CakePHP Cookbook Documentation

Release 5.x

Cake Software Foundation

May 06, 2024

Contents

CHAPTER 1

CakePHP at a Glance

CakePHP is designed to make common web-development tasks simple, and easy. By providing an all-in-one toolbox to get you started the various parts of CakePHP work well together or separately.

The goal of this overview is to introduce the general concepts in CakePHP, and give you a quick overview of how those concepts are implemented in CakePHP. If you are itching to get started on a project, you can *start with the tutorial*, or dive into the docs.

Conventions Over Configuration

CakePHP provides a basic organizational structure that covers class names, filenames, database table names, and other conventions. While the conventions take some time to learn, by following the conventions CakePHP provides you can avoid needless configuration and make a uniform application structure that makes working with various projects simple. The *conventions chapter* covers the various conventions that CakePHP uses.

The Model Layer

The Model layer represents the part of your application that implements the business logic. It is responsible for retrieving data and converting it into the primary meaningful concepts in your application. This includes processing, validating, associating or other tasks related to handling data.

In the case of a social network, the Model layer would take care of tasks such as saving the user data, saving friends' associations, storing and retrieving user photos, finding suggestions for new friends, etc. The model objects can be thought of as "Friend", "User", "Comment", or "Photo". If we wanted to load some data from our `users` table we could do:

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

(continues on next page)

(continued from previous page)

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

You may notice that we didn't have to write any code before we could start working with our data. By using conventions, CakePHP will use standard classes for table and entity classes that have not yet been defined.

If we wanted to make a new user and save it (with validation) we would do something like:

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

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

The View Layer

The View layer renders a presentation of modeled data. Being separate from the Model objects, it is responsible for using the information it has available to produce any presentational interface your application might need.

For example, the view could use model data to render an HTML view template containing it, or a XML formatted result for others to consume:

```
// In a view template file, we'll render an 'element' for each user.
<?php foreach ($users as $user): ?>
    <li class="user">
        <?= $this->element('user_info', ['user' => $user]) ?>
    </li>
<?php endforeach; ?>
```

The View layer provides a number of extension points like *View Templates*, *Elements* and *View Cells* to let you re-use your presentation logic.

The View layer is not only limited to HTML or text representation of the data. It can be used to deliver common data formats like JSON, XML, and through a pluggable architecture any other format you may need, such as CSV.

The Controller Layer

The Controller layer handles requests from users. It is responsible for rendering a response with the aid of both the Model and the View layers.

A controller can be seen as a manager that ensures that all resources needed for completing a task are delegated to the correct workers. It waits for petitions from clients, checks their validity according to authentication or authorization rules, delegates data fetching or processing to the model, selects the type of presentational data that the clients are accepting, and finally delegates the rendering process to the View layer. An example of a user registration controller would be:

```
public function add()
{
    $user = $this->Users->newEmptyEntity();
    if ($this->request->is('post')) {
        $user = $this->Users->patchEntity($user, $this->request->getData());
        if ($this->Users->save($user, ['validate' => 'registration'])) {
            $this->Flash->success(__('You are now registered.'));
        } else {
            $this->Flash->error(__('There were some problems.'));
        }
    }
    $this->set('user', $user);
}
```

You may notice that we never explicitly rendered a view. CakePHP's conventions will take care of selecting the right view and rendering it with the view data we prepared with `set()`.

CakePHP Request Cycle

Now that you are familiar with the different layers in CakePHP, lets review how a request cycle works in CakePHP:

The typical CakePHP request cycle starts with a user requesting a page or resource in your application. At a high level each request goes through the following steps:

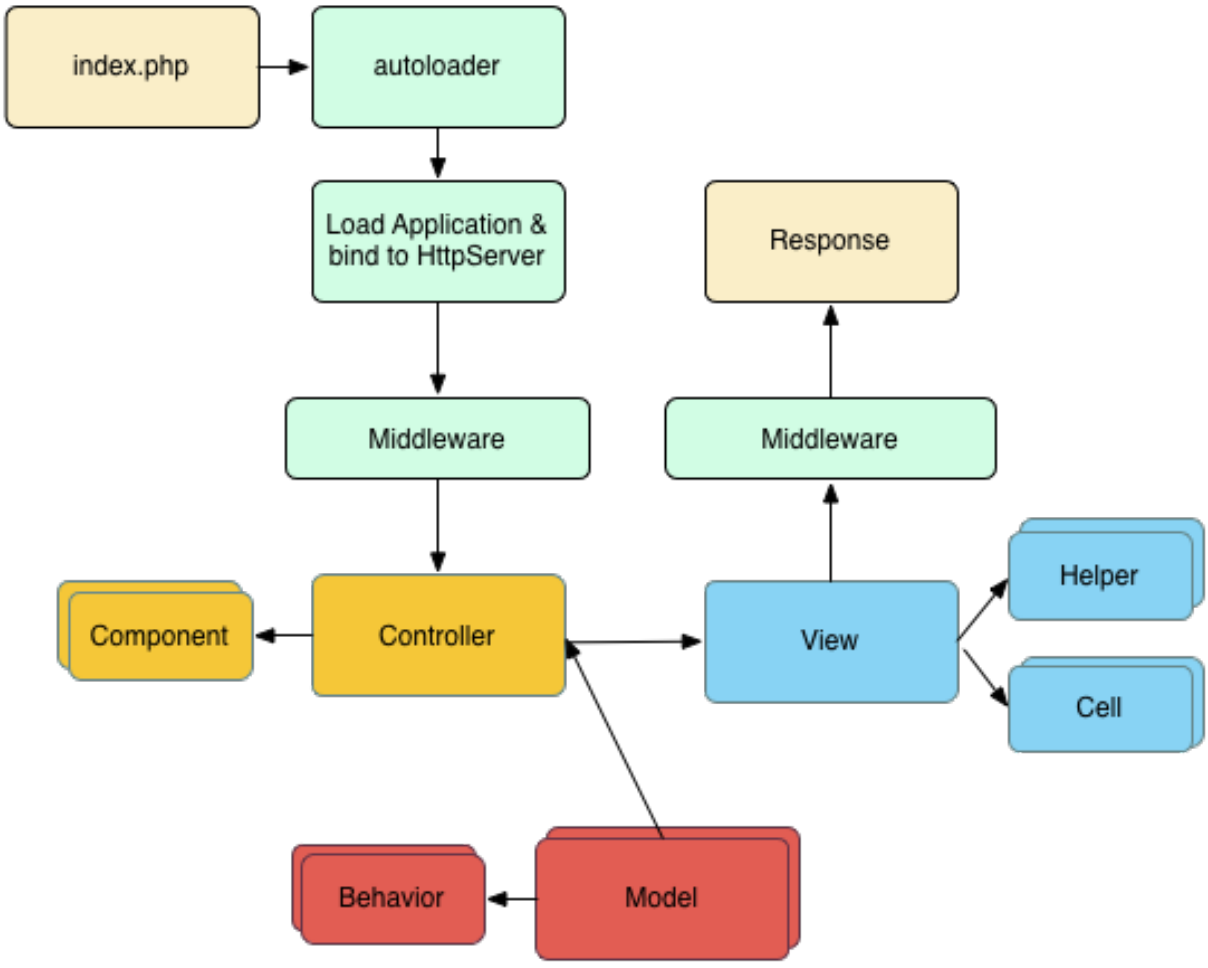
1. The webserver rewrite rules direct the request to **webroot/index.php**.
2. Your Application is loaded and bound to an `HttpServer`.
3. Your application's middleware is initialized.
4. A request and response is dispatched through the PSR-7 Middleware that your application uses. Typically this includes error trapping and routing.
5. If no response is returned from the middleware and the request contains routing information, a controller & action are selected.
6. The controller's action is called and the controller interacts with the required Models and Components.
7. The controller delegates response creation to the View to generate the output resulting from the model data.
8. The view uses Helpers and Cells to generate the response body and headers.
9. The response is sent back out through the /controllers/middleware.
10. The `HttpServer` emits the response to the webserver.

Just the Start

Hopefully this quick overview has piqued your interest. Some other great features in CakePHP are:

- A *caching* framework that integrates with Memcached, Redis and other backends.
- Powerful code generation tools so you can start immediately.
- *Integrated testing framework* so you can ensure your code works perfectly.

The next obvious steps are to *download CakePHP*, read the *tutorial and build something awesome*.


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

Additional Reading

Where to Get Help

The Official CakePHP website

https://cakephp.org

The Official CakePHP website is always a great place to visit. It features links to oft-used developer tools, screencasts, donation opportunities, and downloads.

The Cookbook

https://book.cakephp.org

This manual should probably be the first place you go to get answers. As with many other open source projects, we get new folks regularly. Try your best to answer your questions on your own first. Answers may come slower, but will remain longer – and you'll also be lightening our support load. Both the manual and the API have an online component.

The Bakery

https://bakery.cakephp.org

The CakePHP Bakery is a clearing house for all things regarding CakePHP. Check it out for tutorials, case studies, and code examples. Once you're acquainted with CakePHP, log on and share your knowledge with the community and gain instant fame and fortune.

The API

https://api.cakephp.org/

Straight to the point and straight from the core developers, the CakePHP API (Application Programming Interface) is the most comprehensive documentation around for all the nitty gritty details of the internal workings of the framework. It's a straight forward code reference, so bring your propeller hat.

The Test Cases

If you ever feel the information provided in the API is not sufficient, check out the code of the test cases provided with CakePHP. They can serve as practical examples for function and data member usage for a class.

```
tests/TestCase/
```

The IRC Channel

IRC Channels on irc.freenode.net:

- #cakephp – General Discussion
- #cakephp-docs – Documentation
- #cakephp-bakery – Bakery
- #cakephp-fr – French Canal.

If you're stumped, give us a holler in the CakePHP IRC channel. Someone from the development team[4] is usually there, especially during the daylight hours for North and South America users. We'd love to hear from you, whether you need some help, want to find users in your area, or would like to donate your brand new sports car.

Official CakePHP Forum

CakePHP Official Forum[5]

Our official forum where you can ask for help, suggest ideas and have a talk about CakePHP. It's a perfect place for quickly finding answers and help others. Join the CakePHP family by signing up.

Stackoverflow

https://stackoverflow.com/[6]

Tag your questions with `cakephp` and the specific version you are using to enable existing users of stackoverflow to find your questions.

Where to get Help in your Language

Danish

- Danish CakePHP Slack Channel[7]

French

- French CakePHP Community[8]

[4] https://cakephp.org/team
[5] https://discourse.cakephp.org
[6] https://stackoverflow.com/questions/tagged/cakephp/
[7] https://cakesf.slack.com/messages/denmark/
[8] https://cakephp-fr.org

German

- German CakePHP Slack Channel[9]
- German CakePHP Facebook Group[10]

Iranian

- Iranian CakePHP Community[11]

Dutch

- Dutch CakePHP Slack Channel[12]

Japanese

- Japanese CakePHP Slack Channel[13]
- Japanese CakePHP Facebook Group[14]

Portuguese

- Portuguese CakePHP Google Group[15]

Spanish

- Spanish CakePHP Slack Channel[16]
- Spanish CakePHP IRC Channel
- Spanish CakePHP Google Group[17]

CakePHP Conventions

We are big fans of convention over configuration. While it takes a bit of time to learn CakePHP's conventions, you save time in the long run. By following conventions, you get free functionality, and you liberate yourself from the maintenance nightmare of tracking config files. Conventions also make for a very uniform development experience, allowing other developers to jump in and help.

[9] https://cakesf.slack.com/messages/german/
[10] https://www.facebook.com/groups/146324018754907/
[11] https://cakephp.ir
[12] https://cakesf.slack.com/messages/netherlands/
[13] https://cakesf.slack.com/messages/japanese/
[14] https://www.facebook.com/groups/304490963004377/
[15] https://groups.google.com/group/cakephp-pt
[16] https://cakesf.slack.com/messages/spanish/
[17] https://groups.google.com/group/cakephp-esp

Controller Conventions

Controller class names are plural, CamelCased, and end in `Controller`. `UsersController` and `MenuLinksController` are both examples of conventional controller names.

Public methods on Controllers are often exposed as 'actions' accessible through a web browser. They are camelBacked. For example the `/users/view-me` maps to the `viewMe()` method of the `UsersController` out of the box (if one uses default dashed inflection in routing). Protected or private methods cannot be accessed with routing.

URL Considerations for Controller Names

As you've just seen, single word controllers map to a simple lower case URL path. For example, `UsersController` (which would be defined in the file name **UsersController.php**) is accessed from `http://example.com/users`.

While you can route multiple word controllers in any way you like, the convention is that your URLs are lowercase and dashed using the `DashedRoute` class, therefore `/menu-links/view-all` is the correct form to access the `MenuLinksController::viewAll()` action.

