d H:i:s');
```

Si les appels aux méthodes sont réordonnés, ou si `someOtherFunction` évolue la sortie peut être inattendue. La mutabilité de vos objets crée un couplage temporal. Si nous utilisions des objets immutables, 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 immutables sont utiles dans les entities car elles évitent les modifications accidentelles, et forcent les modifications à être explicitement exprimées. Utiliser des objets immutables 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 Requêtées 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*.

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 [187] et SimpleXML [188] 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

`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)

187. http://php.net/domdocument
188. http://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>
```

(suite sur la page suivante)

(suite de la page précédente)

```
<?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 $default = null*)
Nouveau dans la version 3.1.1 : Le paramètre `$default` a été ajouté.

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 [189].

189. https://book.cakephp.org/chronos/1.x/fr/

CHAPITRE 43

Debug Kit

Cette page a été déplacée [190].

190. https://book.cakephp.org/debugkit/3.x/fr/

CHAPITRE 44

Migrations

Cette page a été déplacée [191].

191. https://book.cakephp.org/migrations/2.x/fr/

CHAPITRE 45

ElasticSearch

Cette page a été déplacée [192].

192. https://book.cakephp.org/elasticsearch/2.x/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.

3.x Guide de Migration

3.x Guide de Migration

Les guides de migration contiennent des informations sur les nouvelles fonctionnalités introduites dans chaque version et le chemin de migration entre les versions 2.x et 3.x. Si vous utilisez actuellement la version 1.x, vous devrez d'abord mettre à jour vers la version 2.x. Consultez la documentation de la version 2.x pour avoir les guides de mise à jour qui s'y rapportent.

3.6 Guide de Migration

3.6 Migration Guide

CakePHP 3.6 est une mise à jour de CakePHP 3.5 dont la compatibilité API est complète. Cette page souligne les changements et améliorations faits dans 3.6.

Pour mettre à jour vers 3.6.x, lancez la commande suivante :

```
php composer.phar require --update-with-dependencies "cakephp/cakephp:3.6.*"
```

Deprecations

La liste qui suit regroupe les méthodes, les propriétés et les comportements dépréciés. Ces différents éléments continueront de fonctionner jusqu'à la version 4.0.0, à partir de laquelle ils seront supprimés.

— `bin/cake orm_cache` est maintenant `bin/cake schema_cache`.

Changement de Comportements

Les changements suivants sont compatibles avec l'API, mais elles représentent des écarts mineurs de comportement qui pourrait affecter votre application :

— `Cake\Utility\Security::randomBytes()` lancera maintenant une exception lorsqu'une source sécurisée d'entropie ne peut pas être trouvée en PHP5.

— Les tokens générés par `SecurityComponent` incluent maintenant l'id de session de l'utilisateur, pour éviter que le token soit réutilisé entre des utilisateurs/sessions. Cela change la valeur des tokens de sécurité et va faire que les formulaires créés dans les versions précédentes de CakePHP ne vont pas passer la validation dans la version 3.6.

— `Cake\Database\Query::page()` lance maintenant des exceptions quand les valeurs de la page sont < à 1.

— Pagination permet maintenant de trier selon plusieurs champs à travers toutes les pages. Avant, seule la première page pouvait être triée avec plus d'une colonne. De plus, les conditions de tri définies dans la query string sont maintenant *préfixées* aux paramètres de tri par défaut plutôt que remplaçant complètement le tri par défaut.

— Les classes de Shell vont maintenant lancer des exceptions quand les classes de task ne sont pas trouvées. Avant, les tasks non valides étaient ignorées en silence.

— Le code interne de CakePHP chaîne maintenant les exceptions quand c'est possible, ce qui permet d'afficher les premières causes d'erreur.

Core

— `getTypeName()` a été ajoutée pour vous aider à récupérer le nom de la classe ou le type de variable, ce qui vous permettra de construire des messages d'erreur plus explicites.

ORM

— `EntityTrait::isEmpty()` et `EntityTrait::hasValue()` ont été ajoutées.

Shell

— La commande `cake assets copy` dispose maintenant d'une option `--overwrite` pour écraser les assets de plugin si ils existent déjà dans le webroot de l'application.

Validation

- `Validation::compareFields()` a été ajoutée en tant que version plus flexible que la fonction `Validation::compareWith()`.
- `Validator::notSameAs()` a été ajoutée pour vérifier facilement si un champ n'est pas identique à un autre champ.

3.5 Guide de Migration

3.5 Guide de Migration

CakePHP 3.5 est une mise à jour de CakePHP 3.4 dont la compatibilité API est complète. Cette page souligne les changements et améliorations faits dans 3.5.

Pour mettre à jour vers 3.5.x, lancez la commande suivante :