When you create links using `this->Html->link()`, you can use the following conventions for the url array:

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

For more information on CakePHP URLs and parameter handling, see *Connecting Routes*.

File and Class Name Conventions

In general, filenames match the class names, and follow the PSR-4 standard for autoloading. The following are some examples of class names and their filenames:

- The Controller class `LatestArticlesController` would be found in a file named **LatestArticlesController.php**
- The Component class `MyHandyComponent` would be found in a file named **MyHandyComponent.php**
- The Table class `OptionValuesTable` would be found in a file named **OptionValuesTable.php**.
- The Entity class `OptionValue` would be found in a file named **OptionValue.php**.
- The Behavior class `EspeciallyFunkableBehavior` would be found in a file named **EspeciallyFunkableBehavior.php**
- The View class `SuperSimpleView` would be found in a file named **SuperSimpleView.php**
- The Helper class `BestEverHelper` would be found in a file named **BestEverHelper.php**

Each file would be located in the appropriate folder/namespace in your app folder.

Database Conventions

Table names corresponding to CakePHP models are plural and underscored. For example `users`, `menu_links`, and `user_favorite_pages` respectively. Table name whose name contains multiple words should only pluralize the last word, for example, `menu_links`.

Column names with two or more words are underscored, for example, `first_name`.

Foreign keys in hasMany, belongsTo/hasOne relationships are recognized by default as the (singular) name of the related table followed by `_id`. So if Users hasMany Articles, the `articles` table will refer to the `users` table via a `user_id` foreign key. For a table like `menu_links` whose name contains multiple words, the foreign key would be `menu_link_id`.

Join (or "junction") tables are used in BelongsToMany relationships between models. These should be named for the tables they connect. The names should be pluralized and sorted alphabetically: `articles_tags`, not `tags_articles` or `article_tags`. *The bake command will not work if this convention is not followed.* If the junction table holds any data other than the linking foreign keys, you should create a concrete entity/table class for the table.

In addition to using an auto-incrementing integer as primary keys, you can also use UUID columns. CakePHP will create UUID values automatically using (*Cake\Utility\Text::uuid()*) whenever you save new records using the `Table::save()` method.

Model Conventions

Table class names are plural, CamelCased and end in `Table`. `UsersTable`, `MenuLinksTable`, and `UserFavoritePagesTable` are all examples of table class names matching the `users`, `menu_links` and `user_favorite_pages` tables respectively.

Entity class names are singular CamelCased and have no suffix. `User`, `MenuLink`, and `UserFavoritePage` are all examples of entity names matching the `users`, `menu_links` and `user_favorite_pages` tables respectively.

Enum class names should use a `{Entity}{Column}` convention, and enum cases should use CamelCased names.

View Conventions

View template files are named after the controller functions they display, in an underscored form. The `viewAll()` function of the `ArticlesController` class will look for a view template in **templates/Articles/view_all.php**.

The basic pattern is **templates/Controller/underscored_function_name.php**.

Note: By default CakePHP uses English inflections. If you have database tables/columns that use another language, you will need to add inflection rules (from singular to plural and vice-versa). You can use *Cake\Utility\Inflector* to define your custom inflection rules. See the documentation about *Inflector* for more information.

Plugins Conventions

It is useful to prefix a CakePHP plugin with "cakephp-" in the package name. This makes the name semantically related on the framework it depends on.

Do **not** use the CakePHP namespace (cakephp) as vendor name as this is reserved to CakePHP owned plugins. The convention is to use lowercase letters and dashes as separator:

```
// Bad
cakephp/foo-bar

// Good
your-name/cakephp-foo-bar
```

See awesome list recommendations[18] for details.

Summarized

By naming the pieces of your application using CakePHP conventions, you gain functionality without the hassle and maintenance tethers of configuration. Here's a final example that ties the conventions together:

- Database table: "articles", "menu_links"
- Table class: `ArticlesTable`, found at **src/Model/Table/ArticlesTable.php**
- Entity class: `Article`, found at **src/Model/Entity/Article.php**
- Controller class: `ArticlesController`, found at **src/Controller/ArticlesController.php**
- View template, found at **templates/Articles/index.php**

Using these conventions, CakePHP knows that a request to `http://example.com/articles` maps to a call on the `index()` method of the `ArticlesController`, where the `Articles` model is automatically available. None of these relationships have been configured by any means other than by creating classes and files that you'd need to create anyway.

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

Example	articles	menu_links	
Database Table	articles	menu_links	Table names corresponding to CakePHP models are plural and underscored.
File	ArticlesController.php	MenuLinksController.php	
Table	ArticlesTable.php	MenuLinksTable.pl	Table class names are plural, CamelCased and end in Table
Entity	Article.php	MenuLink.php	Entity class names are singular, CamelCased: Article and MenuLink
Class	ArticlesController	MenuLinksController	
Controller	ArticlesController	MenuLinksController	Plural, CamelCased, end in Controller
Templates	Articles/index.php Articles/add.php Articles/get_list.php	MenuLinks/index.p MenuLinks/add.ph MenuLinks/get_list	View template files are named after the controller functions they display, in an underscored form
Behavior	ArticlesBehavior.php	MenuLinksBehavior.php	
View	ArticlesView.php	MenuLinksView.ph	
Helper	ArticlesHelper.php	MenuLinksHelper.p	
Component	ArticlesComponent.php	MenuLinksComponent.php	
Plugin	Bad: cakephp/articles Good: you/cakephp-articles	cakephp/menu-links you/cakephp-menu-links	Useful to prefix a CakePHP plugin with "cakephp-" in the package name. Do not use the CakePHP namespace (cakephp) as vendor name as this is reserved to CakePHP owned plugins. The convention is to use lowercase letters and dashes as separator.
Each file would be located in the appropriate folder/namespace in your app folder.			

Database Convention Summary

Foreign keys hasMany belongsTo/ hasOne BelongsToMany	Relationships are recognized by default as the (singular) name of the related table followed by `_id`. Users hasMany Articles, `articles` table will refer to the `users` table via a `user_id` foreign key.
Multiple Words	`menu_links` whose name contains multiple words, the foreign key would be `menu_link_id`.
Auto Increment	In addition to using an auto-incrementing integer as primary keys, you can also use UUID columns. CakePHP will create UUID values automatically using (*Cake\Utility\Text::uuid()*) whenever you save new records using the `Table::save()` method.
Join tables	Should be named after the model tables they will join or the bake command won't work, arranged in alphabetical order (`articles_tags` rather than `tags_articles`). Additional columns on the junction table you should create a separate entity/table class for that table.

Now that you've been introduced to CakePHP's fundamentals, you might try a run through the *Content Management Tutorial* to see how things fit together.

CakePHP Folder Structure

After you've downloaded the CakePHP application skeleton, there are a few top level folders you should see:

- The *bin* folder holds the Cake console executables.
- The *config* folder holds the *Configuration* files CakePHP uses. Database connection details, bootstrapping, core configuration files and more should be stored here.
- The *plugins* folder is where the *Plugins* your application uses are stored.
- The *logs* folder normally contains your log files, depending on your log configuration.
- The *src* folder will be where your application's source files will be placed.
- The *templates* folder has presentational files placed here: elements, error pages, layouts, and view template files.
- The *resources* folder has sub folder for various types of resource files. The *locales* sub folder stores language files for internationalization.
- The *tests* folder will be where you put the test cases for your application.
- The *tmp* folder is where CakePHP stores temporary data. The actual data it stores depends on how you have CakePHP configured, but this folder is usually used to store translation messages, model descriptions and sometimes session information.
- The *vendor* folder is where CakePHP and other application dependencies will be installed by Composer[19]. Editing these files is not advised, as Composer will overwrite your changes next time you update.
- The *webroot* directory is the public document root of your application. It contains all the files you want to be publicly reachable.

  Make sure that the *tmp* and *logs* folders exist and are writable, otherwise the performance of your application will be severely impacted. In debug mode, CakePHP will warn you, if these directories are not writable.

[19] https://getcomposer.org

The src Folder

CakePHP's *src* folder is where you will do most of your application development. Let's look a little closer at the folders inside *src*.

Command
: Contains your application's console commands. See *Command Objects* to learn more.

Console
: Contains the installation script executed by Composer.

Controller
: Contains your application's *Controllers* and their components.

Middleware
: Stores any /controllers/middleware for your application.

Model
: Contains your application's tables, entities and behaviors.

View
: Presentational classes are placed here: views, cells, helpers.

Note: The folder `Command` is not present by default. You can add it when you need it.

CHAPTER 2

Quick Start Guide

The best way to experience and learn CakePHP is to sit down and build something. To start off we'll build a simple Content Management application.

Content Management Tutorial

This tutorial will walk you through the creation of a simple CMS (Content Management System) application. To start with, we'll be installing CakePHP, creating our database, and building simple article management.

Here's what you'll need:

1. A database server. We're going to be using MySQL server in this tutorial. You'll need to know enough about SQL in order to create a database, and run SQL snippets from the tutorial. CakePHP will handle building all the queries your application needs. Since we're using MySQL, also make sure that you have `pdo_mysql` enabled in PHP.
2. Basic PHP knowledge.

Before starting you should make sure that you're using a supported PHP version:

```
php -v
```

You should at least have got installed PHP 8.1 (CLI) or higher. Your webserver's PHP version must also be of 8.1 or higher, and should be the same version your command line interface (CLI) PHP is.

Getting CakePHP

The easiest way to install CakePHP is to use Composer. Composer is a simple way of installing CakePHP from your terminal or command line prompt. First, you'll need to download and install Composer if you haven't done so already. If you have cURL installed, run the following:

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

Or, you can download `composer.phar` from the Composer website[20].

Then simply type the following line in your terminal from your installation directory to install the CakePHP application skeleton in the **cms** directory of the current working directory:

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

If you downloaded and ran the Composer Windows Installer[21], then type the following line in your terminal from your installation directory (ie. C:\wamp\www\dev):

```
composer self-update && composer create-project --prefer-dist cakephp/app:5.* cms
```

The advantage to using Composer is that it will automatically complete some important set up tasks, such as setting the correct file permissions and creating your **config/app.php** file for you.

There are other ways to install CakePHP. If you cannot or don't want to use Composer, check out the *Installation* section.

Regardless of how you downloaded and installed CakePHP, once your set up is completed, your directory setup should look like the following, though other files may also be present:

```
cms/
  bin/
  config/
  plugins/
  resources/
  src/
  templates/
  tests/
  tmp/
  vendor/
  webroot/
  composer.json
  index.php
  README.md
```

Now might be a good time to learn a bit about how CakePHP's directory structure works: check out the *CakePHP Folder Structure* section.

If you get lost during this tutorial, you can see the finished result on GitHub[22].

Tip: The `bin/cake` console utility can build most of the classes and data tables in this tutorial automatically. However, we recommend following along with the manual code examples to understand how the pieces fit together and how to add your application logic.

---

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

Checking our Installation

We can quickly check that our installation is correct, by checking the default home page. Before you can do that, you'll need to start the development server:

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

bin/cake server
```

Note: For Windows, the command needs to be `bin\cake server` (note the backslash).

This will start PHP's built-in webserver on port 8765. Open up **http://localhost:8765** in your web browser to see the welcome page. All the bullet points should be green chef hats other than CakePHP being able to connect to your database. If not, you may need to install additional PHP extensions, or set directory permissions.

Next, we will build our *Database*.

CMS Tutorial - Creating the Database

Now that we have CakePHP installed, let's set up the database for our CMS application. If you haven't already done so, create an empty database for use in this tutorial, with the name of your choice such as `cake_cms`. If you are using MySQL/MariaDB, you can execute the following SQL to create the necessary tables:

```
CREATE DATABASE cake_cms;

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),
```

(continues on next page)

(continued from previous page)

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

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

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

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

If you are using PostgreSQL, connect to the `cake_cms` database and execute the following SQL instead:

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

You may have noticed that the `articles_tags` table uses a composite primary key. CakePHP supports composite primary keys almost everywhere, allowing you to have simpler schemas that don't require additional `id` columns.

The table and column names we used were not arbitrary. By using CakePHP's *naming conventions*, we can leverage CakePHP more effectively and avoid needing to configure the framework. While CakePHP is flexible enough to accommodate almost any database schema, adhering to the conventions will save you time as you can leverage the convention-based defaults CakePHP provides.

Database Configuration

Next, let's tell CakePHP where our database is and how to connect to it. Replace the values in the `Datasources.default` array in your **config/app_local.php** file with those that apply to your setup. A sample completed configuration array might look something like the following:

```
<?php
// config/app_local.php
return [
    // More configuration above.
    'Datasources' => [
        'default' => [
            'host' => 'localhost',
            'username' => 'cakephp',
            'password' => 'AngelF00dC4k3~',
            'database' => 'cake_cms',
            'url' => env('DATABASE_URL', null),
        ],
    ],
    // More configuration below.
];
```

Once you've saved your **config/app_local.php** file, you should see that the 'CakePHP is able to connect to the database' section has a green chef hat.

Note: The file **config/app_local.php** is a local override of the file **config/app.php** used to configure your development environment quickly.

Migrations

The SQL statements to create the tables for this tutorial can also be generated using the Migrations Plugin. Migrations provide a platform-independent way to run queries so the subtle differences between MySQL, PostgreSQL, SQLite, etc. don't become obstacles.