```
php composer.phar require --update-with-dependencies "cakephp/cakephp:3.5.*"
```

Dépréciations

La liste qui suit regroupe les méthodes, les propriétés et les comportements dépréciés. Ces différents éléments continueront de fonctionner jusqu'à la version 4.0.0, à partir de laquelle ils seront supprimés.

- `Cake\Http\Client\CookieCollection` est dépréciée. Utilisez `Cake\Http\Cookie\CookieCollection` à la place.
- `Cake\View\Helper\RssHelper` est dépréciée. Vu son peu d'utilisation, le RssHelper est déprécié.
- `Cake\Controller\Component\CsrfComponent` est dépréciée. Utilisez le csrf-middleware à la place.
- `Cake\Datasource\TableSchemaInterface` est dépréciée. Utilisez `Cake\Database\TableSchemaAwareInterface` à la place.
- `Cake\Console\ShellDispatcher` est dépréciée. Vous devez mettre à jour vos Applications pour qu'elles utilisent `Cake\Console\CommandRunner` à la place.
- `Cake\Database\Schema\TableSchema::column()` est dépréciée. Utilisez `Cake\Database\Schema\TableSchema::getColumn()` à la place.
- `Cake\Database\Schema\TableSchema::constraint()` est dépréciée. Utilisez `Cake\Database\Schema\TableSchema::getConstraint()` à la place.
- `Cake\Database\Schema\TableSchema::index()` est dépréciée. Utilisez `Cake\Database\Schema\TableSchema::getIndex()` à la place.

Dépréciation des Méthodes Get / Set combinées

Dans le passé, CakePHP a utilisé des méthodes "modal" qui proposaient à la fois un mode get et un mode set. Ces méthodes compliquent l'auto-complétion des IDE et notre capacité à ajouter des *return type* stricts dans le futur. Pour ces raisons, ces méthodes get / set combinées sont scindées en méthodes get et set séparées.

La liste suivante regroupe les méthodes dépréciées et qu'il faudra remplacer par des méthodes `getX()` et `setX()` :

`Cake\Cache\Cache`
- `config()`
- `registry()`

`Cake\Console\Shell`
- `io()`

`Cake\Console\ConsoleIo`

— `outputAs()`

Cake\Console\ConsoleOutput
— `outputAs()`

Cake\Database\Connection
— `logger()`

Cake\Database\TypedResultInterface
— `returnType()`

Cake\Database\TypedResultTrait
— `returnType()`

Cake\Database\Log\LoggingStatement
— `logger()`

Cake\Datasource\ModelAwareTrait
— `modelType()`

Cake\Database\Query
— la partie « getter » de `valueBinder()` (maintenant `getValueBinder()`)

Cake\Database\Schema\TableSchema
— `columnType()`

Cake\Datasource\QueryTrait
— la partie « getter » de `eagerLoaded()` (maintenant `isEagerLoaded()`)
— `eagerLoaded()` (maintenant `isEagerLoaded()`)

Cake\Event\EventDispatcherInterface
— `eventManager()`

Cake\Event\EventDispatcherTrait
— `eventManager()`

Cake\Error\Debugger
— `outputAs()` (maintenant `getOutputFormat()` / `setOutputFormat()`)

Cake\Http\ServerRequest
— `env()` (maintenant `getEnv()` / `withEnv()`)

Cake\I18n\I18n
— `locale()`
— `translator()`
— `defaultLocale()`
— `defaultFormatter()`

Cake\ORM\Association\BelongsToMany
— `sort()`

Cake\ORM\LocatorAwareTrait
— `tableLocator()`

Cake\ORM\EntityTrait
— `invalid()` (maintenant `getInvalid()`, `setInvalid()`, `setInvalidField()`, maintenant `getInvalidField()`)

Cake\ORM\Table
— `validator()`

Cake\Routing\RouteBuilder
— `extensions()`
— `routeClass()`

Cake\Routing\RouteCollection
— `extensions()`

Cake\TestSuite\TestFixture

— `schema()`

`Cake\Utility\Security`
— `salt()`

`Cake\View\View`
— `template()`
— `layout()`
— `theme()`
— `templatePath()`
— `layoutPath()`
— `autoLayout()` (maintenant `isAutoLayoutEnabled()` / `enableAutoLayout()`)

Changement de Comportements

Bien que ces changements garde la compatibilité API, ce sont tout de même des variations mineures qui pourraient avoir un impact sur votre application.

— `BehaviorRegistry`, `HelperRegistry` et `ComponentRegistry` lanceront maintenant une exception quand `unload()` est appelé avec un nom d'objet inconnu. Ce changement devrait aider à trouver plus rapidement les erreurs.
— Les associations `HasMany` gèrent maintenant mieux les valeurs "vides" pour les propriétés d'associations, de la même manière que `BelongsToMany` : elles traitent maintenant `false`, `null` et les chaînes vides comme des tableaux vides. Pour les associations `HasMany`, cela veut dire que les résultats des enregistrements associés sont maintenant supprimés / déliés quand la stratégie de sauvegarde `replace` est utilisée. Ce qui veut dire vous pouvez maintenant supprimer / déliés des enregistrements associés en passant une chaîne vide : vous auriez dû utiliser une logique spécifique de marshalling pour faire cela auparavant.
— `ORM\Table::newEntity()` marque maintenant *dirty* les propriétés correspondant à des associations si les enregistrements de l'association *marshallé* sont *dirty*. Dans les cas où une entity d'association est créée et qu'elle ne contient aucune propriété, elle ne sera pas marquée pour être persistée.
— `Http\Client` n'utilise plus le résultat de la méthode `cookie()` lors de la construction de requêtes. A la place, le header `Cookie` et la `CookieCollection` interne sont utilisés. Cela ne devrait affecter que les applications qui ont des adapter HTTP personnalisés dans leurs clients.
— Les sous-commandes composées de plusieurs mots devaient impérativement être appelées avec leur nom camelBacked pour être utilisées. Les sous-commandes peuvent maintenant être appelées via leur nom au format *underscored_names*. Par exemple : `cake tool initMyDb` peut maintenant être appelée via `cake tool init_my_db`. Si vos Shells liaient 2 sous-commandes avec le même nom mais 2 inflections différentes, seule la dernière commande liée fonctionnera.
— `SecurityComponent` bloquera les requête POST qui ne passent pas de données dans la requête (pas de *request data*). Cela aide à la protection des actions qui créent des enregistrements en base en utilisant seulement les *defaults* de la base de données.
— `Cake\ORM\Table::addBehavior()` et `removeBehavior()` retournent maintenant `$this` pour faciliter la définition d'objets Table avec une interface fluide.
— Les moteurs de Cache ne lancent maintenant plus d'exception quand ils échouent ou qu'ils sont mals configurés. Ils se rabattent, à la place, sur le moteur *noop* `NullEngine`. Les *fallbacks* peuvent aussi être configurés par moteur.

Nouvelles Fonctionnalités

Middleware sur Scopes

Les middlewares peuvent maintenant être appliqués conditionnellement à des routes sur des URL "scopées". Cela vous permet de construire des stacks de middlewares spécifiques pour différentes parties de votre application sans avoir à faire des tests sur l'URL dans le code de vos middlewares. Plus d'informations dans la section *Connecter des Middlewares à un scope*.

Nouveau Lanceur de Console

3.5.0 ajoute `Cake\Console\CommandRunner`. Cette classe, avec `Cake\Console\CommandCollection`, intègre l'environnement CLI dans la nouvelle classe `Application`. Les classes `Application` peuvent maintenant implémenter un hook `console()` qui permet d'avoir un contrôle complet sur les commandes CLI exposées, comment elles sont nommées et comment les shells récupèrent leurs dépendances. Adopter cette nouvelle classe nécessite que vous remplaciez le contenu de votre fichier `bin/cake.php` par le fichier suivant [193].

Fallbacks pour les Moteurs de Cache

Les moteurs de cache peuvent maintenant être configurés avec une clé `fallback` qui permet de définir une configuration de cache sur laquelle se rabattre si le moteur était mal configuré ou indisponible. Reportez-vous à la section *Configurer un Fallback de Cache* pour plus d'informations sur la configuration de "fallbacks" pour vos configurations de cache.

Support de dotenv au squelette d'Application

Le squelette d'application possède maintenant une intégration « dotenv », facilitant l'utilisation de variables d'environnement pour configurer votre application. Référez-vous à la section *Variables d'Environnement* pour plus d'informations.

Console Integration Testing

La classe `Cake\TestSuite\ConsoleIntegrationTestCase` a été ajoutée pour faciliter les tests d'intégration des applications console. Pour plus d'informations rendez-vous à la section *Tests d'Intégration de la Console*. Cette classe de test est compatible avec le dispatcher de shell actuel mais aussi avec le nouveau `Cake\Console\CommandRunner`.

Collection

— `Cake\Collection\Collection::avg()` a été ajoutée.
— `Cake\Collection\Collection::median()` a été ajoutée.

193. https://github.com/cakephp/app/tree/3.next/bin/cake.php

Core

— `Cake\Core\ObjectRegistry` implémente maintenant les interfaces `Countable` et `IteratorAggregate`.

Console

— `Cake\Console\ConsoleOptionParser::setHelpAlias()` a été ajoutée. Cette méthode permet de définir le nom de la commande à utiliser lors de l'affichage des aides. Par défaut, la valeur est `cake`
— `Cake\Console\CommandRunnner` a été ajoutée en remplacement de `Cake\Console\ShellDispatcher`.
— `Cake\Console\CommandCollection` a été ajouté afin de fournir une interface pour que les applications puissent définir les outils en ligne de commande qu'elles offrent.

Database

— Une nouvelle option `mask` pour le driver SQLite a été ajoutée. Cette option vous permet de définir les droits sur le fichier de la base SQLite quand il est créé.

Datasource

— `Cake\Datasource\SchemaInterface` a été ajoutée.
— De nouveaux types abstraits ont été définis pour `smallinteger` et `tinyinteger`. Les colonnes existantes en `SMALLINT` and `TINYINT` seront maintenant introspectées via ces nouveaux types abstraits. Les colonnes `TINYINT(1)` continueront à être traitées comme des booléen dans MySQL.
— `Cake\Datasource\PaginatorInterface` a été ajoutée. Le `PaginatorComponent` utilise maintenant cette interface pour intéragir avec les paginators. Cela permet à des implémentations *ORM-like* d'être paginées par le component.
— `Cake\Datasource\Paginator` a été ajouté pour paginer les instances des requêtes ORM/Database.

Event

— Les méthodes `Cake\Event\EventManager::on()` et `off()` peuvent maintenant être chainées ce qui rend plus simple la définition de plusieurs événements à la fois.

Http

— Les classes `Cookie` & `CookieCollection` ont été ajoutées. Ces classes vous permettent de travailler avec les cookies de manière orientée objet et sont disponibles dans `Cake\Http\ServerRequest`, `Cake\Http\Response`, et `Cake\Http\Client\Response`. Référez-vous aux sections *Cookies* et *Définir des Cookies* pour plus d'informations.
— Un nouveau middleware a été ajouté pour permettre d'appliquer des headers de sécurité plus facilement. Référez-vous à security-header-middleware pour plus d'informations.
— Un nouveau middleware a été ajouté pour chiffrer de manière transparente les données de cookie. Référez-vous à encrypted-cookie-middleware pour plus d'informations.
— Un nouveau middleware a été ajouté pour permettre une protection CSRF plus simple. Référez-vous à csrf-middleware pour plus d'informations.
— `Cake\Http\Client::addCookie()` a été ajoutée pour faciliter l'ajoute de cookies à une instance d'un client.

InstanceConfigTrait

— `InstanceConfigTrait::getConfig()` accepte maintenant un second paramètre `$default`. Si aucune valeur n'est disponible pour la `$key` spécifiée, la valeur de `$default` sera retournée.

ORM

— `Cake\ORM\Query::contain()` vous permet de l'appeler sans le tableau quand vous faites un contain() sur une seule association. `contain('Comments', function () { ... });` fonctionnera maintenant. Cela rend `contain()` plus cohérent avec d'autres méthodes d'eagerloading comme `leftJoinWith()` et `matching()`.

Routing

— `Cake\Routing\Router::reverseToArray()` a été ajoutée. Cette méthode vous permet de convertir des objets de requête en tableau qui peuvent être utilisés pour générer des URL sous forme de chaîne.
— `Cake\Routing\RouteBuilder::resources()` s'est vue ajouter une option `path`. Cette option vous permet de faire en sorte que le chemin de la ressource et le nom du controller ne correspondent pas.
— `Cake\Routing\RouteBuilder` a maintenant des méthodes pour créer des routes spécifiques à des méthodes HTTP comme `get()` et `post()` par exemple.
— `Cake\Routing\RouteBuilder::loadPlugin()` a été ajoutée.
— `Cake\Routing\Route` a maintenant des méthodes « fluide » (*fluent interface*) pour définir ses options.

TestSuite

— `TestCase::loadFixtures()` chargera maintenant toutes les fixtures si aucun argument n'est passé.
— `IntegrationTestCase::head()` a été ajoutée.
— `IntegrationTestCase::options()` a été ajoutée.
— `IntegrationTestCase::disableErrorHandlerMiddleware()` a été ajoutée pour faciliter le debugging des tests d'intégration.

Validation

— `Cake\Validation\Validator::regex()` a été ajoutée afin de permettre de faire de la validation par regex plus facilement.
— `Cake\Validation\Validator::addDefaultProvider()` a été ajoutée. Cette méthode vous permet d'injecter des providers de validation dans tous les validators créés dans votre application.
— `Cake\Validation\ValidatorAwareInterface` a été ajouté pour définir les méthodes implémentées par `Cake\Validation\ValidatorAwareTrait`.

View

— `Cake\View\Helper\PaginatorHelper::limitControl()` a été ajoutée. Cette méthode vous permet de créer un formulaire avec un select qui permet de mettre à jour la valeur « limite » d'un résultat paginé.

3.4 Guide de Migration

3.4 Guide de Migration

CakePHP 3.4 est une mise à jour de CakePHP 3.3 dont la compatibilité API est complète. Cette page souligne les changements et améliorations faits dans 3.4.

PHP 5.6 devient le minimum requis

CakePHP 3.4 a maintenant besoin d'au minimum PHP 5.6.0 puisque PHP 5.5 n'est plus supporté et ne recevra plus de correctifs de sécurité.

Dépréciations

La liste qui suit regroupe les méthodes, les propriétés et les comportements dépréciés. Ces différents éléments continueront de fonctionner jusqu'à la version 4.0.0, à partir de laquelle ils seront supprimés.

Dépréciations sur Request & Response

La majorité des dépréciations de la version 3.4 se trouve sur les objets `Request` et `Response`. Les méthodes existantes qui modifient directement les objets sont maintenant dépréciées et remplacées par les méthodes qui suivent le pattern des « objets immutables » décrit dans le standard PSR-7.

Plusieurs propriétés de `Cake\Network\Request` ont été dépréciées :

— `Request::$params` est dépréciée. Utilisez `Request::getAttribute('params')` à la place.
— `Request::$data` est dépréciée. Utilisez `Request::getData()` à la place.
— `Request::$query` est dépréciée. Utilisez `Request::getQueryParams()` à la place.
— `Request::$cookies` est dépréciée. Utilisez `Request::getCookie()` à la place.
— `Request::$base` est dépréciée. Utilisez `Request::getAttribute('base')` à la place.
— `Request::$webroot` est dépréciée. Utilisez `Request::getAttribute('webroot')` à la place.
— `Request::$here` est dépréciée. Utilisez `Request::getRequestTarget()` à la place.
— `Request::$_session` a été renommée `Request::$session`.

Certaines méthodes de `Cake\Network\Request` ont été dépréciées :

— Les méthodes `__get()` & `__isset()` sont dépréciées. Utilisez `getParam()` à la place.
— `method()` est dépréciée. Utilisez plutôt `getMethod()`.
— `setInput()` est dépréciée. Utilisez plutôt `withBody()`.
— Les méthodes `ArrayAccess` ont toutes été dépréciées.
— `Request::param()` est dépréciée. Utilisez plutôt `Request::getParam()` .
— `Request::data()` est dépréciée. Utilisez plutôt `Request::getData()`.
— `Request::query()` est dépréciée. Utilisez plutôt `Request::getQuery()`.
— `Request::cookie()` est dépréciée. Utilisez plutôt `Request::getCookie()`.

Plusieurs méthodes de `Cake\Network\Response` ont été dépréciées soit parce qu'elles faisaient doublon avec les méthodes PSR-7, soit parce qu'elles ont été rendues obsolètes du fait de l'introduction du stack PSR-7 :

— `Response::header()` est dépréciée. Utilisez plutôt `getHeaderLine()`, `hasHeader()` ou `Response::getHeader()`.

— `Response::body()` est dépréciée. Utilisez plutôt `Response::withBody()`.
— `Response::statusCode()` est dépréciée. Utilisez plutôt `Response::getStatusCode()`.
— `Response::httpCodes()` Cette méthode ne devrait plus être utilisée. CakePHP supporte maintenant tous les statuts standards recommandés.
— `Response::protocol()` est dépréciée. Utilisez plutôt `Response::getProtocolVersion()`.
— `send()`, `sendHeaders()`, `_sendHeader()`, `_sendContent()`, `_setCookies()`, `_setContentType()`, et `stop()` sont dépréciées et rendues obsolètes par le stack HTTP PSR-7.

Du fait que les `responses` sont maintenant des objets immutables du fait des recommandations du standard PSR-7, de nombreuses méthodes « helper » de l'objet `Response` ont été dépréciées. Leurs variantes immutables sont maintenant recommandées :

— `Response::location()` devient `Response::withLocation()`
— `Response::disableCache()` devient `Response::withDisabledCache()`
— `Response::type()` devient `Response::withType()`
— `Response::charset()` devient `Response::withCharset()`
— `Response::cache()` devient `Response::withCache()`
— `Response::modified()` devient `Response::withModified()`
— `Response::expires()` devient `Response::withExpires()`
— `Response::sharable()` devient `Response::withSharable()`
— `Response::maxAge()` devient `Response::withMaxAge()`
— `Response::vary()` devient `Response::withVary()`
— `Response::etag()` devient `Response::withEtag()`
— `Response::compress()` devient `Response::withCompression()`
— `Response::length()` devient `Response::withLength()`
— `Response::mustRevalidate()` devient `Response::withMustRevalidate()`
— `Response::notModified()` devient `Response::withNotModified()`
— `Response::cookie()` devient `Response::withCookie()`
— `Response::file()` devient `Response::withFile()`
— `Response::download()` devient `Response::withDownload()`

Veuillez vous référez à la section *Adopter les Responses Immutable* pour plus d'informations avant de mettre à jour votre code car utiliser les méthodes immutables va demander plus de changements que le simple remplacement des méthodes.

Autres dépréciations

— Les propriétés _public_ de `Cake\Event\Event` sont dépréciées, de nouvelles méthodes ont été ajoutées pour lire et écrire ces propriétés.
— `Event::name()` est dépréciée. Utilisez `Event::getName()` à la place.
— `Event::subject()` est dépréciée. Utilisez `Event::getSubject()` à la place.
— `Event::result()` est dépréciée. Utilisez `Event::getResult()` à la place.
— `Event::data()` est dépréciée. Utilisez `Event::getData()` à la place.
— La valeur de `Auth.redirect` stockée en session n'est plus utilisée. Un paramètre d'URL est maintenant utilisé pour stocker l'URL de redirection. Ceci retire cependant la possibilité de définir une URL de redirection en session en dehors des scénarios de login.
— `AuthComponent` ne stocke plus les URLs de redirection quand l'URL non autorisée n'est pas une action `GET`.
— L'option `ajaxLogin` du `AuthComponent` est dépréciée. Vous devez maintenant utiliser le code de statut HTTP `403` pour déclencher le bon comportement côté client.
— La méthode `beforeRedirect` du `RequestHandlerComponent` est dépréciée.
— Le code de statut HTTP `306` de `Cake\Network\Response` est dépréciée. Sa phrase de statut est maintenant "Unused" car ce code de statut n'est pas standard.
— `Cake\Database\Schema\Table` a été renommée en `Cake\Database\Schema\TableSchema`. Le nom précédent portait à confusion pour de nombreux utilisateurs.

- L'option `fieldList` pour `Cake\ORM\Table::newEntity()` et `patchEntity()` a été renommée en `fields` pour être plus cohérent avec les autres parties de l'ORM.
- `Router::parse()` est dépréciée. `Router::parseRequest()` est maintenant la méthode recommandée car elle accepte une request en argument et donne plus de contrôle et de flexibilité dans la manipulation des requêtes entrantes.
- `Route::parse()` est dépréciée. `Route::parseRequest()` est maintenant la méthode recommandée car elle accepte une request en argument et donne plus de contrôle et de flexibilité dans la manipulation des requêtes entrantes.
- `FormHelper::input()` est dépréciée. Utilisez `FormHelper::control()` à la place.
- `FormHelper::inputs()` est dépréciée. Utilisez `FormHelper::controls()` à la place.
- `FormHelper::allInputs()` est dépréciée. Utilisez `FormHelper::allControls()` à la place.
- `Mailer::layout()` est dépréciée. Utilisez `Mailer::setLayout()` exposée par `Mailer::__call()` à la place.

Dépréciation des getters / setters combinés

Par le passé, CakePHP a exposé des méthodes combinées qui opéraient à la fois comme getter et comme setter. Ces méthodes compliquaient l'auto-complétion de certains IDE et auraient compliqué la mise en place de typage strictes sur les retours des méthodes dans le futur. Pour ces raisons, les getters / setters combinés sont maintenant séparés dans différentes méthodes.

La liste qui suit regroupe les méthodes qui sont dépréciées et remplacées par des méthodes `getX()` et `setX()` :

`Cake\Core\InstanceConfigTrait`
- `config()`

`Cake\Core\StaticConfigTrait`
- `config()`
- `dsnClassMap()`

CakeConsoleConsoleOptionParse
- `command()`
- `description()`
- `epliog()`

CakeDatabaseConnection
- `driver()`
- `schemaCollection()`
- `useSavePoints()` (devenue `enableSavePoints()` / `isSavePointsEnabled()`)

CakeDatabaseDriver
- `autoQuoting` (devenue `enableAutoQuoting()` / `isAutoQuotingEnabled()`)

CakeDatabaseExpressionFunctionExpression
- `name()`

CakeDatabaseExpressionQueryExpression
- `tieWith()` (devenue `setConjunction()` / `getConjunction()`)

CakeDatabaseExpressionValuesExpression
- `columns()`
- `values()`
- `query()`

CakeDatabaseQuery
- `connection()`
- `selectTypeMap()`
- `bufferResults()` (devenue `enableBufferedResults()` / `isBufferedResultsEnabled()`)

CakeDatabaseSchemaCachedCollection
- `cacheMetadata()`

CakeDatabaseSchemaTableSchema
- `options()`
- `temporary()` (devenue `setTemporary()` / `isTemporary()`)

CakeDatabaseTypeMap
- `defaults()`
- `types()`

CakeDatabaseTypeMapTrait
- `typeMap()`
- `defaultTypes()`

`Cake\ORM\Association`
- `name()`
- `cascadeCallbacks()`
- `source()`
- `target()`
- `conditions()`
- `bindingKey()`
- `foreignKey()`
- `dependent()`
- `joinType()`
- `property()`
- `strategy()`
- `finder()`

`Cake\ORM\Association\BelongsToMany`
- `targetForeignKey()`
- `saveStrategy()`
- `conditions()`

`Cake\ORM\Association\HasMany`
- `saveStrategy()`
- `foreignKey()`
- `sort()`

`Cake\ORM\Association\HasOne`
- `foreignKey()`

CakeORMEagerLoadable
- `config()`
- setter part of `canBeJoined()` (devenue `setCanBeJoined()`)

CakeORMEagerLoader
- `matching()` (`getMatching()` devra être appelée après `setMatching()` pour conserver l'ancien comportement)
- `autoFields()` (devenue `enableAutoFields()` / `isAutoFieldsEnabled()`)

CakeORMLocatorTableLocator
- `config()`

CakeORMQuery
- `eagerLoader()`
- `hydrate()` (now `enableHydration()`/`isHydrationEnabled()`)
- `autoFields()` (now `enableAutoFields()`/`isAutoFieldsEnabled()`)

CakeORMTable
- `table()`
- `alias()`
- `registryAlias()`
- `connection()`
- `schema()`

— `primaryKey()`
— `displayField()`
— `entityClass()`

CakeMailerEmail
— `from()`
— `sender()`
— `replyTo()`
— `readReceipt()`
— `returnPath()`
— `to()`
— `cc()`
— `bcc()`
— `charset()`
— `headerCharset()`
— `emailPattern()`
— `subject()`
— `template()` (devenue `setTemplate()` / `getTemplate()` et `setLayout()` / `getLayout()`)
— `viewRender()` (devenue `setViewRenderer()` / `getViewRenderer()`)
— `viewVars()`
— `theme()`
— `helpers()`
— `emailFormat()`
— `transport()`
— `messageId()`
— `domain()`
— `attachments()`
— `configTransport()`
— `profile()`

CakeValidationValidator
— `provider()`

CakeViewStringTemplateTrait
— `templates()`

CakeViewViewBuilder
— `templatePath()`
— `layoutPath()`
— `plugin()`
— `helpers()`
— `theme()`
— `template()`
— `layout()`
— `options()`
— `name()`
— `className()`
— `autoLayout()` (devenue `enableAutoLayout()` / `isAutoLayoutEnabled()`)

Adopter les Responses Immutable

Avant de migrer votre code pour qu'il utilise les nouvelles méthodes de l'objet Response, sachez que les nouvelles méthodes sont bâties sur un concept différent. Les objets immutables sont généralement indiquées par le préfixe `with` (par exemple : `withLocation()`). Du fait que ces méthodes évoluent dans un contexte immutable, elles retournent de nouvelles instances que vous devez assigner à des variables ou des propriétés. Partons du principe que vous aviez du code de Controller similaire à celui-ci :

```
$response = $this->response;
$response->location('/login')
$response->header('X-something', 'a value');
```

Si vous faites un simple « rechercher / remplacer » pour changer le nom des méthodes, cela ne fonctionnera pas. Vous devriez plutôt remplacer votre code pour qu'il ressemble à ceci :

```
$this->response = $this->response
    ->withLocation('/login')
    ->withHeader('X-something', 'a value');
```

Voici les points clés à comprendre :

1. Le résultat de vos changements doit être ré-assigné à `$this->response`. C'est le point le plus important pour conserver le fonctionnement souhaité.
2. Les méthodes « setter » peuvent être chaînées. Cela vous permet d'éviter de stocker tous les états intermédiaires.

Astuces pour Migrer vos Components

Dans les versions précédentes de CakePHP, les Components possédaient souvent des références aux objets Request et Response pour pouvoir les modifier. Avant d'utiliser les méthodes immutable, vous devriez utiliser les Response attachées au Controller :

```
// Dans une méthode de Component (or callback)
$this->response->header('X-Rate-Limit', $this->remaining);