```
bin/cake bake migration CreateUsers email:string password:string created modified
bin/cake bake migration CreateArticles user_id:integer title:string␣
↪slug:string[191]:unique body:text published:boolean created modified
bin/cake bake migration CreateTags title:string[191]:unique created modified
bin/cake bake migration CreateArticlesTags article_id:integer:primary tag_
↪id:integer:primary created modified
```

Note: Some adjustments to the generated code might be necessary. For example, the composite primary key on `articles_tags` will be set to auto-increment both columns:

```
$table->addColumn('article_id', 'integer', [
    'autoIncrement' => true,
    'default' => null,
    'limit' => 11,
    'null' => false,
]);
$table->addColumn('tag_id', 'integer', [
    'autoIncrement' => true,
    'default' => null,
    'limit' => 11,
    'null' => false,
]);
```

Remove those lines to prevent foreign key problems. Once adjustments are done:

```
bin/cake migrations migrate
```

Likewise, the starter data records can be done with seeds.

```
bin/cake bake seed Users
bin/cake bake seed Articles
```

Fill the seed data above into the new `UsersSeed` and `ArticlesSeed` classes, then:

```
bin/cake migrations seed
```

Read more about building migrations and data seeding: Migrations[23]

With the database built, we can now build *Models*.

[23] https://book.cakephp.org/migrations/4/

CMS Tutorial - Creating the Articles Controller

With our model created, we need a controller for our articles. Controllers in CakePHP handle HTTP requests and execute business logic contained in model methods, to prepare the response. We'll place this new controller in a file called **ArticlesController.php** inside the **src/Controller** directory. Here's what the basic controller should look like:

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

namespace App\Controller;

class ArticlesController extends AppController
{
}
```

Now, let's add an action to our controller. Actions are controller methods that have routes connected to them. For example, when a user requests **www.example.com/articles/index** (which is also the same as **www.example.com/articles**), CakePHP will call the `index` method of your `ArticlesController`. This method should query the model layer, and prepare a response by rendering a Template in the View. The code for that action would look like this:

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

namespace App\Controller;

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

By defining function `index()` in our `ArticlesController`, users can now access the logic there by requesting **www.example.com/articles/index**. Similarly, if we were to define a function called `foobar()`, users would be able to access that at **www.example.com/articles/foobar**. You may be tempted to name your controllers and actions in a way that allows you to obtain specific URLs. Resist that temptation. Instead, follow the *CakePHP Conventions* creating readable, meaningful action names. You can then use *Routing* to connect the URLs you want to the actions you've created.

Our controller action is very simple. It fetches a paginated set of articles from the database, using the Articles Model that is automatically loaded via naming conventions. It then uses `set()` to pass the articles into the Template (which we'll create soon). CakePHP will automatically render the template after our controller action completes.

Create the Article List Template

Now that we have our controller pulling data from the model, and preparing our view context, let's create a view template for our index action.

CakePHP view templates are presentation-flavored PHP code that is inserted inside the application's layout. While we'll be creating HTML here, Views can also generate JSON, CSV or even binary files like PDFs.

A layout is presentation code that is wrapped around a view. Layout files contain common site elements like headers, footers and navigation elements. Your application can have multiple layouts, and you can switch between them, but for now, let's just use the default layout.

CakePHP's template files are stored in **templates** inside a folder named after the controller they correspond to. So we'll have to create a folder named 'Articles' in this case. Add the following code to your application:

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

<h1>Articles</h1>
<table>
    <tr>
        <th>Title</th>
        <th>Created</th>
    </tr>

    <!-- Here is where we iterate through our $articles query object, printing out␣
↪article info -->

    <?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>
```

In the last section we assigned the 'articles' variable to the view using `set()`. Variables passed into the view are available in the view templates as local variables which we used in the above code.

You might have noticed the use of an object called `$this->Html`. This is an instance of the CakePHP *HtmlHelper*. CakePHP comes with a set of view helpers that make tasks like creating links, forms, and pagination buttons. You can learn more about *Helpers* in their chapter, but what's important to note here is that the `link()` method will generate an HTML link with the given link text (the first parameter) and URL (the second parameter).

When specifying URLs in CakePHP, it is recommended that you use arrays or *named routes*. These syntaxes allow you to leverage the reverse routing features CakePHP offers.

At this point, you should be able to point your browser to **http://localhost:8765/articles/index**. You should see your list view, correctly formatted with the title and table listing of the articles.

Create the View Action

If you were to click one of the 'view' links in our Articles list page, you'd see an error page saying that action hasn't been implemented. Lets fix that now:

```
// Add to existing src/Controller/ArticlesController.php file

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

While this is a simple action, we've used some powerful CakePHP features. We start our action off by using `findBySlug()` which is a *Dynamic Finder*. This method allows us to create a basic query that finds articles by a given slug. We then use `firstOrFail()` to either fetch the first record, or throw a `NotFoundException`.

Our action takes a `$slug` parameter, but where does that parameter come from? If a user requests `/articles/view/first-post`, then the value 'first-post' is passed as `$slug` by CakePHP's routing and dispatching layers. If we reload our browser with our new action saved, we'd see another CakePHP error page telling us we're missing a view template; let's fix that.

Create the View Template

Let's create the view for our new 'view' action and place it in **templates/Articles/view.php**

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

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

You can verify that this is working by trying the links at `/articles/index` or manually requesting an article by accessing URLs like `/articles/view/first-post`.

Adding Articles

With the basic read views created, we need to make it possible for new articles to be created. Start by creating an `add()` action in the `ArticlesController`. Our controller should now look like:

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

use App\Controller\AppController;

class ArticlesController extends AppController
{
    public function index()
    {
        $articles = $this->paginate($this->Articles);
```

(continues on next page)

(continued from previous page)

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

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

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

            // 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(__('Your article has been saved.'));
                return $this->redirect(['action' => 'index']);
            }
            $this->Flash->error(__('Unable to add your article.'));
        }
        $this->set('article', $article);
    }
}
```

Note: You need to include the *Flash* component in any controller where you will use it. Often it makes sense to include it in your `AppController`, which is there already for this tutorial.

Here's what the `add()` action does:

- If the HTTP method of the request was POST, try to save the data using the Articles model.
- If for some reason it doesn't save, just render the view. This gives us a chance to show the user validation errors or other warnings.

Every CakePHP request includes a request object which is accessible using `$this->request`. The request object contains information regarding the request that was just received. We use the *Cake\Http\ServerRequest::is()* method to check that the request is a HTTP POST request.

Our POST data is available in `$this->request->getData()`. You can use the `pr()` or `debug()` functions to print it out if you want to see what it looks like. To save our data, we first 'marshal' the POST data into an Article Entity. The Entity is then persisted using the ArticlesTable we created earlier.

After saving our new article we use FlashComponent's `success()` method to set a message into the session. The `success` method is provided using PHP's magic method features[24]. Flash messages will be displayed on the next page after redirecting. In our layout we have `<?= $this->Flash->render() ?>` which displays flash messages and clears the corresponding session variable. Finally, after saving is complete, we use *Cake\Controller\Controller::redirect* to send the user back to the articles list. The param `['action' => 'index']` translates to

[24] https://php.net/manual/en/language.oop5.overloading.php#object.call

URL `/articles` i.e the index action of the `ArticlesController`. You can refer to `Cake\Routing\Router::url()` function on the API[25] to see the formats in which you can specify a URL for various CakePHP functions.

Create Add Template

Here's our add view template:

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

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

We use the FormHelper to generate the opening tag for an HTML form. Here's the HTML that `$this->Form->create()` generates:

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

Because we called `create()` without a URL option, `FormHelper` assumes we want the form to submit back to the current action.

The `$this->Form->control()` method is used to create form elements of the same name. The first parameter tells CakePHP which field they correspond to, and the second parameter allows you to specify a wide array of options - in this case, the number of rows for the textarea. There's a bit of introspection and conventions used here. The `control()` will output different form elements based on the model field specified, and use inflection to generate the label text. You can customize the label, the input or any other aspect of the form controls using options. The `$this->Form->end()` call closes the form.

Now let's go back and update our **templates/Articles/index.php** view to include a new "Add Article" link. Before the `<table>`, add the following line:

```
<?= $this->Html->link('Add Article', ['action' => 'add']) ?>
```

Adding Simple Slug Generation

If we were to save an Article right now, saving would fail as we are not creating a slug attribute, and the column is `NOT NULL`. Slug values are typically a URL-safe version of an article's title. We can use the *beforeSave() callback* of the ORM to populate our slug:

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

use Cake\ORM\Table;
```

(continues on next page)

[25] https://api.cakephp.org

(continued from previous page)

```
// the Text class
use Cake\Utility\Text;
// the EventInterface class
use Cake\Event\EventInterface;

// Add the following method.

public function beforeSave(EventInterface $event, $entity, $options)
{
    if ($entity->isNew() && !$entity->slug) {
        $sluggedTitle = Text::slug($entity->title);
        // trim slug to maximum length defined in schema
        $entity->slug = substr($sluggedTitle, 0, 191);
    }
}
```

This code is simple, and doesn't take into account duplicate slugs. But we'll fix that later on.

Add Edit Action

Our application can now save articles, but we can't edit them. Lets rectify that now. Add the following action to your `ArticlesController`:

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

// Add the following method.

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(__('Your article has been updated.'));
            return $this->redirect(['action' => 'index']);
        }
        $this->Flash->error(__('Unable to update your article.'));
    }

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

This action first ensures that the user has tried to access an existing record. If they haven't passed in an `$slug` parameter, or the article does not exist, a `NotFoundException` will be thrown, and the CakePHP ErrorHandler will render the appropriate error page.

Next the action checks whether the request is either a POST or a PUT request. If it is, then we use the POST/PUT data to update our article entity by using the `patchEntity()` method. Finally, we call `save()`, set the appropriate flash message, and either redirect or display validation errors.

Create Edit Template

The edit template should look like this:

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

<h1>Edit 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(__('Save Article'));
    echo $this->Form->end();
?>
```

This template outputs the edit form (with the values populated), along with any necessary validation error messages.

You can now update your index view with links to edit specific articles:

```
<!-- File: templates/Articles/index.php  (edit links added) -->

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

<!-- Here's where we iterate through our $articles query object, printing out article␣
↪info -->

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

</table>
```

Update Validation Rules for Articles

Up until this point our Articles had no input validation done. Lets fix that by using *a validator*:

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

// add this use statement right below the namespace declaration to import
// the Validator class
use Cake\Validation\Validator;

// Add the following method.
public function validationDefault(Validator $validator): Validator
{
    $validator
        ->notEmptyString('title')
        ->minLength('title', 10)
        ->maxLength('title', 255)

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

    return $validator;
}
```

The `validationDefault()` method tells CakePHP how to validate your data when the `save()` method is called. Here, we've specified that both the title, and body fields must not be empty, and have certain length constraints.

CakePHP's validation engine is powerful and flexible. It provides a suite of frequently used rules for tasks like email addresses, IP addresses etc. and the flexibility for adding your own validation rules. For more information on that setup, check the *Validation* documentation.

Now that your validation rules are in place, use the app to try to add an article with an empty title or body to see how it works. Since we've used the *Cake\View\Helper\FormHelper::control()* method of the FormHelper to create our form elements, our validation error messages will be shown automatically.

Add Delete Action

Next, let's make a way for users to delete articles. Start with a `delete()` action in the `ArticlesController`:

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

// Add the following method.

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

    $article = $this->Articles->findBySlug($slug)->firstOrFail();
    if ($this->Articles->delete($article)) {
        $this->Flash->success(__('The {0} article has been deleted.', $article->title));
        return $this->redirect(['action' => 'index']);
    }
}
```

This logic deletes the article specified by `$slug`, and uses `$this->Flash->success()` to show the user a confirmation message after redirecting them to `/articles`. If the user attempts to delete an article using a GET request, `allowMethod()` will throw an exception. Uncaught exceptions are captured by CakePHP's exception handler, and a nice error page is displayed. There are many built-in *Exceptions* that can be used to indicate the various HTTP errors your application might need to generate.

Warning: Allowing content to be deleted using GET requests is *very* dangerous, as web crawlers could accidentally delete all your content. That is why we used `allowMethod()` in our controller.

Because we're only executing logic and redirecting to another action, this action has no template. You might want to update your index template with links that allow users to delete articles:

```
<!-- File: templates/Articles/index.php  (delete links added) -->

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

<!-- Here's where we iterate through our $articles query object, printing out article␣
↪info -->

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

</table>
```

Using *postLink()* will create a link that uses JavaScript to do a POST request deleting our article.

Note: This view code also uses the `FormHelper` to prompt the user with a JavaScript confirmation dialog before they

attempt to delete an article.

Tip: The `ArticlesController` can also be built with `bake`:

```
/bin/cake bake controller articles
```

However, this does not build the **templates/Articles/*.php** files.

With a basic articles management setup, we'll create the *basic actions for our Tags and Users tables*.

CHAPTER 3

Migration Guides

Migration guides contain information regarding the new features introduced in each version and the migration path between 4.x and 5.x.

5.0 Upgrade Guide

First, check that your application is running on latest CakePHP 4.x version.

Fix Deprecation Warnings

Once your application is running on latest CakePHP 4.x, enable deprecation warnings in **config/app.php**:

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

Now that you can see all the warnings, make sure these are fixed before proceeding with the upgrade.

Some potentially impactful deprecations you should make sure you have addressed are:

- `Table::query()` was deprecated in 4.5.0. Use `selectQuery()`, `updateQuery()`, `insertQuery()` and `deleteQuery()` instead.

Upgrade to PHP 8.1

If you are not running on **PHP 8.1 or higher**, you will need to upgrade PHP before updating CakePHP.

Note: CakePHP 5.0 requires **a minimum of PHP 8.1**.

Use the Upgrade Tool

Note: The upgrade tool only works on applications running on latest CakePHP 4.x. You cannot run the upgrade tool after updating to CakePHP 5.0.

Because CakePHP 5 leverages union types and `mixed`, there are many backwards incompatible changes concerning method signatures and file renames. To help expedite fixing these tedious changes there is an upgrade CLI tool:

```
# Install the upgrade tool
git clone https://github.com/cakephp/upgrade
cd upgrade
git checkout 5.x
composer install --no-dev
```

With the upgrade tool installed you can now run it on your application or plugin:

```
bin/cake upgrade rector --rules cakephp50 <path/to/app/src>
bin/cake upgrade rector --rules chronos3 <path/to/app/src>
```

Update CakePHP Dependency

After applying rector refactorings you need to upgrade CakePHP, its plugins, PHPUnit and maybe other dependencies in your `composer.json`. This process heavily depends on your application so we recommend you compare your `composer.json` with what is present in cakephp/app[26].

After the version strings are adjusted in your `composer.json` execute `composer update -W` and check its output.

Update app files based upon latest app template

Next, ensure the rest of your application has been updated to be based upon the latest version of cakephp/app[27].

[26] https://github.com/cakephp/app/blob/5.x/composer.json
[27] https://github.com/cakephp/app/blob/5.x/

5.0 Migration Guide

CakePHP 5.0 contains breaking changes, and is not backwards compatible with 4.x releases. Before attempting to upgrade to 5.0, first upgrade to 4.5 and resolve all deprecation warnings.

Refer to the *5.0 Upgrade Guide* for step by step instructions on how to upgrade to 5.0.

Deprecated Features Removed

All methods, properties and functionality that were emitting deprecation warnings as of 4.5 have been removed.

Breaking Changes

In addition to the removal of deprecated features there have been breaking changes made:

Global

- Type declarations were added to all function parameter and returns where possible. These are intended to match the docblock annotations, but include fixes for incorrect annotations.
- Type declarations were added to all class properties where possible. These also include some fixes for incorrect annotations.
- The `SECOND`, `MINUTE`, `HOUR`, `DAY`, `WEEK`, `MONTH`, `YEAR` constants were removed.
- **Use of `#[\AllowDynamicProperties]` removed everywhere. It was used for the following classes:**
  - `Command/Command`
  - `Console/Shell`
  - `Controller/Component`
  - `Controller/Controller`
  - `Mailer/Mailer`
  - `View/Cell`
  - `View/Helper`
  - `View/View`
- **The supported database engine versions were updated:**
  - MySQL (5.7 or higher)
  - MariaDB (10.1 or higher)
  - PostgreSQL (9.6 or higher)
  - Microsoft SQL Server (2012 or higher)
  - SQLite 3 (3.16 or higher)

Auth

- *Auth* has been removed. Use the cakephp/authentication[28] and cakephp/authorization[29] plugins instead.

Cache

- The `Wincache` engine was removed. The wincache extension is not supported on PHP 8.

Collection

- `combine()` now throws an exception if the key path or group path doesn't exist or contains a null value. This matches the behavior of `indexBy()` and `groupBy()`.

Console

- `BaseCommand::__construct()` was removed.
- `ConsoleIntegrationTestTrait::useCommandRunner()` was removed since it's no longer needed.
- `Shell` has been removed and should be replaced with Command[30]
- `ConsoleOptionParser::addSubcommand()` was removed alongside the removal of `Shell`. Subcommands should be replaced with `Command` classes that implement `Command::defaultName()` to define the necessary command name.
- `BaseCommand` now emits `Command.beforeExecute` and `Command.afterExecute` events around the command's `execute()` method being invoked by the framework.

Connection

- `Connection::prepare()` has been removed. You can use `Connection::execute()` instead to execute a SQL query by specifing the SQL string, params and types in a single call.
- `Connection::enableQueryLogging()` has been removed. If you haven't enabled logging through the connection config then you can later set the logger instance for the driver to enable query logging `$connection->getDriver()->setLogger()`.

Controller

- The method signature for `Controller::__construct()` has changed. So you need to adjust your code accordingly if you are overriding the constructor.
- After loading components are no longer set as dynamic properties. Instead `Controller` uses `__get()` to provide property access to components. This change can impact applications that use `property_exists()` on components.
- The components' `Controller.shutdown` event callback has been renamed from `shutdown` to `afterFilter` to match the controller one. This makes the callbacks more consistent.
- `PaginatorComponent` has been removed and should be replaced by calling `$this->paginate()` in your controller or using `Cake\Datasource\Paging\NumericPaginator` directly

[28] https://book.cakephp.org/authentication/2/en/index.html
[29] https://book.cakephp.org/authorization/2/en/index.html
[30] https://book.cakephp.org/5/en/console-commands/commands.html

- `RequestHandlerComponent` has been removed. See the 4.4 migration[31] guide for how to upgrade
- `SecurityComponent` has been removed. Use `FormProtectionComponent` for form tampering protection or `HttpsEnforcerMiddleware` to enforce use of HTTPS for requests instead.
- `Controller::paginate()` no longer accepts query options like `contain` for its `$settings` argument. You should instead use the `finder` option `$this->paginate($this->Articles, ['finder' => 'published'])`. Or you can create required select query before hand and then pass it to `paginate()` `$query = $this->Articles->find()->where(['is_published' => true]); $this->paginate($query);`.

Core

- The function `getTypeName()` has been dropped. Use PHP's `get_debug_type()` instead.
- The dependency on `league/container` was updated to `4.x`. This will require the addition of typehints to your `ServiceProvider` implementations.
- `deprecationWarning()` now has a `$version` parameter.
- The `App.uploadedFilesAsObjects` configuration option has been removed alongside of support for PHP file upload shaped arrays throughout the framework.
- `ClassLoader` has been removed. Use composer to generate autoload files instead.

Database

- The `DateTimeType` and `DateType` now always return immutable objects. Additionally the interface for `Date` objects reflects the `ChronosDate` interface which lacks all of the time related methods that were present in CakePHP 4.x.
- `DateType::setLocaleFormat()` no longer accepts an array.
- `Query` now accepts only `\Closure` parameters instead of `callable`. Callables can be converted to closures using the new first-class array syntax in PHP 8.1.
- `Query::execute()` no longer runs results decorator callbacks. You must use `Query::all()` instead.
- `TableSchemaAwareInterface` was removed.
- `Driver::quote()` was removed. Use prepared statements instead.
- `Query::orderBy()` was added to replace `Query::order()`.
- `Query::groupBy()` was added to replace `Query::group()`.
- `SqlDialectTrait` has been removed and all its functionality has been moved into the `Driver` class itself.
- `CaseExpression` has been removed and should be replaced with `QueryExpression::case()` or `CaseStatementExpression`
- `Connection::connect()` has been removed. Use `$connection->getDriver()->connect()` instead.
- `Connection::disconnect()` has been removed. Use `$connection->getDriver()->disconnect()` instead.
- `cake.database.queries` has been added as an alternative to the `queriesLog` scope
- The ability to enable/disable ResultSet buffering has been removed. Results are always buffered.

[31] https://book.cakephp.org/4/en/appendices/4-4-migration-guide.html#requesthandlercomponent

Datasource

- The `getAccessible()` method was added to `EntityInterface`. Non-ORM implementations need to implement this method now.
- The `aliasField()` method was added to `RepositoryInterface`. Non-ORM implementations need to implement this method now.

Event

- Event payloads must be an array. Other object such as `ArrayAccess` are no longer cast to array and will raise a `TypeError` now.
- It is recommended to adjust event handlers to be void methods and use `$event->setResult()` instead of returning the result

Error

- `ErrorHandler` and `ConsoleErrorHandler` have been removed. See the 4.4 migration[32] guide for how to upgrade
- `ExceptionRenderer` has been removed and should be replaced with `WebExceptionRenderer`
- `ErrorLoggerInterface::log()` has been removed and should be replaced with `ErrorLoggerInterface::logException()`
- `ErrorLoggerInterface::logMessage()` has been removed and should be replaced with `ErrorLoggerInterface::logError()`

Filesystem

- The Filesystem package was removed, and `Filesystem` class was moved to the Utility package.

Http

- `ServerRequest` is no longer compatible with `files` as arrays. This behavior has been disabled by default since 4.1.0. The `files` data will now always contain `UploadedFileInterfaces` objects.

I18n

- `FrozenDate` was renamed to *Date* and `FrozenTime` was renamed to *DateTime*.
- `Time` now extends `Cake\Chronos\ChronosTime` and is therefore immutable.
- `Date::parseDateTime()` was removed.
- `Date::parseTime()` was removed.
- `Date::setToStringFormat()` and `Date::setJsonEncodeFormat()` no longer accept an array.
- `Date::i18nFormat()` and `Date::nice()` no longer accept a timezone parameter.
- Translation files for plugins with vendor prefixed names (`FooBar/Awesome`) will now have that prefix in the file name, e.g. `foo_bar_awesome.po` to avoid collision with a `awesome.po` file from a corresponding plugin (`Awesome`).

[32] https://book.cakephp.org/4/en/appendices/4-4-migration-guide.html#errorhandler-consoleerrorhandler

Log

- Log engine config now uses `null` instead of `false` to disable scopes. So instead of `'scopes' => false` you need to use `'scopes' => null` in your log config.

Mailer

- `Email` has been removed. Use Mailer[33] instead.
- `cake.mailer` has been added as an alternative to the `email` scope

ORM

- `EntityTrait::has()` now returns `true` when an attribute exists and is set to `null`. In previous versions of CakePHP this would return `false`. See the release notes for 4.5.0 for how to adopt this behavior in 4.x.
- `EntityTrait::extractOriginal()` now returns only existing fields, similar to `extractOriginalChanged()`.
- Finder arguments are now required to be associative arrays as they were always expected to be.
- `TranslateBehavior` now defaults to the `ShadowTable` strategy. If you are using the `Eav` strategy you will need to update your behavior configuration to retain the previous behavior.
- `allowMultipleNulls` option for `isUnique` rule now default to true matching the original 3.x behavior.
- `Table::query()` has been removed in favor of query-type specific functions.
- `Table::updateQuery()`, `Table::selectQuery()`, `Table::insertQuery()`, and `Table::deleteQuery()`) were added and return the new type-specific query objects below.
- `SelectQuery`, `InsertQuery`, `UpdateQuery` and `DeleteQuery` were added which represent only a single type of query and do not allow switching between query types nor calling functions unrelated to the specific query type.
- `Table::_initializeSchema()` has been removed and should be replaced by calling `$this->getSchema()` inside the `initialize()` method.
- `SaveOptionsBuilder` has been removed. Use a normal array for options instead.

Routing

- Static methods `connect()`, `prefix()`, `scope()` and `plugin()` of the `Router` have been removed and should be replaced by calling their non-static method variants via the `RouteBuilder` instance.
- `RedirectException` has been removed. Use `\Cake\Http\Exception\RedirectException` instead.