// Deviendrait
$controller = $this->getController();
$controller->response = $controller->response->withHeader('X-Rate-Limit', $this->
↪remaining);
```

Dans les callbacks des Components, vous pouvez utiliser l'objet Event pour accéder à la Response / au Controller :

```
public function beforeRender($event)
{
    $controller = $event->getSubject();
    $controller->response = $controller->response->withHeader('X-Teapot', 1);
}
```

Astuce : Plutôt que conserver une référence aux Responses, récupérez plutôt la Response actuelle stockée dans le Controller et modifiez la propriété `response` quand vous avez terminé vos modifications.

Changement de comportements

Bien que ces changements garde la compatibilité API, ce sont tout de même des variations mineures qui pourraient avoir un impact sur votre application :

- Les résultats de `ORM\Query` ne feront plus de typecast sur les alias de colonnes basé sur le type de colonne original. Par exemple, si vous faites un alias de `created` en `created_time`, vous obtiendrez maintenant une instance de `Time` plutôt qu'une chaîne de caractères.
- Le `AuthComponent` utilise maintenant un paramètre URL pour stocker l'adresse de redirection quand un utilisateur non identifié est redirigé sur la page de connexion. Auparavant, l'URL de redirection était stockée en session. Utiliser un paramètre d'URL permet une meilleure compatibilité avec les différents navigateurs.
- Le système de *reflection* de base de données traite maintenant les types de colonnes inconnus comme `string` et non plus comme `text`. L'impact de ce changement est notamment visible sur le `FormHelper` qui va générer des inputs à la place de textarea pour les types de colonnes inconnus.
- `AuthComponent` ne va plus stocker ses messages Flash via la clé "auth". Ils seront maintenant rendu avec le template "error" et sous la clé flash "default". Ceci a été fait dans le but de simplifier `AuthComponent`.
- `Mailer\Email` va maintenant automatiquement détecter les types de contenus des pièces jointes en utilisant `mime_content_type` si le « content-type » n'est pas spécifié. Auparavant, les pièces jointes étaient considérées comme "application/octet-stream" par défaut.
- CakePHP utilise maintenant l'opérateur `...` à la place de `call_user_func_array()`. Si vous passez des tableaux associatifs, vous devez mettre à jour votre code pour passer des tableaux à index numériques en utilisant la méthode `array_values()` pour les méthodes suivantes :
  - `Cake\Mailer\Mailer::send()`
  - `Cake\Controller\Controller::setAction()`
  - `Cake\Http\ServerRequest::is()`

Visibility Changes

- `MailerAwareTrait::getMailer()` est maintenant `protected`.
- `CellTrait::cell()` est maintenant `protected`.

Si les traits ci-dessus sont utilisés dans vos controllers, leurs méthodes publiques pouvaient être appelées par les règles de routing par défaut en tant qu'actions. Ces changements permettent d'apporter une sécurité à vos controllers. Si vous avez besoin que ces méthodes conservent une visibilité `public`, vous aurez besoin de mettre à jour les instructions `use` comme ceci :

```
use CellTrait {
    cell as public;
}
use MailerAwareTrait {
    getMailer as public;
}
```

Collection

- `CollectionInterface::chunkWithKeys()` a été ajoutée. Les implémentations de `CollectionInterface` des utilisateurs devront maintenant implémenter cette méthode.
- `Collection::chunkWithKeys()` a été ajoutée.

Erreur

— `Debugger::setOutputMask()` et `Debugger::outputMask()` ont été ajoutées. Ces méthodes vous permettent de configurer des propriétés / clés de tableau qui devraient être masquées lors d'affichages générés par le `Debugger` (lors d'un appel à `debug()` par exemple).

Event

— `Event::getName()` a été ajoutée.
— `Event::getSubject()` a été ajoutée.
— `Event::getData()` a été ajoutée.
— `Event::setData()` a été ajoutée.
— `Event::getResult()` a été ajoutée.
— `Event::setResult()` a été ajoutée.

I18n

— Vous pouvez maintenant personnaliser le comportement du loader de messages de fallback. Reportez-vous à *Créer des Traducteurs Génériques* pour plus d'information.

Routing

— `RouteBuilder::prefix()` accepte maintenant un tableau de paramètres par défaut à ajouter à chaque route « connectée ».
— Les routes peuvent maintenant être « matché » sur des hosts spécifiques à l'aide de l'option `_host`.

Email

— `Email::setPriority()`/`Email::getPriority()` ont été ajoutées.

HtmlHelper

— `HtmlHelper::scriptBlock()` n'englobe plus le Javascript dans un tag `<![CDATA[ ]]` par défaut. L'option `safe` qui contrôle ce comportement a maintenant sa valeur par défaut à `false`. Utiliser le tag `<![CDATA[ ]]` était seulement requis pour le XHTML qui n'est plus le doctype prédominant pour les pages web actuellement.

BreadcrumbsHelper

— `BreadcrumbsHelper::reset()` a été ajoutée. Cette méthode vous permet de supprimer les éléments déjà présents.

PaginatorHelper

- `PaginatorHelper::numbers()` utilise maintenant une ellipse HTML au lieu de "..." dans les templates par défaut.
- `PaginatorHelper::total()` a été ajoutée et permet de lire le nombre total de pages pour le résultat de requête actuellement paginé.
- `PaginatorHelper::generateUrlParams()` a été ajoutée et est utilisée comme méthode de construction d'URL « bas niveau ».
- `PaginatorHelper::meta()` peut maintenant créer des liens pour "first" et "last".

FormHelper

- Vous pouvez maintenant configurer les sources à partir desquelles FormHelper lit. Ceci simplifie la création des formulaires GET. Consultez *Récupérer les valeurs du formulaire depuis la query string* pour plus d'informations.
- `FormHelper::control()` a été ajoutée.
- `FormHelper::controls()` a été ajoutée.
- `FormHelper::allControls()` a été ajoutée.

Validation

- `Validation::falsey()` et `Validation::truthy()` ont été ajoutées.

TranslateBehavior

- `TranslateBehavior::translationField()` a été ajoutée.

PluginShell

- `cake plugin load` et `cake plugin unload` supportent maintenant une option `--cli` qui permet de mettre à jour `bootstrap_cli.php` à la place de `bootstrap.php`.

TestSuite

- Le support de `PHPUnit 6` a été ajouté. Puisque cette version du framework a au minimum besoin de PHP 5.6.0, les versions supportées de PHPUnit sont maintenant `^5.7|^6.0`

3.3 Guide de Migration

3.3 Guide de Migration

CakePHP 3.3 est une mise à jour de CakePHP 3.2 dont la compatibilité API est complète. Cette page souligne les changements et améliorations faits dans 3.3.

Deprecations

- `Router::mapResources()` est dépréciée. Utilisez les scopes de routing et `$routes->resources()` à la place.
- `Router::redirect()` est dépréciée. Utilisez les scopes de routing et `$routes->redirect()` à la place.
- `Router::parseNamedParams()` est dépréciée. La rétro-compatibilité des paramètres nommés sera retirée dans la version 4.0.0.
- `Cake\Http\Client\Response` a vu ses méthodes suivantes dépréciées car elles se chevauchent avec les méthodes de l'interface PSR-7 :
  - `statusCode()` utilisez `getStatusCode()` à la place.
  - `encoding()` utilisez `getEncoding()` à la place.
  - `header()` utilisez `getHeaderLine()` à la place.
  - `cookie()` utilisez `getCookie()` à la place.
  - `version()` utilisez `getProtocolVersion()` à la place.
- Les Filtres de Dispatcher sont maintenant dépréciés. Utilisez /controllers/middleware à la place.
- `RequestActionTrait` a été dépréciée. Refactorez votre code pour utiliser *View Cells* à la place.
- Le moteur `Cake\\Utility\\Crypto\\Mcrypt` a été déprécié puisque l'extension `mcrypt` est dépréciée dans PHP 7.1. Utilisez `openssl` et `Cake\Utility\Crypto\Openssl` à la place.

Changements de Comportement

Bien que ces changements soient compatibles avec l'API, ils entraînent des variations mineures qui peuvent avoir des effets sur votre application :

- L'encodage du format JSON par défaut pour les instances Date et DateTime est maintenant ISO-8601. Cela signifie que la valeur timezone contient un `:`. Par exemple `2015-11-06T00:00:00+03:00`
- `Controller::referer()` enlève maintenant le chemin de base de l'application de façon cohérente lors de la génération des URLs en local. Avant, les chaînes d'URLs étaient préfixées par le chemin de base alors que les tableaux d'URLs ne l'étaient pas.
- Le `ErrorController` par défaut ne désactive plus les components `Auth` et `Security`, puisqu'il n'étend plus `AppController`. Si vous activez ces components avec des events, vous devrez mettre à jour votre code.
- `Entity::clean` now cleans original values, clearing them on save. This behavior was a bug as the entity's original state should not be retained after a save, but instead reflect the new state of the entity.

Support du Middleware PSR-7 Ajouté

En même temps qu'avec la dépréciation des Filters du Dispatcher, le support pour le middleware PSR-7 a été ajouté. Middleware est une partie de la nouvelle stack HTTP qui est un component au choix de CakePHP 3.3.0. En utilisant la nouvelle stack HTTP, vous pouvez tirer profit des fonctionnalités comme :

- Utilisation du middleware à partir des plugins, et des libraries en-dehors de CakePHP.
- Amène les mêmes méthodes pour l'objet response à la fois pour les réponses que vous obtenez à partir de `Http\Client` et les réponses que votre application génère.
- Être capable d'augmenter les objets response émis par la gestion des erreurs et la délivrance des assets.

Consultez le chapitre /controllers/middleware et les sections adding-http-stack pour plus d'informations sur la façon d'ajouter la nouvelle stack HTTP à une application existante.

Http Client est maintenant compatible avec PSR-7

`Cake\Network\Http\Client` a été déplacée vers `Cake\Http\Client`. Ses objet request et response implémentent maintenant les interfaces PSR-7 [194]. Plusieurs méthodes de `Cake\Http\Client\Response` sont maintenant dépréciées, regardez plus haut pour plus d'informations.

Améliorations de l'ORM

- Le support supplémentaire a été ajouté pour mapper des types de données complexes. Cela facilite le travail pour des types geo-spatiaux et les données qui ne peuvent pas être représentées par des chaînes dans des requêtes SQL. Consultez *Faire correspondre des types de données personnalisés aux expressions SQL* pour plus d'informations.
- Un nouveau `JsonType` a été ajouté. Ce nouveau type vous permet d'utiliser les types natifs JSON disponibles avec MySQL et Postgres. Dans les autres providers de bases de données, le type `json` mappera vers des colonnes `TEXT`.
- `Association::unique()` a été ajoutée. Cette méthode est un proxy pour la méthode `unique()` de la table, mais permet de s'assurer que les conditions des associations soient appliquées.
- Les règles `isUnique` s'appliquent maintenant pour les conditions des associations.
- Quand les entities sont converties en JSON, les objets associés ne sont plus d'abord convertis en tableau avec `toArray()`. A la place, la méthode `jsonSerialize()` sera appelée sur toutes les entities associées. Ceci vous donne plus de flexibilité et de contrôle sur les propriétés à exposer dans les représentations JSON de vos entities.
- `Table::newEntity()` et `Table::patchEntity()` vont maintenant lever une exception quand une association inconnue est dans la clé "associated".
- `RulesChecker::validCount()` a été ajoutée. Cette nouvelle méthode permet d'ajouter des règles qui vérifient le nombre d'enregistrements associés d'une entity.
- L'option `allowNullableNulls` a été ajoutée à la règle `existsIn`. Cette option permet aux règles de passer quand des colonnes sont nulles.
- Sauvegarder des enregistrements traduits est maintenant plus simple. Consultez la section *Sauvegarder Plusieurs Traductions* pour plus d'informations.

Support pour la Pagination Multiple Ajouté

Vous pouvez maintenant paginer plusieurs requêtes dans une action de controller/template de vue. Consultez la section *Requêtes de Paginating Multiple* pour plus de détails.

Shell Cache Ajouté

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
```

194. http://www.php-fig.org/psr/psr-7/

FormHelper

— FormHelper va maintenant automatiquement définir la valeur par défaut des champs avec la valeur par défaut définie dans vos colonnes de base de données. Vous pouvez désactiver ce comportement en définissant l'option `schemaDefault` à false.

Validation

— `Validator::requirePresence()`, `Validator::allowEmpty()` et `Validator::notEmpty()` acceptent maintenant une liste de champs. Ceci vous permet de définir de façon plus concise les champs qui sont requis.

StringTemplate

`StringTemplate::format()` lève maintenant une exception au lieu de retourner `null` quand un template demandé n'est pas trouvé.

Autres Améliorations

— `Collection::transpose()` a été ajoutée. Cette méthode vous permet de transposer les lignes et colonnes d'une matrice avec des colonnes de longueurs égales.
— Le `ErrorController` par défaut charge maintenant `RequestHandlerComponent` pour activer l'entête `Accept` selon le type de contenu pour les pages d'erreur.

Routing

— `Router::parse()`, `RouteCollection::parse()` et `Route::parse()` ont un nouvel argument `$method`. Il est par défaut à "GET". Ce nouveau paramètre réduit le recours à l'état global, et est nécessaire pour le travail d'intégration de la norme PSR-7.
— Quand vous construisez vos resource routes, vous pouvez maintenant définir un préfixe. C'est utile quand vous définissez des ressources imbriquées car vous pouvez créer des controllers spécialisés pour les ressources imbriquées.
— Les Filtres de Dispatcher sont maintenant dépréciés. Utilisez /controllers/middleware à la place.

Console

— Les Shell tasks qui sont appelées directement à partir du CLI n'appellent plus la méthode `_welcome`. Ils vont maintenant aussi avoir le paramètre `requested` défini.
— `Shell::err()` va maintenant appliquer le style "error" au texte. Le style par défaut est le texte rouge.

Request

— `Request::is()` et `Request::addDetector()` supportent maintenant des arguments supplémentaires dans les détecteurs. Cela permet aux détecteurs callables d'opérer sur des paramètres supplémentaires.

Debugging Functions

— Les fonctions `pr()`, `debug()` et `pj()` retournent maintenant la valeur résultante. Cela facilite leur utilisation quand des valeurs sont retournées.
— *`dd()`* a été ajoutée pour complètement arrêter l'exécution.

3.2 Guide de Migration

3.2 Guide de Migration

CakePHP 3.2 est une mise à jour de CakePHP 3.1 dont la compatibilité API est complète. Cette page souligne les changements et améliorations faits dans 3.2.

PHP 5.5 Requis au Minimum

CakePHP 3.2 requiert au moins PHP 5.5.9. En adoptant PHP 5.5, nous pouvons fournir des librairies de Date et de Time et retirer les dépendances qui concernent les librairies de compatibilité pour les mots de passe.

Dépréciations

Pendant que nous continuons à améliorer CakePHP, certaines fonctionnalités sont dépréciées puisqu'elles seront remplacées par de meilleures solutions. Les fonctionnalités dépréciées ne seront pas retirées jusqu'à 4.0 :

— `Shell::error()` est dépréciée car son nom n'indique pas clairement qu'il affiche un message et stoppe l'exécution. Utilisez `Shell::abort()` à la place.
— `Cake\Database\Expression\QueryExpression::type()` est dépréciée. Utilisez `tieWith()` à la place.
— `Cake\Database\Type\DateTimeType::$dateTimeClass` est dépréciée. Utilisez DateTimeType : :useMutable() ou DateTimeType : :useImmutable() à la place.
— `Cake\Database\Type\DateType::$dateTimeClass` est dépréciée. Utilisez `DateTimeType::useMutable()` ou `DateType::useImmutable()` à la place.
— `Cake\ORM\ResultSet::_calculateTypeMap()` n'est maintenant plus utilisée et est dépréciée.
— `Cake\ORM\ResultSet::_castValues()` n'est maintenant plus utilisée et est dépréciée.
— La clé `action` pour `FormHelper::create()` a été dépréciée. Vous devez utiliser la clé `url` directement.