[33] https://book.cakephp.org/5/en/core-libraries/email.html

TestSuite

- `TestSuite` was removed. Users should use environment variables to customize unit test settings instead.
- `TestListenerTrait` was removed. PHPUnit dropped support for these listeners. See *PHPUnit 10 Upgrade*
- `IntegrationTestTrait::configRequest()` now merges config when called multiple times instead of replacing the currently present config.

Validation

- `Validation::isEmpty()` is no longer compatible with file upload shaped arrays. Support for PHP file upload arrays has been removed from `ServerRequest` as well so you should not see this as a problem outside of tests.
- Previously, most data validation error messages were simply `The provided value is invalid`. Now, the data validation error messages are worded more precisely. For example, `The provided value must be greater than or equal to \`5\``.

View

- `ViewBuilder` options are now truly associative (string keys).
- `NumberHelper` and `TextHelper` no longer accept an `engine` config.
- `ViewBuilder::setHelpers()` parameter `$merge` was removed. Use `ViewBuilder::addHelpers()` instead.
- Inside `View::initialize()`, prefer using `addHelper()` instead of `loadHelper()`. All configured helpers will be loaded afterwards, anyway.
- `View\Widget\FileWidget` is no longer compatible with PHP file upload shaped arrays. This is aligned with `ServerRequest` and `Validation` changes.
- `FormHelper` no longer sets `autocomplete=off` on CSRF token fields. This was a workaround for a Safari bug that is no longer relevant.

Deprecations

The following is a list of deprecated methods, properties and behaviors. These features will continue to function in 5.x and will be removed in 6.0.

Database

- `Query::order()` was deprecated. Use `Query::orderBy()` instead now that `Connection` methods are no longer proxied. This aligns the function name with the SQL statement.
- `Query::group()` was deprecated. Use `Query::groupBy()` instead now that `Connection` methods are no longer proxied. This aligns the function name with the SQL statement.

ORM

- Calling `Table::find()` with options array is deprecated. Use named arguments[34] instead. For e.g. instead of `find('all', ['conditions' => $array])` use `find('all', conditions: $array)`. Similarly for custom finder options, instead of `find('list', ['valueField' => 'name'])` use `find('list', valueField: 'name')` or multiple named arguments like `find(type: 'list', valueField: 'name', conditions: $array)`.

New Features

Improved type checking

CakePHP 5 leverages the expanded type system feature available in PHP 8.1+. CakePHP also uses `assert()` to provide improved error messages and additional type soundness. In production mode, you can configure PHP to not generate code for `assert()` yielding improved application performance. See the *Improve Your Application's Performance* for how to do this.

Collection

- Added `unique()` which filters out duplicate value specified by provided callback.
- `reject()` now supports a default callback which filters out truthy values which is the inverse of the default behavior of `filter()`

Core

- The `services()` method was added to `PluginInterface`.
- `PluginCollection::addFromConfig()` has been added to *simplify plugin loading*.

Database

- `ConnectionManager` now supports read and write connection roles. Roles can be configured with `read` and `write` keys in the connection config that override the shared config.
- `Query::all()` was added which runs result decorator callbacks and returns a result set for select queries.
- `Query::comment()` was added to add a SQL comment to the executed query. This makes it easier to debug queries.
- `EnumType` was added to allow mapping between PHP backed enums and a string or integer column.
- `getMaxAliasLength()` and `getConnectionRetries()` were added to `DriverInterface`.
- Supported drivers now automatically add auto-increment only to integer primary keys named "id" instead of all integer primary keys. Setting 'autoIncrement' to false always disables on all supported drivers.

[34] https://www.php.net/manual/en/functions.arguments.php#functions.named-arguments

Http

- Added support for PSR-17[35] factories interface. This allows `cakephp/http` to provide a client implementation to libraries that allow automatic interface resolution like php-http.
- Added `CookieCollection::__get()` and `CookieCollection::__isset()` to add ergonomic ways to access cookies without exceptions.

ORM

Required Entity Fields

Entities have a new opt-in functionality that allows making entities handle properties more strictly. The new behavior is called 'required fields'. When enabled, accessing properties that are not defined in the entity will raise exceptions. This impacts the following usage:

```
$entity->get();
$entity->has();
$entity->getOriginal();
isset($entity->attribute);
$entity->attribute;
```

Fields are considered defined if they pass `array_key_exists`. This includes null values. Because this can be a tedious to enable feature, it was deferred to 5.0. We'd like any feedback you have on this feature as we're considering making this the default behavior in the future.

Typed Finder Parameters

Table finders can now have typed arguments as required instead of an options array. For e.g. a finder for fetching posts by category or user:

```
public function findByCategoryOrUser(SelectQuery $query, array $options)
{
    if (isset($options['categoryId'])) {
        $query->where(['category_id' => $options['categoryId']]);
    }
    if (isset($options['userId'])) {
        $query->where(['user_id' => $options['userId']]);
    }

    return $query;
}
```

can now be written as:

```
public function findByCategoryOrUser(SelectQuery $query, ?int $categoryId = null, ?int
↪$userId = null)
{
    if ($categoryId) {
        $query->where(['category_id' => $categoryId]);
    }
```

(continues on next page)

[35] https://www.php-fig.org/psr/psr-17/

(continued from previous page)

```
    if ($userId) {
        $query->where(['user_id' => $userId]);
    }

    return $query;
}
```

The finder can then be called as `find('byCategoryOrUser', userId: $somevar)`. You can even include the special named arguments for setting query clauses. `find('byCategoryOrUser', userId: $somevar, conditions: ['enabled' => true])`.

A similar change has been applied to the `RepositoryInterface::get()` method:

```
public function view(int $id)
{
    $author = $this->Authors->get($id, [
        'contain' => ['Books'],
        'finder' => 'latest',
    ]);
}
```

can now be written as:

```
public function view(int $id)
{
    $author = $this->Authors->get($id, contain: ['Books'], finder: 'latest');
}
```

TestSuite

- `IntegrationTestTrait::requestAsJson()` has been added to set JSON headers for the next request.

Plugin Installer

- The plugin installer has been updated to automatically handle class autoloading for your app plugins. So you can remove the namespace to path mappings for your plugins from your `composer.json` and just run `composer dumpautoload`.

PHPUnit 10 Upgrade

With CakePHP 5 the minimum PHPUnit version has changed from `^8.5 || ^9.3` to `^10.1`. This introduces a few breaking changes from PHPUnit as well as from CakePHP's side.

phpunit.xml adjustments

It is recommended to let PHPUnit update its configuration file via the following command:

```
vendor/bin/phpunit --migrate-configuration
```

Note: Make sure you are already on PHPUnit 10 via `vendor/bin/phpunit --version` before executing this command!

With this command out of the way your `phpunit.xml` already has most of the recommended changes present.

New event system

PHPUnit 10 removed the old hook system and introduced a new Event system[36] which requires the following code in your `phpunit.xml` to be adjusted from:

```
<extensions>
  <extension class="Cake\TestSuite\Fixture\PHPUnitExtension"/>
</extensions>
```

to:

```
<extensions>
  <bootstrap class="Cake\TestSuite\Fixture\Extension\PHPUnitExtension"/>
</extensions>
```

`->withConsecutive()` has been removed

You can convert the removed `->withConsecutive()` method to a working interim solution like you can see here:

```
->withConsecutive(['firstCallArg'], ['secondCallArg'])
```

should be converted to:

```
->with(
    ...self::withConsecutive(['firstCallArg'], ['secondCallArg'])
)
```

the static `self::withConsecutive()` method has been added via the `Cake\TestSuite\PHPUnitConsecutiveTrait` to the base `Cake\TestSuite\TestCase` class so you don't have to manually add that trait to your Testcase classes.

[36] https://docs.phpunit.de/en/10.5/extending-phpunit.html#extending-the-test-runner

data providers have to be static

If your testcases leverage the data provider feature of PHPUnit then you have to adjust your data providers to be static:

```
public function myProvider(): array
```

should be converted to:

```
public static function myProvider(): array
```

CHAPTER 4

Tutorials & Examples

In this section, you can walk through typical CakePHP applications to see how all of the pieces come together.

Alternatively, you can refer to the non-official CakePHP plugin repository CakePackages[37] and the Bakery[38] for existing applications and components.

Content Management Tutorial

This tutorial will walk you through the creation of a simple CMS application. To start with, we'll be installing CakePHP, creating our database, and building simple article management.

Here's what you'll need:

1. A database server. We're going to be using MySQL server in this tutorial. You'll need to know enough about SQL in order to create a database, and run SQL snippets from the tutorial. CakePHP will handle building all the queries your application needs. Since we're using MySQL, also make sure that you have `pdo_mysql` enabled in PHP.
2. Basic PHP knowledge.

Before starting you should make sure that you're using a supported PHP version:

```
php -v
```

You should at least have got installed PHP 8.1 (CLI) or higher. Your webserver's PHP version must also be of 8.1 or higher, and should be the same version your command line interface (CLI) PHP is.

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

Getting CakePHP

The easiest way to install CakePHP is to use Composer. Composer is a simple way of installing CakePHP from your terminal or command line prompt. First, you'll need to download and install Composer if you haven't done so already. If you have cURL installed, run the following:

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

Or, you can download `composer.phar` from the Composer website[39].

Then simply type the following line in your terminal from your installation directory to install the CakePHP application skeleton in the **cms** directory of the current working directory:

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

If you downloaded and ran the Composer Windows Installer[40], then type the following line in your terminal from your installation directory (ie. C:\wamp\www\dev):

```
composer self-update && composer create-project --prefer-dist cakephp/app:5.* cms
```

The advantage to using Composer is that it will automatically complete some important set up tasks, such as setting the correct file permissions and creating your **config/app.php** file for you.

There are other ways to install CakePHP. If you cannot or don't want to use Composer, check out the *Installation* section.

Regardless of how you downloaded and installed CakePHP, once your set up is completed, your directory setup should look like the following, though other files may also be present:

```
cms/
  bin/
  config/
  plugins/
  resources/
  src/
  templates/
  tests/
  tmp/
  vendor/
  webroot/
  composer.json
  index.php
  README.md
```

Now might be a good time to learn a bit about how CakePHP's directory structure works: check out the *CakePHP Folder Structure* section.

If you get lost during this tutorial, you can see the finished result on GitHub[41].

Tip: The `bin/cake` console utility can build most of the classes and data tables in this tutorial automatically. However, we recommend following along with the manual code examples to understand how the pieces fit together and how to add your application logic.

---

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

Checking our Installation

We can quickly check that our installation is correct, by checking the default home page. Before you can do that, you'll need to start the development server:

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

bin/cake server
```

Note: For Windows, the command needs to be `bin\cake server` (note the backslash).

This will start PHP's built-in webserver on port 8765. Open up **http://localhost:8765** in your web browser to see the welcome page. All the bullet points should be green chef hats other than CakePHP being able to connect to your database. If not, you may need to install additional PHP extensions, or set directory permissions.

Next, we will build our *Database*.

CMS Tutorial - Creating the Database

Now that we have CakePHP installed, let's set up the database for our CMS application. If you haven't already done so, create an empty database for use in this tutorial, with the name of your choice such as `cake_cms`. If you are using MySQL/MariaDB, you can execute the following SQL to create the necessary tables:

```
CREATE DATABASE cake_cms;

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),
```

(continues on next page)

(continued from previous page)

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

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

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

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

If you are using PostgreSQL, connect to the `cake_cms` database and execute the following SQL instead:

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

You may have noticed that the `articles_tags` table uses a composite primary key. CakePHP supports composite primary keys almost everywhere, allowing you to have simpler schemas that don't require additional `id` columns.

The table and column names we used were not arbitrary. By using CakePHP's *naming conventions*, we can leverage CakePHP more effectively and avoid needing to configure the framework. While CakePHP is flexible enough to accommodate almost any database schema, adhering to the conventions will save you time as you can leverage the convention-based defaults CakePHP provides.

Database Configuration

Next, let's tell CakePHP where our database is and how to connect to it. Replace the values in the `Datasources.default` array in your **config/app_local.php** file with those that apply to your setup. A sample completed configuration array might look something like the following:

```
<?php
// config/app_local.php
return [
    // More configuration above.
    'Datasources' => [
        'default' => [
            'host' => 'localhost',
            'username' => 'cakephp',
            'password' => 'AngelF00dC4k3~',
            'database' => 'cake_cms',
            'url' => env('DATABASE_URL', null),
        ],
    ],
    // More configuration below.
];
```

Once you've saved your **config/app_local.php** file, you should see that the 'CakePHP is able to connect to the database' section has a green chef hat.

Note: The file **config/app_local.php** is a local override of the file **config/app.php** used to configure your development environment quickly.

Migrations

The SQL statements to create the tables for this tutorial can also be generated using the Migrations Plugin. Migrations provide a platform-independent way to run queries so the subtle differences between MySQL, PostgreSQL, SQLite, etc. don't become obstacles.

```
bin/cake bake migration CreateUsers email:string password:string created modified
bin/cake bake migration CreateArticles user_id:integer title:string␣
↪slug:string[191]:unique body:text published:boolean created modified
bin/cake bake migration CreateTags title:string[191]:unique created modified
bin/cake bake migration CreateArticlesTags article_id:integer:primary tag_
↪id:integer:primary created modified
```

Note: Some adjustments to the generated code might be necessary. For example, the composite primary key on `articles_tags` will be set to auto-increment both columns:

```
$table->addColumn('article_id', 'integer', [
    'autoIncrement' => true,
    'default' => null,
    'limit' => 11,
    'null' => false,
]);
$table->addColumn('tag_id', 'integer', [
    'autoIncrement' => true,
    'default' => null,
    'limit' => 11,
    'null' => false,
]);
```

Remove those lines to prevent foreign key problems. Once adjustments are done:

```
bin/cake migrations migrate
```

Likewise, the starter data records can be done with seeds.

```
bin/cake bake seed Users
bin/cake bake seed Articles
```

Fill the seed data above into the new `UsersSeed` and `ArticlesSeed` classes, then:

```
bin/cake migrations seed
```

Read more about building migrations and data seeding: Migrations[42]

With the database built, we can now build *Models*.

[42] https://book.cakephp.org/migrations/4/

CMS Tutorial - Creating our First Model

Models are the heart of CakePHP applications. They enable us to read and modify our data. They allow us to build relations between our data, validate data, and apply application rules. Models provide the foundation necessary to create our controller actions and templates.

CakePHP's models are composed of `Table` and `Entity` objects. `Table` objects provide access to the collection of entities stored in a specific table. They are stored in **src/Model/Table**. The file we'll be creating will be saved to **src/Model/Table/ArticlesTable.php**. The completed file should look like this:

```
<?php
// src/Model/Table/ArticlesTable.php
declare(strict_types=1);

namespace App\Model\Table;

use Cake\ORM\Table;

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

We've attached the *Timestamp* behavior, which will automatically populate the `created` and `modified` columns of our table. By naming our Table object `ArticlesTable`, CakePHP can use naming conventions to know that our model uses the `articles` table. CakePHP also uses conventions to know that the `id` column is our table's primary key.

Note: CakePHP will dynamically create a model object for you if it cannot find a corresponding file in **src/Model/Table**. This also means that if you accidentally name your file wrong (i.e. articlestable.php or ArticleTable.php), CakePHP will not recognize any of your settings and will use the generated model instead.

We'll also create an Entity class for our Articles. Entities represent a single record in the database and provide row-level behavior for our data. Our entity will be saved to **src/Model/Entity/Article.php**. The completed file should look like this:

```
<?php
// src/Model/Entity/Article.php
declare(strict_types=1);

namespace App\Model\Entity;

use Cake\ORM\Entity;

class Article extends Entity
{
    protected array $_accessible = [
        'title' => true,
        'body' => true,
```

(continues on next page)

(continued from previous page)

```
        'published' => true,
        'created' => true,
        'modified' => true,
        'users' => true,
    ];
}
```

Right now, our entity is quite slim; we've only set up the `_accessible` property, which controls how properties can be modified by *Mass Assignment*.

Tip: The `ArticlesTable` and `Article` Entity classes can be generated from a terminal:

```
bin/cake bake model articles
```

We can't do much with this model yet. Next, we'll create our first *Controller and Template* to allow us to interact with our model.

CMS Tutorial - Creating the Articles Controller

With our model created, we need a controller for our articles. Controllers in CakePHP handle HTTP requests and execute business logic contained in model methods, to prepare the response. We'll place this new controller in a file called **ArticlesController.php** inside the **src/Controller** directory. Here's what the basic controller should look like:

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

namespace App\Controller;

class ArticlesController extends AppController
{
}
```

Now, let's add an action to our controller. Actions are controller methods that have routes connected to them. For example, when a user requests **www.example.com/articles/index** (which is also the same as **www.example.com/articles**), CakePHP will call the `index` method of your `ArticlesController`. This method should query the model layer, and prepare a response by rendering a Template in the View. The code for that action would look like this:

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

namespace App\Controller;

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

By defining function `index()` in our `ArticlesController`, users can now access the logic there by requesting **www.example.com/articles/index**. Similarly, if we were to define a function called `foobar()`, users would be able to access that at **www.example.com/articles/foobar**. You may be tempted to name your controllers and actions in a way that allows you to obtain specific URLs. Resist that temptation. Instead, follow the *CakePHP Conventions* creating readable, meaningful action names. You can then use *Routing* to connect the URLs you want to the actions you've created.

Our controller action is very simple. It fetches a paginated set of articles from the database, using the Articles Model that is automatically loaded via naming conventions. It then uses `set()` to pass the articles into the Template (which we'll create soon). CakePHP will automatically render the template after our controller action completes.

Create the Article List Template

Now that we have our controller pulling data from the model, and preparing our view context, let's create a view template for our index action.

CakePHP view templates are presentation-flavored PHP code that is inserted inside the application's layout. While we'll be creating HTML here, Views can also generate JSON, CSV or even binary files like PDFs.

A layout is presentation code that is wrapped around a view. Layout files contain common site elements like headers, footers and navigation elements. Your application can have multiple layouts, and you can switch between them, but for now, let's just use the default layout.

CakePHP's template files are stored in **templates** inside a folder named after the controller they correspond to. So we'll have to create a folder named 'Articles' in this case. Add the following code to your application:

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

<h1>Articles</h1>
<table>
    <tr>
        <th>Title</th>
        <th>Created</th>
    </tr>

    <!-- Here is where we iterate through our $articles query object, printing out␣
↪article info -->

    <?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>
```

In the last section we assigned the 'articles' variable to the view using `set()`. Variables passed into the view are available in the view templates as local variables which we used in the above code.

You might have noticed the use of an object called `$this->Html`. This is an instance of the CakePHP *HtmlHelper*. CakePHP comes with a set of view helpers that make tasks like creating links, forms, and pagination buttons. You can

learn more about *Helpers* in their chapter, but what's important to note here is that the `link()` method will generate an HTML link with the given link text (the first parameter) and URL (the second parameter).

When specifying URLs in CakePHP, it is recommended that you use arrays or *named routes*. These syntaxes allow you to leverage the reverse routing features CakePHP offers.

At this point, you should be able to point your browser to **http://localhost:8765/articles/index**. You should see your list view, correctly formatted with the title and table listing of the articles.

Create the View Action

If you were to click one of the 'view' links in our Articles list page, you'd see an error page saying that action hasn't been implemented. Lets fix that now:

```
// Add to existing src/Controller/ArticlesController.php file

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

While this is a simple action, we've used some powerful CakePHP features. We start our action off by using `findBySlug()` which is a *Dynamic Finder*. This method allows us to create a basic query that finds articles by a given slug. We then use `firstOrFail()` to either fetch the first record, or throw a `NotFoundException`.

Our action takes a `$slug` parameter, but where does that parameter come from? If a user requests `/articles/view/first-post`, then the value 'first-post' is passed as `$slug` by CakePHP's routing and dispatching layers. If we reload our browser with our new action saved, we'd see another CakePHP error page telling us we're missing a view template; let's fix that.

Create the View Template

Let's create the view for our new 'view' action and place it in **templates/Articles/view.php**

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

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

You can verify that this is working by trying the links at `/articles/index` or manually requesting an article by accessing URLs like `/articles/view/first-post`.

Adding Articles

With the basic read views created, we need to make it possible for new articles to be created. Start by creating an `add()` action in the `ArticlesController`. Our controller should now look like:

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

use App\Controller\AppController;

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

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

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

            // 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(__('Your article has been saved.'));
                return $this->redirect(['action' => 'index']);
            }
            $this->Flash->error(__('Unable to add your article.'));
        }
        $this->set('article', $article);
    }
}
```

Note: You need to include the *Flash* component in any controller where you will use it. Often it makes sense to include it in your `AppController`, which is there already for this tutorial.

Here's what the `add()` action does:

- If the HTTP method of the request was POST, try to save the data using the Articles model.
- If for some reason it doesn't save, just render the view. This gives us a chance to show the user validation errors or other warnings.

Every CakePHP request includes a request object which is accessible using `$this->request`. The request object contains information regarding the request that was just received. We use the *Cake\Http\ServerRequest::is()* method to check that the request is a HTTP POST request.

Our POST data is available in `$this->request->getData()`. You can use the `pr()` or `debug()` functions to print it out if you want to see what it looks like. To save our data, we first 'marshal' the POST data into an Article Entity. The Entity is then persisted using the ArticlesTable we created earlier.

After saving our new article we use FlashComponent's `success()` method to set a message into the session. The `success` method is provided using PHP's magic method features[43]. Flash messages will be displayed on the next page after redirecting. In our layout we have `<?= $this->Flash->render() ?>` which displays flash messages and clears the corresponding session variable. Finally, after saving is complete, we use *Cake\Controller\Controller::redirect* to send the user back to the articles list. The param `['action' => 'index']` translates to URL `/articles` i.e the index action of the `ArticlesController`. You can refer to `Cake\Routing\Router::url()` function on the API[44] to see the formats in which you can specify a URL for various CakePHP functions.

Create Add Template

Here's our add view template:

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

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

We use the FormHelper to generate the opening tag for an HTML form. Here's the HTML that `$this->Form->create()` generates:

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

Because we called `create()` without a URL option, `FormHelper` assumes we want the form to submit back to the current action.

The `$this->Form->control()` method is used to create form elements of the same name. The first parameter tells CakePHP which field they correspond to, and the second parameter allows you to specify a wide array of options - in this case, the number of rows for the textarea. There's a bit of introspection and conventions used here. The `control()` will output different form elements based on the model field specified, and use inflection to generate the label text. You can customize the label, the input or any other aspect of the form controls using options. The `$this->Form->end()` call closes the form.

Now let's go back and update our **templates/Articles/index.php** view to include a new "Add Article" link. Before the `<table>`, add the following line:

```
<?= $this->Html->link('Add Article', ['action' => 'add']) ?>
```

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

Adding Simple Slug Generation

If we were to save an Article right now, saving would fail as we are not creating a slug attribute, and the column is `NOT NULL`. Slug values are typically a URL-safe version of an article's title. We can use the *beforeSave() callback* of the ORM to populate our slug:

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

use Cake\ORM\Table;
// the Text class
use Cake\Utility\Text;
// the EventInterface class
use Cake\Event\EventInterface;

// Add the following method.

public function beforeSave(EventInterface $event, $entity, $options)
{
    if ($entity->isNew() && !$entity->slug) {
        $sluggedTitle = Text::slug($entity->title);
        // trim slug to maximum length defined in schema
        $entity->slug = substr($sluggedTitle, 0, 191);
    }
}
```

This code is simple, and doesn't take into account duplicate slugs. But we'll fix that later on.

Add Edit Action

Our application can now save articles, but we can't edit them. Lets rectify that now. Add the following action to your `ArticlesController`:

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

// Add the following method.

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(__('Your article has been updated.'));
            return $this->redirect(['action' => 'index']);
        }
        $this->Flash->error(__('Unable to update your article.'));
    }
```

(continues on next page)

(continued from previous page)

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

This action first ensures that the user has tried to access an existing record. If they haven't passed in an `$slug` parameter, or the article does not exist, a `NotFoundException` will be thrown, and the CakePHP ErrorHandler will render the appropriate error page.

Next the action checks whether the request is either a POST or a PUT request. If it is, then we use the POST/PUT data to update our article entity by using the `patchEntity()` method. Finally, we call `save()`, set the appropriate flash message, and either redirect or display validation errors.