Désactiver les Avertissements Liés à l'Obsolescence

Après la mise à niveau vous pouvez rencontrer plusieurs avertissements liés à l'obsolescence. Ces avertissements sont émis par des méthodes, options et fonctionnalités qui seront supprimées dans CakePHP 4.x, mais vont continuer à exister tout au long du cycle de vie de 3.x. Bien que nous vous recommandons de régler les problèmes liés à l'obsolescence au fur et à mesure que vous les rencontrez, ce n'est pas toujours possible. Si vous voulez reporter la correction de ces avertissements, vous pouvez les désactiver dans votre **config/app.php** :

```
'Error' => [
    'errorLevel' => E_ALL & ~E_DEPRECATED & ~E_USER_DEPRECATED,
]
```

Le niveau d'erreur ci-dessus va supprimer les avertissements liés à l'obsolescence de CakePHP.

Nouvelles Améliorations

Carbon Remplacé par Chronos

La librairie Carbon a été remplacée par *cakephp/chronos*. Cette nouvelle librairie est un fork de Carbon sans aucune dépendance supplémentaire. Elle offre également un objet date calendaire, et une version immutable des objets date et datetime.

Nouvel Objet Date

La classe `Date` vous permet de mapper proprement les colonnes `DATE` vers des objets PHP. Les instances de Date définiront toujours leur heure à `00:00:00 UTC`. par défaut, l'ORM crée maintenant des instances de `Date` lorsqu'il mappe des colonnes `DATE`.

Nouveaux Objets Immutables Date et Time

Les classes `FrozenTime` et `FrozenDate` ont été ajoutées. Ces classes offrent la même API que l'objet `Time`. Les classes « frozen » (gelées) fournissent des variantes immutables de `Time` et `Date`. En utilisant les objets immutables, vous pouvez éviter les mutations accidentelles. Au lieu de modifications directes, les méthodes de modification renvoient de *nouvelles* instances :

```
use Cake\I18n\FrozenTime;

$time = new FrozenTime('2016-01-01 12:23:32');
$newTime = $time->modify('+1 day');
```

Dans le code ci-dessus, `$time` et `$newTime` sont des objets différents. L'objet `$time` garde sa valeur originale alors que `$newTime` contient la valeur modifiée. Pour plus d'informations, référez-vous à la section sur les *Temps Immutables*. A partir de 3.2, l'ORM peut mapper les colonnes date/datetime vers des objets immutables. Regardez la section *Activer les Objets DateTime Immutables* pour plus d'informations.

CorsBuilder Ajouté

Afin de faciliter la définition des en-têtes liés aux Requêtes de type Cross-site (Cross Origin Requests = CORS), un nouveau `CorsBuilder` a été ajouté. Cette classe vous laisse définir les en-têtes liés au CORS avec une interface simple. Consultez *Définir les En-têtes de Requête d'Origine Croisée (Cross Origin Request Headers = CORS)* pour plus d'informations.

RedirectRoute lance une Exception en cas de Redirect

`Router::redirect()` lance maintenant une `Cake\Network\Routing\RedirectException` quand une condition de redirect est atteinte. Cette exception est récupérée par le filtre de routing et convertie en une réponse. Ceci remplace les appels à `response->send()` et permet aux filtres du dispatcher d'intéragir avec les réponses du redirect.

Améliorations de l'ORM

- Faire un contain avec la même association plusieurs fois fonctionne maintenant de la façon espérée, et les fonctions du constructeur de requête sont maintenant empilées.
- Les expression de fonctions transforment maintenant correctement leurs résultats dans le type attendu. Ceci signifie que les expressions comme `$query->func()->current_date()` vont retourner des instances de datetime.
- La donnée du champ qui échoue pendant la validation peut maintenant être accessible dans les entities avec la méthode `invalid()`.
- Les recherches avec la méthode d'accesseur de l'entity sont maintenant mises en cache et ont une meilleur performance.

API du Validator Amélioré

L'objet Validator a quelques nouvelles méthodes qui rendent la construction des validateurs moins verbeux. Par exemple, ajouter les règles de validation pour un champ de nom d'utilisateur peut maintenant ressembler à ceci :

```
$validator->email('username')
    ->ascii('username')
    ->lengthBetween('username', [4, 8]);
```

Améliorations de la Console

- `Shell::info()`, `Shell::warn()` et `Shell::success()` ont été ajoutées. Ces méthodes de helper facilitent l'utilisation des styles communément utilisés.
- `Cake\Console\Exception\StopException` ont été ajoutées.
- `Shell::abort()` a été ajoutée pour remplacer `error()`.

StopException Ajoutée

`Shell::_stop()` et `Shell::error()` n'appellent plus `exit()`. A la place, elles lancent une `Cake\Console\Exception\StopException`. Si vos shells/tasks attrapent les `\Exception` là où sont lancées ces méthodes, vous devrez mettre à jour ces blocs de code pour qu'ils n'attrapent pas les `StopException`. En évitant d'utiliser `exit()`, tester vos shells sera plus facile et nécessitera moins de mocks.

Helper initialize() ajouté

Les helpers peuvent maintenant avoir une méthode hook `initialize(array $config)` comme tous les autres types de classe.

Manipulation de la Limite de la Mémoire en cas d'Erreur Fatale

Une nouvelle option de configuration `Error.extraFatalErrorMemory` peut être définie en nombre de mega-octets, pour augmenter la limite de mémoire en cas d'erreur fatale. Cela permet d'allouer un petit espace mémoire supplémentaire pour la journalisation (logging) ainsi que la gestion d'erreur.

Étapes de Migration

Mettre à jour setToStringFormat()

Avant CakePHP 3.2, Time : :setToStringFormat() fonctionnait aussi avec des Objets Date. Après la mise à jour, vous devrez ajouter Date : :setToStringFormat() en plus pour voir de nouveau la Date formatée.

3.1 Guide de Migration

3.1 Guide de Migration

CakePHP 3.1 est une mise à jour de CakePHP 3.0 dont la compatibilité API est complète. Cette page souligne les changements et améliorations faits dans 3.1.

Routing

- La classe de route par défaut a été changée en `DashedRoute` dans le dépôt `cakephp/app`. Votre base de code actuelle n'est pas affectée par ceci mais il est recommandé d'utiliser cette classe de route à partir de maintenant.
- Les options de nom de préfixe ont été ajoutées aux différentes méthodes de construction de route. Regardez la section *Utiliser les Routes Nommées* pour plus d'informations.

Console

- `Shell::dispatchShell()` n'affiche plus le message d'accueil à partir du shell dispatché.
- La fonction `breakpoint()` a été ajoutée. Cette fonction fournit un snippet de code qui peut être utilisé dans un `eval()` pour lancer une console interactive. C'est très utile pour debugger les tests ou tout script CLI.
- Les options de console `--verbose` et `--quiet` controllent maintenant les niveaux de log de stdout/stderr.

Ajout des Shell Helpers

— Les applications de console peuvent maintenant créer des classes Helper qui encapsulent des blocs réutilisables de logique de sortie. Consultez la section sur *Shell Helpers* pour plus d'informations.

RoutesShell

— RoutesShell a été ajouté et vous fournit maintenant un moyen simple pour utiliser l'interface CLI pour tester et débugger les routes. Consultez la section *Shell Routes* pour plus d'informations.

Controller

— Les propriétés de Controller suivantes sont maintenant dépréciées :
  — layout
  — view - remplacée par `template`
  — theme
  — autoLayout
  — viewPath - remplacée par `templatePath`
  — viewClass - remplacée par `className`
  — layoutPath

  Au lieu de définir ces propriétés dans vos controllers, vous devrez les définir dans votre View en utlisant des méthodes de même nom :

```
// Dans un controller au lieu de
$this->layout = 'advanced'

// Vous devez utiliser
$this->viewBuilder()->layout('advanced');
```

Ces méthodes doivent être appelées après que vous ayez déterminé quelle classe de View sera utilisée par le controller/action.

AuthComponent

— Une nouvelle option de configuration `storage` a été ajoutée. Elle contient le nom de la classe de stockage que `AuthComponent` utilise pour stocker l'enregistrement de l'utilisateur. Par défaut `SessionStorage` est utilisée. Si vous utilisez un authentificateur stateless, vous devez configurer `AuthComponent` avec `MemoryStorage` à la place.
— Une nouvelle option de config `checkAuthIn` a été ajoutée. Elle contient le nom de l'event pour lequel les vérifications d'auth doivent être faites. Par défaut `Controller.startup` est utilisé, mais vous pouvez la définir dans `Controller.initialize` si vous souhaitez que l'authentification soit vérifiée avant que la méthode `beforeFilter()` de votre controller ne soit executée.
— Les options `scope` et `contain` de la classe d'authentification sont dépréciées. A la place, utilisez la nouvelle option `finder` pour configurer une méthode de finder personnalisée pour modifier le requête utilisée pour chercher l'utilisateur.
— La logique responsable de définir la variable de session `Auth.redirect`, qui est utilisée pour récupérer l'URL de redirection après la connexion (login) a été modifiée. Elle est maintenant définie uniquement lorsque l'on essaye d'accéder directement à une URL protégée sans authentification. Ainsi `Auth::redirectUrl()` renvoie l'URL protégée après le login. Dans la plupart des cas, lorsqu'un utilisateur accède directement à la page de connexion, `Auth::redirectUrl()` renvoie la valeur définie par la configuration de `loginRedirect`.

FlashComponent

— `FlashComponent` empile maintenant les messages enregistrés avec les méthodes `set()` et `__call()`. Cela signifie que la structure des données stockées dans la Session pour les messages Flash a changé.

CsrfComponent

— Le temps d'expiration du cookie CSRF peut maintenant être défini en une valeur compatible avec `strtotime()`.
— Les tokens CSRF invalides vont maintenant lancer une `Cake\Network\Exception\InvalidCsrfTokenException` plutôt qu'une `Cake\Network\Exception\ForbiddenException`.

RequestHandlerComponent

— `RequestHandlerComponent` échange maintenant le layout et le template selon l'extension parsée ou l'en-tête `Accept` dans le callback `beforeRender()` plutôt que dans `startup()`.
— `addInputType()` et `viewClassMap()` sont dépréciées. Vous devez utiliser `config()` pour modifier ces données de configuration à la volée.
— Quand `inputTypeMap` ou `viewClassMap` sont définies dans les configurations du component, elles vont *surcharger* les valeurs par défaut. Ce changement rend possible la suppression de la configuration par défaut.

Network

HttpClient

— Le type mime utilisé pour envoyer les requêtes par défaut a changé. Précédemment, `multipart/form-data` était toujours utilisé. Dans 3.1, `multipart/form-data` n'est utilisé qu'en cas de transfert de fichiers. Lorsqu'il n'y en pas, `application/x-www-form-urlencoded` est utilisé à la place.

ORM

Vous pouvez maintenant *Charger en Eager des Associations*. Cette fonctionnalité vous permet de charger des associations conditionnellement dans un ensemble de résultats, une entity ou une collection d'entites.

Les méthodes `patchEntity()` et `newEntity()` supportent maintenant l'option `onlyIds`. Cette option vous permet de restreindre la conversion des données des associations hasMany/belongsToMany pour utiliser uniquement la liste des `_ids`. Cette option est par défaut à `false`.

Query

— `Query::notMatching()` a été ajoutée.
— `Query::leftJoinWith()` a été ajoutée.
— `Query::innerJoinWith()` a été ajoutée.
— `Query::select()` supporte maintenant des objets `Table` et `Association` en paramètres. Ces types de paramètres sélectionneront toutes les colonnes de l'instance de la table ou la table ciblée par l'association.
— `Query::distinct()` accepte maintenant une chaine de caractères pour les distinct sur une seule colonne.
— `Table::loadInto()` a été ajoutée.

— Les fonctions SQL brutes `EXTRACT`, `DATE_ADD` et `DAYOFWEEK` ont été ajoutées avec `extract()`, `dateAdd()` et `dayOfWeek()`.

View

— Vous pouvez maintenant définir `_serialized` à `true` pour `JsonView` et `XmlView` pour sérialiser toutes les variables de vue au lieu de les spécifier explicitement.
— `View::$viewPath` est déprécié. Vous devez utilisez `View::templatePath()` à la place.
— `View::$view` est déprécié. Vous devez utilisez `View::template()` à la place.
— `View::TYPE_VIEW` est déprécié. Vous devez utilisez `View::TYPE_TEMPLATE` à la place.

Helper

SessionHelper

— `SessionHelper` a été dépréciée. Vous pouvez directement utiliser `$this->request->session()`.

FlashHelper

— `FlashHelper` peut maintenant rendre plusieurs messages si plusieurs messages ont été enregistrés avec le `FlashComponent`. Chaque message sera rendu dans son propre élément. Les messages seront rendus dans l'ordre dans lequel ils ont été enregistrés.

FormHelper

— Une nouvelle option `templateVars` a été ajoutée. `templateVars` vous permet de passer des variables supplémentaires à vos templates de formulaire personnalisés.

Email

— Les classes `Email` et `Transport` ont été déplacées sous le namespace `Cake\Mailer`. Leur ancien namespace est toujours utilisable car des alias ont été créés.
— Le profil d'email `default` est maintenant automatiquement défini quand une instance `Email` est créée. Ce comportement est le même que dans 2.x.

Mailer

— La classe `Mailer` a été ajoutée. Cette classe aide à créer des emails réutilisables dans une application.

I18n

Time

- `Time::fromNow()` a été ajoutée. Cette méthode facilite le calcul de différence depuis l'instant présent.
- `Time::i18nFormat()` supporte les calendriers non-grégorien lors du formatage des dates.

Validation

- `Validation::geoCoordinate()` a été ajoutée.
- `Validation::latitude()` a été ajoutée.
- `Validation::longitude()` a été ajoutée.
- `Validation::isInteger()` a été ajoutée.
- `Validation::ascii()` a été ajoutée.
- `Validation::utf8()` a été ajoutée.

Testing

TestFixture

La clé `model` est maintenant supportée pour récupérer le nom de la table pour l'import.

3.0 Guide de Migration

Guide de Migration du Nouvel ORM

CakePHP 3.0 apporte un nouvel ORM qui a été réécrit de zéro. Alors que l'ORM utilisé dans 1.x et 2.x nous a bien servi pendant un long moment, il avait quelques problèmes que nous souhaitions régler.

- Frankenstein - est-ce un enregistrement, ou une table ? Actuellement c'est les deux.
- API incohérente - Model : :read() par exemple.
- Pas d'objet query - Les queries sont toujours définies comme des tableaux, ceci amène quelques limitations et restrictions. Par exemple, cela rend les unions et les sous-requêtes plus compliquées.
- Retourne des tableaux. C'est une plainte courante au sujet de CakePHP, et ceci a probablement réduit le passage à certains niveaux.
- Pas d'objet d'enregistrement - Ceci rend l'attachement de méthodes de format difficile/impossible.
- Containable - Devrait être une partie de l'ORM, pas un behaviour compliqué.
- Recursive - Ceci devrait être mieux contrôlé en définissant quelles associations sont inclues, et pas un niveau de récursivité.
- DboSource - C'est un calvaire, et le Model repose dessus plus que sur la source de données. Cette séparation pourrait être plus propre et plus simple.
- Validation - Devrait être séparée, c'est actuellement une énorme fonction ingérable. La rendre un peu plus réutilisable rendrait le framework plus extensible.

L'ORM dans CakePHP 3.0 résout ces problèmes et beaucoup d'autres encore. Le nouvel ORM se focalise actuellement sur les stockages des données relationnelles. Dans le future et à travers les plugins, nous ajouterons des stockages non relationnels comme ElasticSearch et d'autres encore.