Create Edit Template

The edit template should look like this:

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

<h1>Edit 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(__('Save Article'));
    echo $this->Form->end();
?>
```

This template outputs the edit form (with the values populated), along with any necessary validation error messages.

You can now update your index view with links to edit specific articles:

```
<!-- File: templates/Articles/index.php  (edit links added) -->

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

<!-- Here's where we iterate through our $articles query object, printing out article␣
↪info -->

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

(continues on next page)

(continued from previous page)

```
            <?= $article->created->format(DATE_RFC850) ?>
        </td>
        <td>
            <?= $this->Html->link('Edit', ['action' => 'edit', $article->slug]) ?>
        </td>
    </tr>
<?php endforeach; ?>

</table>
```

Update Validation Rules for Articles

Up until this point our Articles had no input validation done. Lets fix that by using *a validator*:

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

// add this use statement right below the namespace declaration to import
// the Validator class
use Cake\Validation\Validator;

// Add the following method.
public function validationDefault(Validator $validator): Validator
{
    $validator
        ->notEmptyString('title')
        ->minLength('title', 10)
        ->maxLength('title', 255)

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

    return $validator;
}
```

The `validationDefault()` method tells CakePHP how to validate your data when the `save()` method is called. Here, we've specified that both the title, and body fields must not be empty, and have certain length constraints.

CakePHP's validation engine is powerful and flexible. It provides a suite of frequently used rules for tasks like email addresses, IP addresses etc. and the flexibility for adding your own validation rules. For more information on that setup, check the *Validation* documentation.

Now that your validation rules are in place, use the app to try to add an article with an empty title or body to see how it works. Since we've used the *`Cake\View\Helper\FormHelper::control()`* method of the FormHelper to create our form elements, our validation error messages will be shown automatically.

Add Delete Action

Next, let's make a way for users to delete articles. Start with a `delete()` action in the `ArticlesController`:

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

// Add the following method.

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

    $article = $this->Articles->findBySlug($slug)->firstOrFail();
    if ($this->Articles->delete($article)) {
        $this->Flash->success(__('The {0} article has been deleted.', $article->title));
        return $this->redirect(['action' => 'index']);
    }
}
```

This logic deletes the article specified by `$slug`, and uses `$this->Flash->success()` to show the user a confirmation message after redirecting them to `/articles`. If the user attempts to delete an article using a GET request, `allowMethod()` will throw an exception. Uncaught exceptions are captured by CakePHP's exception handler, and a nice error page is displayed. There are many built-in *Exceptions* that can be used to indicate the various HTTP errors your application might need to generate.

> **Warning:** Allowing content to be deleted using GET requests is *very* dangerous, as web crawlers could accidentally delete all your content. That is why we used `allowMethod()` in our controller.

Because we're only executing logic and redirecting to another action, this action has no template. You might want to update your index template with links that allow users to delete articles:

```
<!-- File: templates/Articles/index.php  (delete links added) -->

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

<!-- Here's where we iterate through our $articles query object, printing out article␣
↪info -->

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

(continues on next page)

(continued from previous page)

```
            <?= $article->created->format(DATE_RFC850) ?>
        </td>
        <td>
            <?= $this->Html->link('Edit', ['action' => 'edit', $article->slug]) ?>
            <?= $this->Form->postLink(
                'Delete',
                ['action' => 'delete', $article->slug],
                ['confirm' => 'Are you sure?'])
            ?>
        </td>
    </tr>
<?php endforeach; ?>

</table>
```

Using *postLink()* will create a link that uses JavaScript to do a POST request deleting our article.

Note: This view code also uses the `FormHelper` to prompt the user with a JavaScript confirmation dialog before they attempt to delete an article.

Tip: The `ArticlesController` can also be built with `bake`:

```
/bin/cake bake controller articles
```

However, this does not build the **templates/Articles/*.php** files.

With a basic articles management setup, we'll create the *basic actions for our Tags and Users tables*.

CMS Tutorial - Tags and Users

With the basic article creation functionality built, we need to enable multiple authors to work in our CMS. Previously, we built all the models, views and controllers by hand. This time around we're going to use *Bake Console* to create our skeleton code. Bake is a powerful code generation CLI (Command Line Interface) tool that leverages the conventions CakePHP uses to create skeleton CRUD (Create, Read, Update, Delete) applications very efficiently. We're going to use `bake` to build our users code:

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

# You can overwrite any existing files.
bin/cake bake model users
bin/cake bake controller users
bin/cake bake template users
```

These 3 commands will generate:

- The Table, Entity, Fixture files.
- The Controller
- The CRUD templates.

- Test cases for each generated class.

Bake will also use the CakePHP conventions to infer the associations, and validation your models have.

Adding Tagging to Articles

With multiple users able to access our small CMS it would be nice to have a way to categorize our content. We'll use tags and tagging to allow users to create free-form categories and labels for their content. Again, we'll use `bake` to quickly generate some skeleton code for our application:

```
# Generate all the code at once.
bin/cake bake all tags
```

Once you have the scaffold code created, create a few sample tags by going to **http://localhost:8765/tags/add**.

Now that we have a Tags table, we can create an association between Articles and Tags. We can do so by adding the following to the `initialize` method on the `ArticlesTable`:

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

This association will work with this simple definition because we followed CakePHP conventions when creating our tables. For more information, read *Associations - Linking Tables Together*.

Updating Articles to Enable Tagging

Now that our application has tags, we need to enable users to tag their articles. First, update the `add` action to look like:

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

use App\Controller\AppController;

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

            // 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(__('Your article has been saved.'));
                return $this->redirect(['action' => 'index']);
            }
```

(continues on next page)

(continued from previous page)

```
            $this->Flash->error(__('Unable to add your article.'));
        }
        // Get a list of tags.
        $tags = $this->Articles->Tags->find('list')->all();

        // Set tags to the view context
        $this->set('tags', $tags);

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

    // Other actions
}
```

The added lines load a list of tags as an associative array of `id => title`. This format will let us create a new tag input in our template. Add the following to the PHP block of controls in **templates/Articles/add.php**:

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

This will render a multiple select element that uses the `$tags` variable to generate the select box options. You should now create a couple new articles that have tags, as in the following section we'll be adding the ability to find articles by tags.

You should also update the `edit` method to allow adding or editing tags. The edit method should now look like:

```
public function edit($slug)
{
    $article = $this->Articles
        ->findBySlug($slug)
        ->contain('Tags') // load associated Tags
        ->firstOrFail();
    if ($this->request->is(['post', 'put'])) {
        $this->Articles->patchEntity($article, $this->request->getData());
        if ($this->Articles->save($article)) {
            $this->Flash->success(__('Your article has been updated.'));
            return $this->redirect(['action' => 'index']);
        }
        $this->Flash->error(__('Unable to update your article.'));
    }

    // Get a list of tags.
    $tags = $this->Articles->Tags->find('list')->all();

    // Set tags to the view context
    $this->set('tags', $tags);

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

Remember to add the new tags multiple select control we added to the **add.php** template to the **templates/Articles/edit.php** template as well.

Finding Articles By Tags

Once users have categorized their content, they will want to find that content by the tags they used. For this feature we'll implement a route, controller action, and finder method to search through articles by tag.

Ideally, we'd have a URL that looks like **http://localhost:8765/articles/tagged/funny/cat/gifs**. This would let us find all the articles that have the 'funny', 'cat' or 'gifs' tags. Before we can implement this, we'll add a new route. Your **config/routes.php** (with the baked comments removed) should look like:

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

$routes->setRouteClass(DashedRoute::class);

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

    // Add this
    // New route we're adding for our tagged action.
    // The trailing `*` tells CakePHP that this action has
    // passed parameters.
    $builder->scope('/articles', function (RouteBuilder $builder) {
        $builder->connect('/tagged/*', ['controller' => 'Articles', 'action' => 'tags']);
    });

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

The above defines a new 'route' which connects the **/articles/tagged/** path, to `ArticlesController::tags()`. By defining routes, you can isolate how your URLs look, from how they are implemented. If we were to visit **http://localhost:8765/articles/tagged**, we would see a helpful error page from CakePHP informing you that the controller action does not exist. Let's implement that missing method now. In **src/Controller/ArticlesController.php** add the following:

```
public function tags()
{
    // The 'pass' key is provided by CakePHP and contains all
    // the passed URL path segments in the request.
    $tags = $this->request->getParam('pass');

    // Use the ArticlesTable to find tagged articles.
    $articles = $this->Articles->find('tagged', tags: $tags)
        ->all();

    // Pass variables into the view template context.
    $this->set([
        'articles' => $articles,
        'tags' => $tags
    ]);
}
```

To access other parts of the request data, consult the *Request* section.

Since passed arguments are passed as method parameters, you could also write the action using PHP's variadic argument:

```
public function tags(...$tags)
{
    // Use the ArticlesTable to find tagged articles.
    $articles = $this->Articles->find('tagged', tags: $tags)
        ->all();

    // Pass variables into the view template context.
    $this->set([
        'articles' => $articles,
        'tags' => $tags
    ]);
}
```

Creating the Finder Method

In CakePHP we like to keep our controller actions slim, and put most of our application's logic in the model layer. If you were to visit the **/articles/tagged** URL now you would see an error that the `findTagged()` method has not been implemented yet, so let's do that. In **src/Model/Table/ArticlesTable.php** add the following:

```
// add this use statement right below the namespace declaration to import
// the Query class
use Cake\ORM\Query\SelectQuery;

// The $query argument is a query builder instance.
// The $options array will contain the 'tags' option we passed
// to find('tagged') in our controller action.
public function findTagged(SelectQuery $query, array $tags = []): SelectQuery
{
    $columns = [
        'Articles.id', 'Articles.user_id', 'Articles.title',
        'Articles.body', 'Articles.published', 'Articles.created',
        'Articles.slug',
    ];

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

    if (empty($tags)) {
        // If there are no tags provided, find articles that have no tags.
        $query->leftJoinWith('Tags')
            ->where(['Tags.title IS' => null]);
    } else {
        // Find articles that have one or more of the provided tags.
        $query->innerJoinWith('Tags')
            ->where(['Tags.title IN' => $tags]);
    }

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

We just implemented a *custom finder method*. This is a very powerful concept in CakePHP that allows you to package up re-usable queries. Finder methods always get a *Query Builder* object and an array of options as parameters. Finders can manipulate the query and add any required conditions or criteria. When complete, finder methods must return a modified query object. In our finder we've leveraged the `distinct()` and `leftJoin()` methods which allow us to find distinct articles that have a 'matching' tag.

Creating the View

Now if you visit the **/articles/tagged** URL again, CakePHP will show a new error letting you know that you have not made a view file. Next, let's build the view file for our `tags()` action:

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

<section>
<?php foreach ($articles as $article): ?>
    <article>
        <!-- Use the HtmlHelper to create a link -->
        <h4><?= $this->Html->link(
            $article->title,
            ['controller' => 'Articles', 'action' => 'view', $article->slug]
        ) ?></h4>
        <span><?= h($article->created) ?></span>
    </article>
<?php endforeach; ?>
</section>
```

In the above code we use the *Html* and *Text* helpers to assist in generating our view output. We also use the `h` shortcut function to HTML encode output. You should remember to always use `h()` when outputting data to prevent HTML injection issues.

The **tags.php** file we just created follows the CakePHP conventions for view template files. The convention is to have the template use the lower case and underscored version of the controller action name.

You may notice that we were able to use the `$tags` and `$articles` variables in our view template. When we use the `set()` method in our controller, we set specific variables to be sent to the view. The View will make all passed variables available in the template scope as local variables.

You should now be able to visit the **/articles/tagged/funny** URL and see all the articles tagged with 'funny'.

Improving the Tagging Experience

Right now, adding new tags is a cumbersome process, as authors need to pre-create all the tags they want to use. We can improve the tag selection UI by using a comma separated text field. This will let us give a better experience to our users, and use some more great features in the ORM.

Adding a Computed Field

Because we'll want a simple way to access the formatted tags for an entity, we can add a virtual/computed field to the entity. In **src/Model/Entity/Article.php** add the following:

```
// add this use statement right below the namespace declaration to import
// the Collection class
use Cake\Collection\Collection;

// Update the accessible property to contain `tag_string`
protected array $_accessible = [
    //other fields...
    'tag_string' => true
];

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

This will let us access the `$article->tag_string` computed property. We'll use this property in controls later on.

Updating the Views

With the entity updated we can add a new control for our tags. In **templates/Articles/add.php** and **templates/Articles/edit.php**, replace the existing `tags._ids` control with the following:

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

We'll also need to update the article view template. In **templates/Articles/view.php** add the line as shown:

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

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

You should also update the view method to allow retrieving existing tags:

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

public function view($slug = null)
```

(continues on next page)

(continued from previous page)

```
{
   // Update retrieving tags with contain()
   $article = $this->Articles
        ->findBySlug($slug)
        ->contain('Tags')
        ->firstOrFail();
    $this->set(compact('article'));
}
```

Persisting the Tag String

Now that we can view existing tags as a string, we'll want to save that data as well. Because we marked the `tag_string` as accessible, the ORM will copy that data from the request into our entity. We can use a `beforeSave()` hook method to parse the tag string and find/build the related entities. Add the following to **src/Model/Table/ArticlesTable.php**:

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

    // Other code
}

protected function _buildTags($tagString)
{
    // Trim tags
    $newTags = array_map('trim', explode(',', $tagString));
    // Remove all empty tags
    $newTags = array_filter($newTags);
    // Reduce duplicated tags
    $newTags = array_unique($newTags);

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

    // Remove existing tags from the list of new tags.
    foreach ($tags->extract('title') as $existing) {
        $index = array_search($existing, $newTags);
        if ($index !== false) {
            unset($newTags[$index]);
        }
    }
    // Add existing tags.
    foreach ($tags as $tag) {
        $out[] = $tag;
    }
    // Add new tags.
    foreach ($newTags as $tag) {
```

(continues on next page)

(continued from previous page)

```
            $out[] = $this->Tags->newEntity(['title' => $tag]);
        }
        return $out;
    }
```

If you now create or edit articles, you should be able to save tags as a comma separated list of tags, and have the tags and linking records automatically created.

While this code is a bit more complicated than what we've done so far, it helps to showcase how powerful the ORM in CakePHP is. You can manipulate query results using the *Collections* methods, and handle scenarios where you are creating entities on the fly with ease.

Auto-populating the Tag String

Before we finish up, we'll need a mechanism that will load the associated tags (if any) whenever we load an article.

In your **src/Model/Table/ArticlesTable.php**, change:

```
public function initialize(array $config): void
{
    $this->addBehavior('Timestamp');
    // Change this line
    $this->belongsToMany('Tags', [
        'joinTable' => 'articles_tags',
        'dependent' => true
    ]);
}
```

This will tell the Articles table model that there is a join table associated with tags. The 'dependent' option tells the table to delete any associated records from the join table if an article is deleted.

Lastly, update the findBySlug() method calls in **src/Controller/ArticlesController.php**:

```
public function edit($slug)
{
    // Update this line
    $article = $this->Articles
        ->findBySlug($slug)
        ->contain('Tags')
        ->firstOrFail();
...
}

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

The `contain()` method tells the `ArticlesTable` object to also populate the Tags association when the article is loaded. Now when tag_string is called for an Article entity, there will be data present to create the string!