Design du nouvel ORM

Le nouvel ORM résout de nombreux problèmes en ayant des classes plus spécialisées et concentrées. Dans le passé vous utilisiez `Model` et une Source de données pour toutes les opérations. Maintenant l'ORM est séparé en plus de couches :

- `Cake\Database\Connection` - Fournit un moyen de créer et utiliser des connections indépendamment de la plateforme. Cette classe permet d'utiliser les transactions, d'exécuter les queries et d'accéder aux données du schema.
- `Cake\Database\Dialect` - Les classes dans ce namespace fournissent le SQL spécifique à une plateforme et transforment les queries pour fonctionner selon les limitations spécifiques de celle ci.
- `Cake\Database\Type` - Est la classe de passerelle vers le système de conversion de type de base de données de CakePHP. C'est un framework modulable pour l'ajout des types de colonnes abstraites et pour fournir des mappings entre base de données, les représentations de PHP et les liens PDO pour chaque type de données. Par exemple, les colonnes datetime sont maintenant représentées comme des instances `DateTime` dans votre code.
- `Cake\ORM\Table` - Le point d'entrée principal dans le nouvel ORM. Fournit l'accès à une table unique. Gère la définition d'association, utilise les behaviors et la création d'entités et d'objets query.
- `Cake\ORM\Behavior` - La classe de base pour les behaviors, qui agit de façon très similaire aux behaviors dans les versions précédentes de CakePHP.
- `Cake\ORM\Query` - Un générateur de requêtes simple basé sur les objets qui remplace les tableaux profondément imbriqués utilisés dans les versions précédentes de CakePHP.
- `Cake\ORM\ResultSet` - Une collection de résultats qui donne des outils puissants pour manipuler les données dans l'ensemble.
- `Cake\ORM\Entity` - Représente le résultat d'un enregistrement unique. Rend les données accessibles et sérialisable vers des formats divers en un tour de main.

Maintenant que vous êtes plus familier avec certaines des classes avec lesquelles vous intéragissez le plus fréquemment dans le nouvel ORM, il est bon de regarder les trois plus importantes classes. Les classes `Table`, `Query` et `Entity` sont les grandes nouveautés du lifting du nouvel ORM, et chacun sert un objectif différent.

Les Objets Table

Les objets Table sont la passerelle vers vos données. Ils gèrent plusieurs des tâches que le `Model` faisait dans les versions précédentes. Les classes de Table gèrent les tâches telles que :

- Créer des queries.
- Fournir des finders.
- Valider et sauvegarder des entités.
- Supprimer des entités.
- Définir & accéder aux associations.
- Déclencher les événements de callback.
- Interagir avec les behaviors.

Le chapitre de la documentation sur *Les Objets Table* fournit bien plus de détails sur la façon d'utiliser les objets de table que ce guide. Généralement quand on déplace le code du model existant, il va finir dans un objet Table. Les objets Table ne contiennent aucun SQL dépendant de la plateforme. A la place, ils collaborent avec les entités et le générateur de requêtes pour faire leur travail. Les objets Table intéragissent aussi avec les behaviors et d'autres parties à travers les événements publiés.

Les Objets Query

Alors que celles-ci ne sont pas des classes que vous allez construire vous-même, le code de votre application fera un usage intensif du *Générateur de Requêtes* qui est central dans le nouvel ORM. Le générateur de requêtes facilite la construction de requêtes simples ou complexes incluant celles qui étaient précédemment très difficiles dans CakePHP comme `HAVING`, `UNION` et les sous-requêtes.

Les différents appels de find() que votre application utilise couramment auront besoin d'être mis à jour pour utiliser le nouveau générateur de requête. L'objet Query est responsable de la façon de contenir les données pour réaliser une requête sans l'exécuter. Elle collabore avec la connection/dialect pour générer le SQL spécifique à la plateforme qui est exécutée en créant un `ResultSet` en sortie.

Les Objets Entity

Dans les versions précédentes de CakePHP la classe `Model` retournait des tableaux idiots qui ne contenaient pas de logique ou de behavior. Alors que la communauté rendait cela accessible et moins douloureux avec les projets comme CakeEntity, le tableau de résutlats était souvent une source de difficulté pour beaucoup de développeurs. Pour CakePHP 3.0, l'ORM retourne toujours l'ensemble des résultats en objet à moins que vous ne désactiviez explicitement cette fonctionnalité. Le chapitre sur *Entities* couvre les différentes tâches que vous pouvez accomplir avec les entities.

Les entities sont créées en choisissant l'une des deux façons suivantes. Soit en chargeant les données à partir de la base de données, soit en convertissant les données de requête en entities. Une fois créées, les entities vous permettent de manipuler les données qu'elles contiennent et font persister leurs données en collaborant avec les objets Table.

Différences Clé

Le nouvel ORM est un grand renouveau par rapport à la couche `Model` existante. Il y a plusieurs différences importantes à comprendre sur la façon dont le nouvel ORM opère et comment mettre à jour votre code.

Les Règles d'Inflection Mises à Jour

Vous avez peut-être noté que les classes Table ont un nom pluralisé. En plus d'avoir les noms pluralisés, les associations se réfèrent aussi à la forme plurielle. C'est en opposition par rapport au `Model` où les noms et associations étaient au singulier. Il y avait plusieurs raisons pour ce changement :

- Les classes de Table représentent des **collections** de données, pas des enregistrements uniques.
- Les associations lient les tables ensemble, décrivant les relations entre plusieurs choses.

Alors que les conventions pour les objets Table sont de toujours utiliser les formes plurielles, les propriétés d'association de votre entitie seront remplies en se basant sur le type d'association.

Note : Les associations BelongsTo et HasOne utiliseront la forme au singulier, tandis que HasMany et BelongsToMany (HABTM) utiliseront la forme plurielle.

Le changement de convention pour les objects Table est plus apparent lors de la construction de queries. A la place d'expressions de requêtes comme :

```
// Faux
$query->where(['User.active' => 1]);
```

Vous avez besoin d'utiliser la forme au pluriel :

```
// Correct
$query->where(['Users.active' => 1]);
```

Find retourne un Objet Query

Une différence importante dans le nouvel ORM est qu'appeler `find` sur une table ne va pas retourner les résultats immédiatement, mais va retourner un objet Query ; cela sert dans plusieurs cas.

Il est possible de modifier les requêtes plus tard, après avoir appeler `find` :

```
// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->get('Articles');
$query = $articles->find();
$query->where(['author_id' => 1])->order(['title' => 'DESC']);
```

Il est possible d'empiler les finders personnalisés pour ajouter les conditions à la suite, pour trier, limiter et toute autre clause pour la même requête avant qu'elle ne soit exécutée :

```
$query = $articles->find('approved')->find('popular');
$query->find('latest');
```

Vous pouvez composer des requêtes l'une dans l'autre pour créer des sous-requêtes plus facilement que jamais :

```
$query = $articles->find('approved');
$favoritesQuery = $article->find('favorites', ['for' => $user]);
$query->where(['id' => $favoritesQuery->select(['id'])]);
```

Vous pouvez décorer les requêtes avec des itérateurs et des méthodes d'appel sans même toucher à la base de données, c'est bien quand vous avez des parties de votre view mise en cache et avez les résultats pris à partir de la base de données qui n'est en fait pas nécessaire :

```
// Pas de requêtes faites dans cet exemple!
$results = $articles->find()
    ->order(['title' => 'DESC'])
    ->formatResults(function (\Cake\Collection\CollectionInterface $results) {
        return $results->extract('title');
    });
```

Les requêtes peuvent être vues comme un objet de résultat, essayant d'itérer la requête, appelant `toArray()` ou toute méthode héritée de *collection*, va faire que la requête sera exécutée et les résultats vous seront retournés.

La plus grande différence que vous trouverez quand vous venez de CakePHP 2.x est que `find('first')` n'existe plus. Il existe un remplacement trivial pour cela et il s'agit de la méthode `first()` :

```
// Avant
$article = $this->Article->find('first');

// Maintenant
$article = $this->Articles->find()->first();

// Avant
$article = $this->Article->find('first', [
    'conditions' => ['author_id' => 1]
]);
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Maintenant
$article = $this->Articles->find('all', [
    'conditions' => ['author_id' => 1]
])->first();

// Peut aussi être écrit
$article = $this->Articles->find()
    ->where(['author_id' => 1])
    ->first();
```

Si vous chargez un enregistrement unique par sa clé primaire, il serait mieux de juste appeler `get()` :

```
$article = $this->Articles->get(10);
```

Changements de la méthode Finder

Retourner un objet Query à partir d'une méthode find a plusieurs avantages, mais vient avec un coût pour les gens migrant de 2.x. Si vous aviez quelques méthodes find personnalisées dans vos models, elles auront besoin de quelques modifications. C'est de cette façon que vous créez les méthodes finder personnalisées dans 3.0 :

```
class ArticlesTable
{

    public function findPopular(Query $query, array $options)
    {
        return $query->where(['times_viewed' > 1000]);
    }

    public function findFavorites(Query $query, array $options)
    {
        $for = $options['for'];
        return $query->matching('Users.Favorites', function ($q) use ($for) {
            return $q->where(['Favorites.user_id' => $for]);
        });
    }
}
```

Comme vous pouvez le voir, ils sont assez simples, ils obtiennent un objet Query à la place d'un tableau et doivent retourner un objet Query en retour. Pour 2.x, les utilisateurs qui implémentaient la logique afterFind dans les finders personnalisés, vous devez regarder la section *Modifier les Résultats avec Map/Reduce*, ou utiliser les *collections*. Si dans vos models, vous aviez pour habitude d'avoir un afterFind pour toutes les opérations de find, vous pouvez migrer ce code d'une des façons suivantes :

1. Surcharger la méthode constructeur de votre entity et faire le formatage supplémentaire ici.
2. Créer des méthodes accesseurs dans votre entity pour créer les propriétés virtuelles.
3. Redéfinir `findAll()` et utiliser `formatResults`.

Dans le 3ème cas ci-dessus, votre code ressemblerait à :

```
public function findAll(Query $query, array $options)
{
    return $query->formatResults(function (\Cake\Collection\CollectionInterface
↪$results) {
        return $results->map(function ($row) {
            // Votre logique afterfind
```

(suite sur la page suivante)

(suite de la page précédente)

```
        });
    })
}
```

Vous avez peut-être noté que les finders personnalisés reçoivent un tableau d'options, vous pouvez passer toute information supplémentaire à votre finder en utilisant ce paramètre. C'est une bonne nouvelle pour la migration de gens à partir de 2.x. Chacune des clés de requêtées qui a été utilisée dans les versions précédentes sera convertie automatiquement pour vous dans 3.x vers les bonnes fonctions :

```
// Ceci fonctionne dans les deux CakePHP 2.x et 3.0
$articles = $this->Articles->find('all', [
    'fields' => ['id', 'title'],
    'conditions' => [
        'OR' => ['title' => 'Cake', 'author_id' => 1],
        'published' => true
    ],
    'contain' => ['Authors'], // Le seul changement! (notez le pluriel)
    'order' => ['title' => 'DESC'],
    'limit' => 10,
]);
```

Si votre application utilise des *Finders Dynamiques* ou "magiques", vous devrez adapter ces appels. Dans 3.x, les méthodes `findAllBy*` ont été retirées, à la place `findBy*` retourne toujours un objet query. Pour récupérer le premier résultat, vous devrez utiliser la méthode `first()` :

```
$article = $this->Articles->findByTitle('Un super post!')->first();
```

Heureusement, la migration à partir des versions anciennes n'est pas aussi difficile qu'il y paraît, la plupart des fonctionnalités que nous avons ajoutées vous aide à retirer le code puisque vous pouvez mieux exprimer vos exigences en utilisant le nouvel ORM et en même temps les wrappers de compatibilité vous aideront à réécrire ces petites différences d'une façon rapide et sans douleur.

Une des autres améliorations sympas dans 3.x autour des méthodes finder est que les behaviors peuvent implémenter les méthodes finder sans aucun soucis. En définissant simplement une méthode avec un nom matchant et la signature sur un Behavior le finder sera automatiquement disponible sur toute table sur laquelle le behavior est attaché.

Recursive et ContainableBehavior Retirés.

Dans les versions précédentes de CakePHP, vous deviez utiliser `recursive`, `bindModel()`, `unbindModel()` et `ContainableBehavior` pour réduire les données chargées pour l'ensemble des associations que vous souhaitiez. Une tactique habituelle pour gérer les associations était de définir `recursive` à `-1` et d'utiliser Containable pour gérer toutes les associations. Dans CakePHP 3.0 ContainableBehavior, recursive, bindModel, et unbindModel ont été retirées. A la place, la méthode `contain()` a été favorisée pour être une fonctionnalité du cœur du query builder. Les associations sont seulement chargées si elles sont explicitement activées. Par exemple :

```
$query = $this->Articles->find('all');
```

Va **seulement** charger les données à partir de la table `articles` puisque aucune association n'a été inclue. Pour charger les articles et leurs auteurs liés, vous feriez :

```
$query = $this->Articles->find('all')->contain(['Authors']);
```

En chargeant seulement les données associées qui on été spécifiquement requêtées vous ne passez pas votre temps à vous battre avec l'ORM à essayer d'obtenir seulement les données que vous souhaitez.

Pas d'Event afterFind ou de Champs Virtuels

Dans les versions précédentes de CakePHP, vous aviez besoin de rendre extensive l'utilisation du callback `afterFind` et des champs virtuels afin de créer des propriétés de données générées. Ces fonctionnalités ont été retirées dans 3.0. Du fait de la façon dont ResultSets génèrent itérativement les entities, le callback `afterFind` n'était pas possible. afterFind et les champs virtuels peuvent tous deux largement être remplacés par les propriétés virtuelles sur les entities. Par exemple si votre entité User a les deux colonnes first et last name, vous pouvez ajouter un accesseur pour *full_name* et générer la propriété à la volée :

```
namespace App\Model\Entity;

use Cake\ORM\Entity;

class User extends Entity
{
    public function getFullName()
    {
        return $this->first_name . '  ' $this->last_name;
    }
}
```

Une fois définie, vous pouvez accéder à votre nouvelle propriété en utilisant `$user->full_name`. L'utilisation des fonctionnalités *Modifier les Résultats avec Map/Reduce* de l'ORM vous permettent de construire des données agrégées à partir de vos résultats, ce qui était souvent un autre cas d'utilisation callback `afterFind`.

Alors que les champs virtuels ne sont plus une fonctionnalité de l'ORM, l'ajout des champs calculés est facile à faire dans les méthodes finder. En utilisant le query builder et les objets expression, vous pouvez atteindre les mêmes résultats que les champs virtuels, cela donne :

```
namespace App\Model\Table;

use Cake\ORM\Table;
use Cake\ORM\Query;

class ReviewsTable extends Table
{
    public function findAverage(Query $query, array $options = [])
    {
        $avg = $query->func()->avg('rating');
        $query->select(['average' => $avg]);
        return $query;
    }
}
```

Les Associations Ne sont Plus Définies en Propriétés

Dans les versions précédentes de CakePHP, les diverses associations que vos models avaient, ont été définies dans les propriétés comme `$belongsTo` et `$hasMany`. Dans CakePHP 3.0, les associations sont créées avec les méthodes. L'utilisation de méthodes vous permet de mettre de côté plusieurs définitions de classes de limitations, et fournissent seulement une façon de définir les associations. Votre méthode `initialize()` et toutes les autres parties de votre code d'application, interagit avec la même API lors de la manipulation des associations :

```
namespace App\Model\Table;

use Cake\ORM\Table;
```

(suite sur la page suivante)

(suite de la page précédente)

```
use Cake\ORM\Query;

class ReviewsTable extends Table
{

    public function initialize(array $config)
    {
        $this->belongsTo('Movies');
        $this->hasOne('Rating');
        $this->hasMany('Comments')
        $this->belongsToMany('Tags')
    }

}
```

Comme vous pouvez le voir de l'exemple ci-dessus, chaque type d'association utilise une méthode pour créer l'association. Une autre différence est que `hasAndBelongsToMany` a été renommée en `belongsToMany`. Pour en apprendre plus sur la création des associations dans 3.0, regardez la section sur *les associations*.

Une autre amélioration bienvenue de CakePHP est la capacité de créer votre propre classe d'associations. Si vous avez des types d'association qui ne sont pas couverts par les types de relations intégrées, vous pouvez créer une sous-classe `Association` personnalisée et définir la logique d'association dont vous avez besoin.

Validation n'est plus Définie Comme une Propriété

Comme les associations, les règles de validation ont été définies comme une propriété de classe dans les versions précédentes de CakePHP. Ce tableau sera ensuite transformé paresseusement en un objet `ModelValidator`. Cette étape de transformation ajoutée en couche d'indirection, compliquant les changements de règle lors de l'exécution. De plus, les règles de validation étant définies comme propriété rendait difficile pour un model d'avoir plusieurs ensembles de règles de validation. Dans CakePHP 3.0, on a remédié à deux de ces problèmes. Les règles de validation sont toujours construites avec un objet `Validator`, et il est trivial d'avoir plusieurs ensembles de règles :

```
namespace App\Model\Table;

use Cake\ORM\Table;
use Cake\ORM\Query;
use Cake\Validation\Validator;

class ReviewsTable extends Table
{

    public function validationDefault(Validator $validator)
    {
        $validator->requirePresence('body')
            ->add('body', 'length', [
                'rule' => ['minLength', 20],
                'message' => 'Reviews must be 20 characters or more',
            ])
            ->add('user_id', 'numeric', [
                'rule' => 'numeric'
            ]);
        return $validator;
    }

}
```

Vous pouvez définir autant de méthodes de validation que vous souhaitez. Chaque méthode doit être préfixée avec `validation` et accepte un argument `$validator`.

Dans les versions précédentes de CakePHP, la "validation" et les callbacks liés couvraient quelques utilisations liées mais différentes. Dans CakePHP 3.0, ce qui était avant appelé validation est maintenant séparé en deux concepts :

1. Validation du type de données et du format.
2. Vérification des règles métiers.

La validation est maintenant appliquée avant que les entities de l'ORM ne soient créées à partir des données de request. Cette étape permet de vous assurer que les données correspondent au type de données, au format et à la forme de base que votre application attend. Vous pouvez utiliser vos validateurs quand vous convertissez en entities les données de request en utilisant l'option `validate`. Consultez la documentation *Convertir les Données Requêtées en Entities* pour plus d'informations.

Les règles d'Application vous permettent de définir les règles qui s'assurent que vos règles d'application, l'état et les flux de travail sont remplis. Les règles sont définies dans la méthode `buildRules()` de votre Table. Les behaviors peuvent ajouter des règles en utilisant la méthode `buildRules()`. Un exemple de méthode `buildRules()` pour notre table articles pourrait être :

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

use Cake\ORM\Table;
use Cake\ORM\RulesChecker;

class ArticlesTable extends Table
{
    public function buildRules(RulesChecker $rules)
    {
        $rules->add($rules->existsIn('user_id', 'Users'));
        $rules->add(
            function ($article, $options) {
                return ($article->published && empty($article->reviewer));
            },
            'isReviewed',
            [
                'errorField' => 'published',
                'message' => 'Articles must be reviewed before publishing.'
            ]
        );
        return $rules;
    }
}
```

Identifier Quoting Désactivé par Défaut

Dans le passé, CakePHP a toujours quoté les identifieurs. Parser les bouts de code SQL et tenter de quoter les identifiers étaient tous les deux des source d'erreurs et coûteuses. Si vous suivez les conventions que CakePHP définit, les coûts du identifier quoting l'emporte sur tout avantage qu'il fournisse. Puisque ce identifier quoting a été désactivé par défaut dans 3.0. Vous devriez seulement activer le identifier quoting si vous utilisez les noms de colonne ou les noms de table qui contiennent des caractères spéciaux ou sont des mots réservés. Si nécessaire, vous pouvez activer identifier quoting lors de la configuration d'une connection :

```
// Dans config/app.php
'Datasources' => [
```

(suite sur la page suivante)

(suite de la page précédente)

```
    'default' => [
        'className' => 'Cake\Database\Driver\Mysql',
        'username' => 'root',
        'password' => 'super_secret',
        'host' => 'localhost',
        'database' => 'cakephp',
        'quoteIdentifiers' => true
    ]
],
```

Note : Les identifiers dans les objets `QueryExpression` ne seront pas quotés, et vous aurez besoin de les quoter manuellement ou d'utiliser les objets IdentifierExpression.

Mise à jour des behaviors

Comme la plupart des fonctionnalités liées à l'ORM, les behaviors ont aussi changé dans 3.0. Ils attachent maintenant les instances à `Table` qui sont les descendants conceptuels de la classe `Model` dans les versions précédentes de CakePHP. Il y a quelques petites différences clés par rapport aux behaviors de CakePHP 2.x :

- Les Behaviors ne sont plus partagés par plusieurs tables. Cela signifie que vous n'avez plus à "donner un namespace" aux configurations stockés dans behavior. Chaque table utilisant un behavior va créer sa propre instance.
- Les signatures de méthode pour les méthodes mixin a changé.
- Les signatures de méthode pour les méthodes de callback a changé.
- La classe de base pour les behaviors a changé.
- Les Behaviors peuvent ajouter des méthodes find.

Nouvelle classe de Base

La classe de base pour les behaviors a changé. Les Behaviors doivent maintenant étendre `Cake\ORM\Behavior`; si un behavior n'étend pas cette classe, une exception sera lancée. En plus du changement de classe de base, le constructeur pour les behaviors a été modifié, et la méthode `startup()` a été retirée. Les Behaviors qui ont besoin d'accéder à la table à laquelle ils sont attachés, doivent définir un constructeur :

```
namespace App\Model\Behavior;

use Cake\ORM\Behavior;

class SluggableBehavior extends Behavior
{

    protected $_table;

    public function __construct(Table $table, array $config)
    {
        parent::__construct($table, $config);
        $this->_table = $table;
    }

}
```

Changements de Signature des Méthodes Mixin

Les Behaviors continuent d'offrir la possibilité d'ajouter les méthodes "mixin" à des objets Table, cependant la signature de méthode pour ces méthodes a changé. Dans CakePHP 3.0, les méthodes mixin du behavior peuvent attendre les **mêmes** arguments fournis à la table "method". Par exemple :

```
// Supposons que la table a une méthode slug() fournie par un behavior.
$table->slug($someValue);
```

Le behavior qui fournit la méthode `slug()` va recevoir seulement 1 argument, et ses méthodes signature doivent ressembler à ceci :

```
public function slug($value)
{
    // code ici.
}
```

Changements de Signature de Méthode de Callback

Les callbacks de Behavior ont été unifiés avec les autres méthodes listener. Au lieu de leurs arguments précédents, ils attendent un objet event en premier argument :

```
public function beforeFind(Event $event, Query $query, array $options)
{
    // code.
}
```

Regardez *Callbacks du Cycle de Vie* pour les signatures de tous les callbacks auquel un behavior peut souscrire.

Informations générales

Le Processus de Développement CakePHP

Ici, nous tenterons d'expliquer le processus que nous utilisons lors de l'élaboration du CakePHP. Nous comptons beaucoup sur l'interaction communautaire par le biais billets et par le chat IRC. IRC est le meilleur endroit pour trouver des membres de l'équipe de développement [195] et pour discuter d'idées, du dernier code, et de faire des commentaires généraux. Si quelque chose de plus formel doit être proposé ou s'il y a un problème avec une version sortie, le système de ticket est le meilleur endroit pour partager vos idées.

Nous maintenons 4 versions de CakePHP.

- **version taggée** : Versions taggées pour la production où la stabilité est plus importante que les fonctionnalités. Les questions déposées pour ces versions seront réglées dans la branche connexe, et feront parti de la prochaine version.
- **branche principale** : Ces branches contiennent tous les correctifs de bug. Les versions stables sont taggées à partir de ces branches. `master` est la branche principale pour les séries de versions actuelles. `2.x` est la branche de maintenance pour les séries de la version 2.x. Si vous utilisez une version stable et que vous avez besoin de correctifs qui n'ont pas fait leur chemin dans une version taggée, vérifiez ici.
- **Branches de développement** : Les branches de développement contiennent les derniers correctifs et fonctionnalités. Elles sont nommées d'après le numéro de version pour lesquels elles sont faites. Par ex : *3.next*. Une fois que les branches de développement ont atteint un niveau de version stable, elles sont fusionnées dans la branche principale.

195. https://github.com/cakephp?tab=members

— **Branches de fonctionnalité** : Les branches de fonctionnalité contiennent des fonctionnalités non-finies, possiblement instable et sont recommandées uniquement pour les utilisateurs avertis intéressés dans la fonctionnalité la plus avancée et qui souhaitent contribuer à la communauté. Les branches de fonctionnalité sont nommées selon la convention suivante de *version-fonctionnalité*. Un exemple serait *3.3-router* qui contiendrait de nouvelles fonctionnalités pour le Routeur dans 3.3.

Espérons que cela vous aide à comprendre quelle version est bonne pour vous. Une fois que vous choisissez votre version, vous vous sentirez peut-être contraints de contribuer à un report de bug ou de faire des commentaires généraux sur le code.

— Si vous utilisez une version stable ou une branche de maintenance, merci de soumettre des tickets ou discuter avec nous sur IRC.
— Si vous utilisez la branche de développement ou la branche de fonctionnalité, le premier endroit où aller est IRC. Si vous avez un commentaire et ne pouvez pas nous atteindre sur IRC après un jour ou deux, merci de nous soumettre un ticket.

Si vous trouvez un problème, la meilleure réponse est d'écrire un test. Le meilleur conseil que nous pouvons offrir dans l'écriture des tests est de regarder ceux qui sont inclus dans le cœur.

Comme toujours, si vous avez n'importe quelle question ou commentaire, rendez-nous une visite sur #cakephp sur irc.freenode.net.

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, 549
Cake\Collection, 739
Cake\Console\Exception, 617
Cake\Controller, 219
Cake\Controller\Component, 256
Cake\Controller\Exception, 617
Cake\Core, 165
Cake\Core\Exception, 618
Cake\Database, 392
Cake\Database\Exception, 617
Cake\Database\Schema, 542
Cake\Datasource, 391
Cake\Datasource\Exception, 617
Cake\Error, 596
Cake\Filesystem, 759
Cake\Form, 653
Cake\Http, 783
Cake\Http\Client, 789
Cake\Http\Cookie, 215
Cake\Http\Exception, 615
Cake\I18n, 813
Cake\Log, 651
Cake\Mailer, 603
Cake\ORM, 433
Cake\ORM\Behavior, 532
Cake\ORM\Exception, 617
Cake\Routing, 171
Cake\Routing\Exception, 617
Cake\Utility, 821
Cake\Validation, 725
Cake\View, 273
Cake\View\Exception, 616
Cake\View\Helper, 376

Index

Symboles

() (*méthode*), **270**
:action, 173
:controller, 173
:plugin, 173
$this->request, 199
$this->response, 206
__() (*global function*), **825**
__d() (*global function*), **825**
__dn() (*global function*), **826**
__dx() (*global function*), **826**
__dxn() (*global function*), **826**
__n() (*global function*), **826**
__x() (*global function*), **826**
__xn() (*global function*), **826**

A

acceptLanguage() (*Cake\Http\ServerRequest method*), **206**
accepts() (*Cake\Http\ServerRequest method*), **206**
accepts() (*méthode RequestHandlerComponent*), **262**
addArgument() (*Cake\Console\ConsoleOptionParser method*), **576**
addArguments() (*Cake\Console\ConsoleOptionParser method*), **576**
addBehavior() (*Cake\ORM\Table method*), **439**
addCrumb() (*Cake\View\Helper\HtmlHelper method*), **354**
addDetector() (*Cake\Http\ServerRequest method*), **203**
addOption() (*Cake\Console\ConsoleOptionParser method*), **577**
addOptions() (*Cake\Console\ConsoleOptionParser method*), **577**
addPathElement() (*Cake\Filesystem\Folder method*), **760**
addSubcommand() (*Cake\Console\ConsoleOptionParser method*), **578**
admin routing, 180
afterDelete() (*Cake\ORM\Table method*), **438**
afterDeleteCommit() (*Cake\ORM\Table method*), **439**
afterFilter() (*Cake\Controller\Controller method*), **227**
afterLayout() (*méthode Helper*), **384**
afterRender() (*méthode Helper*), **384**
afterRenderFile() (*méthode Helper*), **384**
afterRules() (*Cake\ORM\Table method*), **438**
afterSave() (*Cake\ORM\Table method*), **438**
afterSaveCommit() (*Cake\ORM\Table method*), **438**
alert() (*Cake\Log\Log method*), **651**
allControls() (*Cake\View\Helper\FormHelper method*), **337**
allow() (*méthode AuthComponent*), **244**
allowMethod() (*Cake\Http\ServerRequest method*), **205**
App (*classe dans Cake\Core*), **735**
APP (*global constant*), **827**
app.php, 161
app.php.default, 161
APP_DIR (*global constant*), **827**
append() (*Cake\Collection\Collection method*), **754**
append() (*Cake\Filesystem\File method*), **764**
application exceptions, 618
apply() (*Cake\Utility\Hash method*), **777**
attachments() (*Cake\Mailer\Email method*), **608**
attributs HTML, **875**
AuthComponent (*class*), **228**
autoLink() (*Cake\View\Helper\TextHelper method*), **372**
autoLinkEmails() (*Cake\View\Helper\TextHelper method*), **371**
autoLinkUrls() (*Cake\View\Helper\TextHelper method*), **372**

autoParagraph() (*Cake\View\Helper\TextHelper method*), 372
avg() (*Cake\Collection\Collection method*), 747

B

BadRequestException, 615
beforeDelete() (*Cake\ORM\Table method*), 438
beforeFilter() (*Cake\Controller\Controller method*), 227
beforeFind() (*Cake\ORM\Table method*), 437
beforeLayout() (*méthode Helper*), 384
beforeMarshal() (*Cake\ORM\Table method*), 437
beforeRender() (*Cake\Controller\Controller method*), 227
beforeRender() (*méthode Helper*), 384
beforeRenderFile() (*méthode Helper*), 384
beforeRules() (*Cake\ORM\Table method*), 437
beforeSave() (*Cake\ORM\Table method*), 438
blackHole() (*méthode SecurityComponent*), 253
BreadcrumbsHelper (*classe dans Cake\View\Helper*), 295
breakpoint() (*global function*), 595
buffered() (*Cake\Collection\Collection method*), 757
build() (*Cake\Utility\Xml method*), 821
build() (*Cake\View\Helper\UrlHelper method*), 377
buildFromArray() (*Cake\Console\ConsoleOptionParser method*), 579
buildRules() (*Cake\ORM\Table method*), 437
buildValidator() (*Cake\ORM\Table method*), 437
button() (*Cake\View\Helper\FormHelper method*), 331

C

Cache (*classe dans Cake\Cache*), 549
CACHE (*global constant*), 827
cache() (*Cake\View\View method*), 284
CacheEngine (*classe dans Cake\Cache*), 557
CAKE (*global constant*), 827
CAKE_CORE_INCLUDE_PATH (*global constant*), 827
Cake\Cache (*namespace*), 549
Cake\Collection (*namespace*), 739
Cake\Console (*namespace*), 561
Cake\Console\Exception (*namespace*), 617
Cake\Controller (*namespace*), 219
Cake\Controller\Component (*namespace*), 247, 251, 256
Cake\Controller\Exception (*namespace*), 617
Cake\Core (*namespace*), 165, 735
Cake\Core\Exception (*namespace*), 618
Cake\Database (*namespace*), 392
Cake\Database\Exception (*namespace*), 617
Cake\Database\Schema (*namespace*), 542
Cake\Datasource (*namespace*), 391
Cake\Datasource\Exception (*namespace*), 617
Cake\Error (*namespace*), 596
Cake\Filesystem (*namespace*), 759
Cake\Form (*namespace*), 653
Cake\Http (*namespace*), 199, 783
Cake\Http\Client (*namespace*), 789
Cake\Http\Cookie (*namespace*), 215
Cake\Http\Exception (*namespace*), 615
Cake\I18n (*namespace*), 797, 813
Cake\Log (*namespace*), 651
Cake\Mailer (*namespace*), 603
Cake\ORM (*namespace*), 402, 433, 441, 451, 484, 507
Cake\ORM\Behavior (*namespace*), 520, 522, 524, 532
Cake\ORM\Exception (*namespace*), 617
Cake\Routing (*namespace*), 171
Cake\Routing\Exception (*namespace*), 617
Cake\Utility (*namespace*), 673, 767, 793, 805, 821
Cake\Validation (*namespace*), 725
Cake\View (*namespace*), 273
Cake\View\Exception (*namespace*), 616
Cake\View\Helper (*namespace*), 295, 298, 299, 341, 355, 359, 368, 371, 376
camelize() (*Cake\Utility\Inflector method*), 795
cd() (*Cake\Filesystem\Folder method*), 760
CDN, 875
champ(s), 875
charset() (*Cake\View\Helper\HtmlHelper method*), 342
check() (*Cake\Controller\Component\CookieComponent method*), 249
check() (*Cake\Core\Configure method*), 166
check() (*Cake\Utility\Hash method*), 773
check() (*Cake\View\Helper\SessionHelper method*), 371
check() (*méthode Session*), 683
checkbox() (*Cake\View\Helper\FormHelper method*), 314
checkNotModified() (*Cake\Http\Response method*), 214
chmod() (*Cake\Filesystem\Folder method*), 760
chunk() (*Cake\Collection\Collection method*), 743
chunkWithKeys() (*Cake\Collection\Collection method*), 744
classify() (*Cake\Utility\Inflector method*), 795
classname() (*Cake\Core\App method*), 735
cleanInsert() (*Cake\Utility\Text method*), 807
clear() (*Cake\Cache\Cache method*), 554
clear() (*Cake\Cache\CacheEngine method*), 557
clear() (*Cake\ORM\TableRegistry method*), 441
clearGroup() (*Cake\Cache\Cache method*), 556
clearGroup() (*Cake\Cache\CacheEngine method*), 558
Client (*classe dans Cake\Http*), 783
clientIp() (*Cake\Http\ServerRequest method*), 205

close() (*Cake\Filesystem\File method*), 764
Collection (*classe dans Cake\Collection*), 739
Collection (*classe dans Cake\Database\Schema*), 546
collection() (*global function*), 826
colonnes, 875
combine() (*Cake\Collection\Collection method*), 742
combine() (*Cake\Utility\Hash method*), 770
compile() (*Cake\Collection\Collection method*), 758
components (*Cake\Controller\Controller property*), 226
CONFIG (*global constant*), 827
config() (*Cake\Cache\Cache method*), 550
config() (*Cake\I18n\Number method*), 802
config() (*Cake\Log\Log method*), 651
configTransport() (*Cake\Mailer\Email method*), 605
configuration, 161
Configure (*classe dans Cake\Core*), 165
configured() (*Cake\Log\Log method*), 651
ConflictException, 615
connect() (*Cake\Routing\Router method*), 173
Connection (*classe dans Cake\Database*), 396
ConnectionManager (*classe dans Cake\Datasource*), 391
ConsoleException, 617
ConsoleOptionParser (*classe dans Cake\Console*), 572
consume() (*Cake\Core\Configure method*), 167
consume() (*méthode Session*), 683
contains() (*Cake\Collection\Collection method*), 752
contains() (*Cake\Utility\Hash method*), 773
control() (*Cake\View\Helper\FormHelper method*), 304
Controller (*classe dans Cake\Controller*), 219
controls() (*Cake\View\Helper\FormHelper method*), 336
Cookie (*classe dans Cake\Http\Cookie*), 216
CookieCollection (*classe dans Cake\Http\Cookie*), 215
CookieComponent (*classe dans Cake\Controller\Component*), 247
copy() (*Cake\Filesystem\File method*), 764
copy() (*Cake\Filesystem\Folder method*), 760
core() (*Cake\Core\App method*), 736
CORE_PATH (*global constant*), 827
correctSlashFor() (*Cake\Filesystem\Folder method*), 761
countBy() (*Cake\Collection\Collection method*), 748
counter() (*Cake\View\Helper\PaginatorHelper method*), 364
CounterCacheBehavior (*classe dans Cake\ORM\Behavior*), 520
create() (*Cake\Filesystem\File method*), 764
create() (*Cake\Filesystem\Folder method*), 761
create() (*Cake\View\Helper\FormHelper method*), 300
createFile() (*Cake\Console\Shell method*), 567
critical() (*Cake\Log\Log method*), 651
CSRF, 875
css() (*Cake\View\Helper\HtmlHelper method*), 342
currency() (*Cake\I18n\Number method*), 798
currency() (*Cake\View\Helper\NumberHelper method*), 355
current() (*Cake\View\Helper\PaginatorHelper method*), 364

D

dasherize() (*Cake\Utility\Inflector method*), 795
date() (*Cake\View\Helper\FormHelper method*), 323
dateTime() (*Cake\View\Helper\FormHelper method*), 322
DAY (*global constant*), 828
day() (*Cake\View\Helper\FormHelper method*), 326
dd() (*global function*), 827
debug() (*Cake\Log\Log method*), 651
debug() (*global function*), 826
Debugger (*classe dans Cake\Error*), 596
decrement() (*Cake\Cache\Cache method*), 555
decrement() (*Cake\Cache\CacheEngine method*), 558
decrypt() (*Cake\Utility\Security method*), 673
defaultCurrency() (*Cake\I18n\Number method*), 798
defaultCurrency() (*Cake\View\Helper\NumberHelper method*), 356
defaultRouteClass() (*Cake\Routing\Router method*), 192
delete() (*Cake\Cache\Cache method*), 554
delete() (*Cake\Cache\CacheEngine method*), 557
delete() (*Cake\Controller\Component\CookieComponent method*), 249
delete() (*Cake\Core\Configure method*), 167
delete() (*Cake\Filesystem\File method*), 764
delete() (*Cake\Filesystem\Folder method*), 761
delete() (*Cake\ORM\Table method*), 507
delete() (*méthode Session*), 683
deleteAll() (*Cake\ORM\Table method*), 508
deleteMany() (*Cake\Cache\Cache method*), 554
deleteOrFail() (*Cake\ORM\Table method*), 508
deny() (*méthode AuthComponent*), 245
destroy() (*méthode Session*), 683
diff() (*Cake\Utility\Hash method*), 778
dimensions() (*Cake\Utility\Hash method*), 776
dirsize() (*Cake\Filesystem\Folder method*), 761
dirty() (*Cake\ORM\Entity method*), 445
disable() (*Cake\Cache\Cache method*), 557

doc (*role*), 129
docType() (*Cake\View\Helper\HtmlHelper method*), 345
domain() (*Cake\Http\ServerRequest method*), 204
drop() (*Cake\Cache\Cache method*), 552
drop() (*Cake\Core\Configure method*), 167
drop() (*Cake\Log\Log method*), 651
dropTransport() (*Cake\Mailer\Email method*), 606
DRY, 875
DS (*global constant*), 827
DSN, 875
dump() (*Cake\Core\Configure method*), 168
dump() (*Cake\Error\Debugger method*), 596

E

each() (*Cake\Collection\Collection method*), 740
element() (*Cake\View\View method*), 282
Email (*classe dans Cake\Mailer*), 603
emailPattern() (*Cake\Mailer\Email method*), 610
emergency() (*Cake\Log\Log method*), 651
enable() (*Cake\Cache\Cache method*), 557
enabled() (*Cake\Cache\Cache method*), 557
encrypt() (*Cake\Utility\Security method*), 673
end() (*Cake\View\Helper\FormHelper method*), 332
Entity (*classe dans Cake\ORM*), 441
env() (*Cake\Http\ServerRequest method*), 201
env() (*global function*), 826
error() (*Cake\Log\Log method*), 651
error() (*Cake\View\Helper\FormHelper method*), 329
errors() (*Cake\Filesystem\Folder method*), 761
errors() (*Cake\ORM\Entity method*), 446
every() (*Cake\Collection\Collection method*), 745
Exception, 618
ExceptionRenderer (*classe dans Cake\Core\Exception*), 618
excerpt() (*Cake\Error\Debugger method*), 597
excerpt() (*Cake\Utility\Text method*), 811
excerpt() (*Cake\View\Helper\TextHelper method*), 375
executable() (*Cake\Filesystem\File method*), 764
execute() (*Cake\Database\Connection method*), 397
exists() (*Cake\Filesystem\File method*), 764
expand() (*Cake\Utility\Hash method*), 775
ext() (*Cake\Filesystem\File method*), 764
extensions() (*Cake\Routing\Router method*), 184
extract() (*Cake\Collection\Collection method*), 741
extract() (*Cake\Utility\Hash method*), 768

F

fallbacks() (*Cake\Routing\Router method*), 193
File (*classe dans Cake\Filesystem*), 763
file extensions, 183
file() (*Cake\View\Helper\FormHelper method*), 320
filter() (*Cake\Collection\Collection method*), 744
filter() (*Cake\Utility\Hash method*), 774
find() (*Cake\Filesystem\Folder method*), 761
find() (*Cake\ORM\Table method*), 452
findRecursive() (*Cake\Filesystem\Folder method*), 762
first() (*Cake\Collection\Collection method*), 754
first() (*Cake\View\Helper\PaginatorHelper method*), 363
firstMatch() (*Cake\Collection\Collection method*), 745
FlashComponent (*classe dans Cake\Controller\Component*), 251
FlashHelper (*classe dans Cake\View\Helper*), 298
flatten() (*Cake\Utility\Hash method*), 774
Folder (*Cake\Filesystem\File property*), 763
Folder (*classe dans Cake\Filesystem*), 760
Folder() (*Cake\Filesystem\File method*), 764
ForbiddenException, 615
Form (*classe dans Cake\Form*), 653
format() (*Cake\I18n\Number method*), 800
format() (*Cake\Utility\Hash method*), 772
format() (*Cake\View\Helper\NumberHelper method*), 357
formatDelta() (*Cake\I18n\Number method*), 801
formatDelta() (*Cake\View\Helper\NumberHelper method*), 358
FormHelper (*classe dans Cake\View\Helper*), 299
FrozenDate (*classe dans Cake\I18n*), 820
FrozenTime (*classe dans Cake\I18n*), 820

G

gc() (*Cake\Cache\Cache method*), 555
gc() (*Cake\Cache\CacheEngine method*), 558
generateUrl() (*Cake\View\Helper\PaginatorHelper method*), 365
get() (*Cake\Datasource\ConnectionManager method*), 391
get() (*Cake\ORM\Entity method*), 443
get() (*Cake\ORM\Table method*), 451
get() (*Cake\ORM\TableRegistry method*), 440
get() (*Cake\Utility\Hash method*), 768
getCrumbList() (*Cake\View\Helper\HtmlHelper method*), 354
getCrumbs() (*Cake\View\Helper\HtmlHelper method*), 354
getData() (*Cake\Http\ServerRequest method*), 201
getMethod() (*Cake\Http\ServerRequest method*), 204
getQuery() (*Cake\Http\ServerRequest method*), 200
getType() (*Cake\Error\Debugger method*), 598
GoneException, 616
greedy star, 173
group() (*Cake\Filesystem\File method*), 764
groupBy() (*Cake\Collection\Collection method*), 747
groupConfigs() (*Cake\Cache\Cache method*), 556

H

h() (*global function*), 826
handle (*Cake\Filesystem\File property*), 763
Hash (*classe dans Cake\Utility*), 767
hash() (*Cake\Utility\Security method*), 674
hasNext() (*Cake\View\Helper\PaginatorHelper method*), 364
hasPage() (*Cake\View\Helper\PaginatorHelper method*), 364
hasPrev() (*Cake\View\Helper\PaginatorHelper method*), 364
Helper (*class*), 384
helpers (*Cake\Controller\Controller property*), 227
hidden() (*Cake\View\Helper\FormHelper method*), 310
highlight() (*Cake\Utility\Text method*), 809
highlight() (*Cake\View\Helper\TextHelper method*), 372
host() (*Cake\Http\ServerRequest method*), 204
HOUR (*global constant*), 828
hour() (*Cake\View\Helper\FormHelper method*), 327
HtmlHelper (*classe dans Cake\View\Helper*), 341

I

i18nFormat() (*Cake\I18n\Time method*), 815
identify() (*méthode AuthComponent*), 232
image() (*Cake\View\Helper\HtmlHelper method*), 345
in() (*Cake\Console\Shell method*), 567
inCakePath() (*Cake\Filesystem\Folder method*), 762
increment() (*Cake\Cache\Cache method*), 555
increment() (*Cake\Cache\CacheEngine method*), 558
indexBy() (*Cake\Collection\Collection method*), 748
Inflector (*classe dans Cake\Utility*), 793
info (*Cake\Filesystem\File property*), 763
info() (*Cake\Filesystem\File method*), 764
info() (*Cake\Log\Log method*), 651
initialize() (*Cake\Console\Shell method*), 572
initialize() (*Cake\ORM\Table method*), 436
inPath() (*Cake\Filesystem\Folder method*), 762
input() (*Cake\Http\ServerRequest method*), 201
insert() (*Cake\Collection\Collection method*), 755
insert() (*Cake\Utility\Hash method*), 768
insert() (*Cake\Utility\Text method*), 807
InternalErrorException, 616
InvalidCsrfTokenException, 615
is() (*Cake\Http\ServerRequest method*), 202
isAbsolute() (*Cake\Filesystem\Folder method*), 762
isAtom() (*méthode RequestHandlerComponent*), 263
isEmpty() (*Cake\Collection\Collection method*), 752
isFieldError() (*Cake\View\Helper\FormHelper method*), 330
isMobile() (*méthode RequestHandlerComponent*), 263
isRss() (*méthode RequestHandlerComponent*), 262
isSlashTerm() (*Cake\Filesystem\Folder method*), 762
isThisMonth() (*Cake\I18n\Time method*), 818
isThisWeek() (*Cake\I18n\Time method*), 818
isThisYear() (*Cake\I18n\Time method*), 818
isWap() (*méthode RequestHandlerComponent*), 263
isWindowsPath() (*Cake\Filesystem\Folder method*), 762
isWithinNext() (*Cake\I18n\Time method*), 819
isXml() (*méthode RequestHandlerComponent*), 262
isYesterday() (*Cake\I18n\Time method*), 818

J

JsonView (*class*), 293

L

label() (*Cake\View\Helper\FormHelper method*), 328
last() (*Cake\Collection\Collection method*), 754
last() (*Cake\View\Helper\PaginatorHelper method*), 364
lastAccess() (*Cake\Filesystem\File method*), 764
lastChange() (*Cake\Filesystem\File method*), 764
levels() (*Cake\Log\Log method*), 651
limitControl() (*Cake\View\Helper\PaginatorHelper method*), 365
link() (*Cake\View\Helper\HtmlHelper method*), 346
listNested() (*Cake\Collection\Collection method*), 751
load() (*Cake\Core\Configure method*), 168
loadComponent() (*Cake\Controller\Controller method*), 226
loadModel() (*Cake\Controller\Controller method*), 225
lock (*Cake\Filesystem\File property*), 763
Log (*classe dans Cake\Log*), 651
log() (*Cake\Error\Debugger method*), 597
log() (*Cake\Log\LogTrait method*), 652
logout() (*méthode AuthComponent*), 242
LOGS (*global constant*), 827
LogTrait (*trait in Cake\Log*), 652

M

map() (*Cake\Collection\Collection method*), 740
map() (*Cake\Database\Type method*), 393
map() (*Cake\Utility\Hash method*), 777
match() (*Cake\Collection\Collection method*), 745
max() (*Cake\Collection\Collection method*), 746
maxDimensions() (*Cake\Utility\Hash method*), 777
md5() (*Cake\Filesystem\File method*), 764
media() (*Cake\View\Helper\HtmlHelper method*), 348
median() (*Cake\Collection\Collection method*), 747
merge() (*Cake\Console\ConsoleOptionParser method*), 580

merge() (*Cake\Utility\Hash method*), 775
mergeDiff() (*Cake\Utility\Hash method*), 778
meridian() (*Cake\View\Helper\FormHelper method*), 328
messages() (*Cake\Filesystem\Folder method*), 762
meta() (*Cake\View\Helper\HtmlHelper method*), 343
MethodNotAllowedException, 615
mime() (*Cake\Filesystem\File method*), 765
min() (*Cake\Collection\Collection method*), 746
MINUTE (*global constant*), 828
minute() (*Cake\View\Helper\FormHelper method*), 327
MissingActionException, 617
MissingBehaviorException, 617
MissingCellException, 617
MissingCellViewException, 617
MissingComponentException, 617
MissingConnectionException, 617
MissingControllerException, 617
MissingDispatcherFilterException, 617
MissingDriverException, 617
MissingElementException, 617
MissingEntityException, 617
MissingExtensionException, 617
MissingHelperException, 616
MissingLayoutException, 616
MissingRouteException, 617
MissingShellException, 617
MissingShellMethodException, 617
MissingTableException, 617
MissingTaskException, 617
MissingTemplateException, 616
MissingViewException, 616
mode (*Cake\Filesystem\Folder property*), 760
MONTH (*global constant*), 828
month() (*Cake\View\Helper\FormHelper method*), 326
move() (*Cake\Filesystem\Folder method*), 762

N

name (*Cake\Filesystem\File property*), 763
name() (*Cake\Filesystem\File method*), 764
namespaceSplit() (*global function*), 826
nest() (*Cake\Collection\Collection method*), 750
nest() (*Cake\Utility\Hash method*), 780
nestedList() (*Cake\View\Helper\HtmlHelper method*), 350
newQuery() (*Cake\Database\Connection method*), 397
next() (*Cake\View\Helper\PaginatorHelper method*), 363
nice() (*Cake\I18n\Time method*), 816
normalize() (*Cake\Utility\Hash method*), 779
normalizePath() (*Cake\Filesystem\Folder method*), 762
NotAcceptableException, 615
notation avec points, 875
NotFoundException, 615
notice() (*Cake\Log\Log method*), 651
NotImplementedException, 616
Number (*classe dans Cake\I18n*), 797
NumberHelper (*classe dans Cake\View\Helper*), 355
numbers() (*Cake\View\Helper\PaginatorHelper method*), 362
numeric() (*Cake\Utility\Hash method*), 776

O

offset() (*Cake\Filesystem\File method*), 764
open() (*Cake\Filesystem\File method*), 764
options() (*Cake\View\Helper\PaginatorHelper method*), 366
ordinal() (*Cake\I18n\Number method*), 801
ordinal() (*Cake\View\Helper\NumberHelper method*), 358
owner() (*Cake\Filesystem\File method*), 764

P

PaaS, 876
paginate() (*Cake\Controller\Controller method*), 226
PaginatorComponent (*classe dans Cake\Controller\Component*), 256
PaginatorHelper (*classe dans Cake\View\Helper*), 359
parseFileSize() (*Cake\Utility\Text method*), 807
passed arguments, 189
password() (*Cake\View\Helper\FormHelper method*), 310
path (*Cake\Filesystem\File property*), 764
path (*Cake\Filesystem\Folder property*), 760
path() (*Cake\Core\App method*), 736
path() (*Cake\Core\Plugin method*), 736
perms() (*Cake\Filesystem\File method*), 764
PersistenceFailedException, 617
php:attr (*directive*), 131
php:attr (*role*), 132
php:class (*directive*), 130
php:class (*role*), 131
php:const (*directive*), 130
php:const (*role*), 131
php:exc (*role*), 132
php:exception (*directive*), 130
php:func (*role*), 131
php:function (*directive*), 130
php:global (*directive*), 130
php:global (*role*), 131
php:meth (*role*), 131
php:method (*directive*), 130
php:staticmethod (*directive*), 131
pj() (*global function*), 827

plugin routing, 181
plugin() (*Cake\Routing\Router method*), **181**
pluginSplit() (*global function*), **826**
pluralize() (*Cake\Utility\Inflector method*), **794**
postButton() (*Cake\View\Helper\FormHelper method*), **332**
postLink() (*Cake\View\Helper\FormHelper method*), **333**
pr() (*global function*), **827**
precision() (*Cake\I18n\Number method*), **799**
precision() (*Cake\View\Helper\NumberHelper method*), **356**
prefers() (*méthode RequestHandlerComponent*), **264**
prefix routing, 180
prefix() (*Cake\Routing\Router method*), **180**
prepare() (*Cake\Filesystem\File method*), **764**
prev() (*Cake\View\Helper\PaginatorHelper method*), **363**
PrivateActionException, **617**
propriétés, **876**
pwd() (*Cake\Filesystem\File method*), **765**
pwd() (*Cake\Filesystem\Folder method*), **763**

Q

Query (*classe dans Cake\ORM*), **402**
query() (*Cake\Database\Connection method*), **397**

R

radio() (*Cake\View\Helper\FormHelper method*), **315**
randomBytes() (*Cake\Utility\Security method*), **675**
read() (*Cake\Cache\Cache method*), **553**
read() (*Cake\Cache\CacheEngine method*), **557**
read() (*Cake\Controller\Component\CookieComponent method*), **248**
read() (*Cake\Core\Configure method*), **166**
read() (*Cake\Filesystem\File method*), **765**
read() (*Cake\Filesystem\Folder method*), **763**
read() (*Cake\View\Helper\SessionHelper method*), **371**
read() (*méthode Session*), **683**
readable() (*Cake\Filesystem\File method*), **765**
readMany() (*Cake\Cache\Cache method*), **554**
readOrFail() (*Cake\Core\Configure method*), **166**
realpath() (*Cake\Filesystem\Folder method*), **763**
RecordNotFoundException, **617**
redirect() (*Cake\Controller\Controller method*), **224**
redirectUrl() (*méthode AuthComponent*), **233**
reduce() (*Cake\Collection\Collection method*), **746**
reduce() (*Cake\Utility\Hash method*), **777**
ref (*role*), 129
referer() (*Cake\Http\ServerRequest method*), **205**
reject() (*Cake\Collection\Collection method*), **744**
remember() (*Cake\Cache\Cache method*), **553**
remove() (*Cake\Utility\Hash method*), **769**
render() (*Cake\Controller\Controller method*), **223**
renderAs() (*méthode RequestHandlerComponent*), **265**
renew() (*méthode Session*), **684**
replaceText() (*Cake\Filesystem\File method*), **765**
RequestHandlerComponent (*class*), **261**
requireAuth() (*méthode SecurityComponent*), **253**
requireSecure() (*méthode SecurityComponent*), **253**
respondAs() (*méthode RequestHandlerComponent*), **265**
Response (*classe dans Cake\Http*), **207**
Response (*classe dans Cake\Http\Client*), **789**
responseHeader() (*Cake\Core\Exception\Exception method*), **618**
responseType() (*méthode RequestHandlerComponent*), **265**
restore() (*Cake\Core\Configure method*), **169**
RFC
  RFC 2606, 145
  RFC 2616#section-10.4, 616
  RFC 2616#section-10.5, 616
  RFC 4122, 807
ROOT (*global constant*), **827**
Router (*classe dans Cake\Routing*), **171**
routes.php, **876**
routes.php, 171
RssHelper (*classe dans Cake\View\Helper*), **368**
rules() (*Cake\Utility\Inflector method*), **796**

S

safe() (*Cake\Filesystem\File method*), **765**
sample() (*Cake\Collection\Collection method*), **753**
save() (*Cake\ORM\Table method*), **497**
saveMany() (*Cake\ORM\Table method*), **505**
saveOrFail() (*Cake\ORM\Table method*), **505**
script() (*Cake\View\Helper\HtmlHelper method*), **348**
scriptBlock() (*Cake\View\Helper\HtmlHelper method*), **349**
scriptEnd() (*Cake\View\Helper\HtmlHelper method*), **350**
scriptStart() (*Cake\View\Helper\HtmlHelper method*), **350**
SECOND (*global constant*), **828**
secure() (*Cake\View\Helper\FormHelper method*), **341**
Security (*classe dans Cake\Utility*), **673**
SecurityComponent (*class*), **252**
select() (*Cake\View\Helper\FormHelper method*), **316**
ServerRequest (*classe dans Cake\Http*), **199**
ServiceUnavailableException, **616**
Session (*class*), **682**
SessionHelper (*classe dans Cake\View\Helper*), **371**

set() (*Cake\Controller\Controller method*), 222
set() (*Cake\ORM\Entity method*), 444
set() (*Cake\View\View method*), 275
setAction() (*Cake\Controller\Controller method*), 225
setDescription() (*Cake\Console\ConsoleOptionParser method*), 573
setEpilog() (*Cake\Console\ConsoleOptionParser method*), 574
setHelpAlias() (*Cake\Console\ConsoleOptionParser method*), 574
setJsonEncodeFormat() (*Cake\I18n\Time method*), 815
setTemplates() (*Cake\View\Helper\HtmlHelper method*), 353
setTemplates() (*Cake\View\Helper\PaginatorHelper method*), 360
setUser() (*méthode AuthComponent*), 241
Shell (*classe dans Cake\Console*), 563
shuffle() (*Cake\Collection\Collection method*), 753
singularize() (*Cake\Utility\Inflector method*), 794
size() (*Cake\Filesystem\File method*), 765
skip() (*Cake\Collection\Collection method*), 754
slashTerm() (*Cake\Filesystem\Folder method*), 763
slug() (*Cake\Utility\Inflector method*), 796
slug() (*Cake\Utility\Text method*), 806
some() (*Cake\Collection\Collection method*), 745
sort (*Cake\Filesystem\Folder property*), 760
sort() (*Cake\Utility\Hash method*), 778
sort() (*Cake\View\Helper\PaginatorHelper method*), 361
sortBy() (*Cake\Collection\Collection method*), 749
sortDir() (*Cake\View\Helper\PaginatorHelper method*), 362
sortKey() (*Cake\View\Helper\PaginatorHelper method*), 362
stackTrace() (*global function*), 595
startup() (*Cake\Console\Shell method*), 572
stopWhen() (*Cake\Collection\Collection method*), 742
store() (*Cake\Core\Configure method*), 168
stripLinks() (*Cake\Utility\Text method*), 809
stripLinks() (*Cake\View\Helper\TextHelper method*), 373
style() (*Cake\View\Helper\HtmlHelper method*), 343
subdomains() (*Cake\Http\ServerRequest method*), 204
submit() (*Cake\View\Helper\FormHelper method*), 330
sumOf() (*Cake\Collection\Collection method*), 746
syntaxe de plugin, 876

T

Table (*classe dans Cake\ORM*), 451
tableau de routing, 876
tableCells() (*Cake\View\Helper\HtmlHelper method*), 352
tableHeaders() (*Cake\View\Helper\HtmlHelper method*), 351
tableize() (*Cake\Utility\Inflector method*), 795
TableRegistry (*classe dans Cake\ORM*), 440
TableSchema (*classe dans Cake\Database\Schema*), 543
tail() (*Cake\Utility\Text method*), 810
tail() (*Cake\View\Helper\TextHelper method*), 374
take() (*Cake\Collection\Collection method*), 753
TESTS (*global constant*), 827
Text (*classe dans Cake\Utility*), 805
text() (*Cake\View\Helper\FormHelper method*), 310
textarea() (*Cake\View\Helper\FormHelper method*), 311
TextHelper (*classe dans Cake\View\Helper*), 371
through() (*Cake\Collection\Collection method*), 756
Time (*classe dans Cake\I18n*), 813
time() (*Cake\View\Helper\FormHelper method*), 324
TIME_START (*global constant*), 828
timeAgoInWords() (*Cake\I18n\Time method*), 817
TimeHelper (*classe dans Cake\View\Helper*), 376
TimestampBehavior (*classe dans Cake\ORM\Behavior*), 522
TMP (*global constant*), 827
tokenize() (*Cake\Utility\Text method*), 807
toList() (*Cake\Utility\Text method*), 811
toList() (*Cake\View\Helper\TextHelper method*), 375
toPercentage() (*Cake\I18n\Number method*), 799
toPercentage() (*Cake\View\Helper\NumberHelper method*), 356
toQuarter() (*Cake\I18n\Time method*), 818
toReadableSize() (*Cake\I18n\Number method*), 799
toReadableSize() (*Cake\View\Helper\NumberHelper method*), 357
total() (*Cake\View\Helper\PaginatorHelper method*), 364
trace() (*Cake\Error\Debugger method*), 597
trailing star, 173
transactional() (*Cake\Database\Connection method*), 398
TranslateBehavior (*classe dans Cake\ORM\Behavior*), 524
transliterate() (*Cake\Utility\Text method*), 806
transpose() (*Cake\Collection\Collection method*), 753
tree() (*Cake\Filesystem\Folder method*), 763
TreeBehavior (*classe dans Cake\ORM\Behavior*), 532
truncate() (*Cake\Utility\Text method*), 809
truncate() (*Cake\View\Helper\TextHelper method*), 373

Type (*classe dans Cake\Database*), 392

U

UnauthorizedException, 615
underscore() (*Cake\Utility\Inflector method*), 795
unfold() (*Cake\Collection\Collection method*), 743
unlockField() (*Cake\View\Helper\FormHelper method*), 341
updateAll() (*Cake\ORM\Table method*), 506
url() (*Cake\Routing\Router method*), 190
UrlHelper (*classe dans Cake\View\Helper*), 376
user() (*méthode AuthComponent*), 242
uuid() (*Cake\Utility\Text method*), 807

V

Validator (*classe dans Cake\Validation*), 725
variable() (*Cake\Utility\Inflector method*), 796
vendor/cakephp-plugins.php, 658
View (*classe dans Cake\View*), 273

W

warning() (*Cake\Log\Log method*), 651
wasWithinLast() (*Cake\I18n\Time method*), 819
WEEK (*global constant*), 828
withBody() (*Cake\Http\Response method*), 210
withCache() (*Cake\Http\Response method*), 211
withCharset() (*Cake\Http\Response method*), 211
withDisabledCache() (*Cake\Http\Response method*), 211
withEtag() (*Cake\Http\Response method*), 213
withExpires() (*Cake\Http\Response method*), 212
withFile() (*Cake\Http\Response method*), 207
withHeader() (*Cake\Http\Response method*), 209
withModified() (*Cake\Http\Response method*), 213
withSharable() (*Cake\Http\Response method*), 212
withStringBody() (*Cake\Http\Response method*), 209
withType() (*Cake\Http\Response method*), 207
withVary() (*Cake\Http\Response method*), 214
wrap() (*Cake\Utility\Text method*), 808
wrapBlock() (*Cake\Utility\Text method*), 808
writable() (*Cake\Filesystem\File method*), 765
write() (*Cake\Cache\Cache method*), 552
write() (*Cake\Cache\CacheEngine method*), 557
write() (*Cake\Controller\Component\CookieComponent method*), 248
write() (*Cake\Core\Configure method*), 165
write() (*Cake\Filesystem\File method*), 765
write() (*Cake\Log\Log method*), 651
write() (*méthode Session*), 683
writeMany() (*Cake\Cache\Cache method*), 552
WWW_ROOT (*global constant*), 827

X

Xml (*classe dans Cake\Utility*), 821
XmlView (*class*), 293

Y

YEAR (*global constant*), 828
year() (*Cake\View\Helper\FormHelper method*), 325

Z

zip() (*Cake\Collection\Collection method*), 748