Next we'll be adding *authentication*.

CMS Tutorial - Authentication

Now that our CMS has users, we can enable them to login using the cakephp/authentication[45] plugin. We'll start off by ensuring passwords are stored securely in our database. Then we are going to provide a working login and logout, and enable new users to register.

Installing Authentication Plugin

Use composer to install the Authentication Plugin:

```
composer require "cakephp/authentication:^3.0"
```

Adding Password Hashing

You need to have created the `Controller`, `Table`, `Entity` and templates for the `users` table in your database. You can do this manually like you did before for the ArticlesController, or you can use the bake shell to generate the classes for you using:

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

If you create or update a user with this setup, you might notice that the passwords are stored in plain text. This is really bad from a security point of view, so lets fix that.

This is also a good time to talk about the model layer in CakePHP. In CakePHP, we use different classes to operate on collections of records and single records. Methods that operate on the collection of entities are put in the `Table` class, while features belonging to a single record are put on the `Entity` class.

For example, password hashing is done on the individual record, so we'll implement this behavior on the entity object. Because we want to hash the password each time it is set, we'll use a mutator/setter method. CakePHP will call a convention based setter method any time a property is set in one of your entities. Let's add a setter for the password. In **src/Model/Entity/User.php** add the following:

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

use Authentication\PasswordHasher\DefaultPasswordHasher; // Add this line
use Cake\ORM\Entity;

class User extends Entity
{
    // Code from bake.

    // Add this method
    protected function _setPassword(string $password) : ?string
    {
```

(continues on next page)

[45] https://book.cakephp.org/authentication/2

(continued from previous page)

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

Now, point your browser to **http://localhost:8765/users** to see a list of users. Remember you'll need to have your local server running. Start a standalone PHP server using `bin/cake server`.

You can edit the default user that was created during *Installation*. If you change that user's password, you should see a hashed password instead of the original value on the list or view pages. CakePHP hashes passwords with bcrypt[46] by default. We recommend bcrypt for all new applications to keep your security standards high. This is the recommended password hash algorithm for PHP[47].

Note: Create a hashed password for at least one of the user accounts now! It will be needed in the next steps. After updating the password, you'll see a long string stored in the password column. Note bcrypt will generate a different hash even for the same password saved twice.

Adding Login

Now it's time to configure the Authentication Plugin. The Plugin will handle the authentication process using 3 different classes:

- `Application` will use the Authentication Middleware and provide an AuthenticationService, holding all the configuration we want to define how are we going to check the credentials, and where to find them.
- `AuthenticationService` will be a utility class to allow you configure the authentication process.
- `AuthenticationMiddleware` will be executed as part of the middleware queue, this is before your Controllers are processed by the framework, and will pick the credentials and process them to check if the user is authenticated.

If you remember, we used *AuthComponent* before to handle all these steps. Now the logic is divided into specific classes and the authentication process happens before your controller layer. First it checks if the user is authenticated (based on the configuration you provided) and injects the user and the authentication results into the request for further reference.

In **src/Application.php**, add the following imports:

```
// In src/Application.php add the following imports
use Authentication\AuthenticationService;
use Authentication\AuthenticationServiceInterface;
use Authentication\AuthenticationServiceProviderInterface;
use Authentication\Middleware\AuthenticationMiddleware;
use Cake\Routing\Router;
use Psr\Http\Message\ServerRequestInterface;
```

Then implement the authentication interface on your `Application` class:

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

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

Then add the following:

```
// src/Application.php
public function middleware(MiddlewareQueue $middlewareQueue): MiddlewareQueue
{
    $middlewareQueue
        // ... other middleware added before
        ->add(new RoutingMiddleware($this))
        ->add(new BodyParserMiddleware())
        // Add the AuthenticationMiddleware. It should be after routing and body parser.
        ->add(new AuthenticationMiddleware($this));

    return $middlewareQueue;
}

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

    // Load identifiers, ensure we check email and password fields
    $authenticationService->loadIdentifier('Authentication.Password', [
        'fields' => [
            'username' => 'email',
            'password' => 'password',
        ],
    ]);

    // Load the authenticators, you want session first
    $authenticationService->loadAuthenticator('Authentication.Session');
    // Configure form data check to pick email and password
    $authenticationService->loadAuthenticator('Authentication.Form', [
        'fields' => [
            'username' => 'email',
            'password' => 'password',
        ],
        'loginUrl' => Router::url('/users/login'),
    ]);

    return $authenticationService;
}
```

In your `AppController` class add the following code:

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

(continues on next page)

(continued from previous page)

```
public function initialize(): void
{
    parent::initialize();
    $this->loadComponent('Flash');

    // Add this line to check authentication result and lock your site
    $this->loadComponent('Authentication.Authentication');
```

Now, on every request, the `AuthenticationMiddleware` will inspect the request session to look for an authenticated user. If we are loading the `/users/login` page, it will also inspect the posted form data (if any) to extract the credentials. By default the credentials will be extracted from the `username` and `password` fields in the request data. The authentication result will be injected in a request attribute named `authentication`. You can inspect the result at any time using `$this->request->getAttribute('authentication')` from your controller actions. All your pages will be restricted as the `AuthenticationComponent` is checking the result on every request. When it fails to find any authenticated user, it will redirect the user to the `/users/login` page. Note at this point, the site won't work as we don't have a login page yet. If you visit your site, you'll get an "infinite redirect loop" so let's fix that.

Note: If your application serves from both SSL and non-SSL protocols, then you might have problems with sessions being lost, in case your application is on non-SSL protocol. You need to enable access by setting session.cookie_secure to false in your config config/app.php or config/app_local.php. (See *CakePHP's defaults on session.cookie_secure*)

In your `UsersController`, add the following code:

```
public function beforeFilter(\Cake\Event\EventInterface $event)
{
    parent::beforeFilter($event);
    // Configure the login action to not require authentication, preventing
    // the infinite redirect loop issue
    $this->Authentication->addUnauthenticatedActions(['login']);
}

public function login()
{
    $this->request->allowMethod(['get', 'post']);
    $result = $this->Authentication->getResult();
    // regardless of POST or GET, redirect if user is logged in
    if ($result && $result->isValid()) {
        // redirect to /articles after login success
        $redirect = $this->request->getQuery('redirect', [
            'controller' => 'Articles',
            'action' => 'index',
        ]);

        return $this->redirect($redirect);
    }
    // display error if user submitted and authentication failed
    if ($this->request->is('post') && !$result->isValid()) {
        $this->Flash->error(__('Invalid username or password'));
    }
}
```

Add the template logic for your login action:

```
<!-- in /templates/Users/login.php -->
<div class="users form">
    <?= $this->Flash->render() ?>
    <h3>Login</h3>
    <?= $this->Form->create() ?>
    <fieldset>
        <legend><?= __('Please enter your username and password') ?></legend>
        <?= $this->Form->control('email', ['required' => true]) ?>
        <?= $this->Form->control('password', ['required' => true]) ?>
    </fieldset>
    <?= $this->Form->submit(__('Login')); ?>
    <?= $this->Form->end() ?>

    <?= $this->Html->link("Add User", ['action' => 'add']) ?>
</div>
```

Now login page will allow us to correctly login into the application. Test it by requesting any page of your site. After being redirected to the `/users/login` page, enter the email and password you picked previously when creating your user. You should be redirected successfully after login.

We need to add a couple more details to configure our application. We want all `view` and `index` pages accessible without logging in so we'll add this specific configuration in AppController:

```
// in src/Controller/AppController.php
public function beforeFilter(\Cake\Event\EventInterface $event)
{
    parent::beforeFilter($event);
    // for all controllers in our application, make index and view
    // actions public, skipping the authentication check
    $this->Authentication->addUnauthenticatedActions(['index', 'view']);
}
```

Note: If you don't have a user with a hashed password yet, comment the `$this->loadComponent('Authentication.Authentication')` line in your AppController and all other lines where Authentication is used. Then go to `/users/add` to create a new user picking email and password. Afterward, make sure to uncomment the lines we just temporarily commented!

Try it out by visiting `/articles/add` before logging in! Since this action is not allowed, you will be redirected to the login page. After logging in successfully, CakePHP will automatically redirect you back to `/articles/add`.

Logout

Add the logout action to the `UsersController` class:

```
// in src/Controller/UsersController.php
public function logout()
{
    $result = $this->Authentication->getResult();
    // regardless of POST or GET, redirect if user is logged in
    if ($result && $result->isValid()) {
        $this->Authentication->logout();
```

(continues on next page)

(continued from previous page)

```
        return $this->redirect(['controller' => 'Users', 'action' => 'login']);
    }
}
```

Now you can visit `/users/logout` to log out. You should then be sent to the login page.

Enabling Registrations

If you try to visit **/users/add** without being logged in, you will be redirected to the login page. We should fix that as we want to allow people to sign up for our application. In the `UsersController` fix the following line:

```
// Add to the beforeFilter method of UsersController
$this->Authentication->addUnauthenticatedActions(['login', 'add']);
```

The above tells `AuthenticationComponent` that the `add()` action of the `UsersController` does *not* require authentication or authorization. You may want to take the time to clean up the **Users/add.php** and remove the misleading links, or continue on to the next section. We won't be building out user editing, viewing or listing in this tutorial, but that is an exercise you can complete on your own.

Now that users can log in, we'll want to limit users to only edit articles that they created by *applying authorization policies*.

CMS Tutorial - Authorization

With users now able to login to our CMS, we want to apply authorization rules to ensure that each user only edits the posts they own. We'll use the authorization plugin[48] to do this.

Installing Authorization Plugin

Use composer to install the Authorization Plugin:

```
composer require "cakephp/authorization:^3.0"
```

Load the plugin by adding the following statement to the `bootstrap()` method in **src/Application.php**:

```
$this->addPlugin('Authorization');
```

Enabling the Authorization Plugin

The Authorization plugin integrates into your application as a middleware layer and optionally a component to make checking authorization easier. First, lets apply the middleware. In **src/Application.php** add the following to the class imports:

```
use Authorization\AuthorizationService;
use Authorization\AuthorizationServiceInterface;
use Authorization\AuthorizationServiceProviderInterface;
use Authorization\Middleware\AuthorizationMiddleware;
use Authorization\Policy\OrmResolver;
```

[48] https://book.cakephp.org/authorization/2

Add the `AuthorizationServiceProviderInterface` to the implemented interfaces on your application:

```
class Application extends BaseApplication
    implements AuthenticationServiceProviderInterface,
    AuthorizationServiceProviderInterface
```

Then add the following to your `middleware()` method:

```
// Add authorization **after** authentication
$middlewareQueue->add(new AuthorizationMiddleware($this));
```

The `AuthorizationMiddleware` will call a hook method on your application when it starts handling the request. This hook method allows your application to define the `AuthorizationService` it wants to use. Add the following method your **src/Application.php**:

```
public function getAuthorizationService(ServerRequestInterface $request):␣
↪AuthorizationServiceInterface
{
    $resolver = new OrmResolver();

    return new AuthorizationService($resolver);
}
```

The OrmResolver lets the authorization plugin find policy classes for ORM entities and queries. Other resolvers can be used to find policies for other resources types.

Next, lets add the `AuthorizationComponent` to `AppController`. In **src/Controller/AppController.php** add the following to the `initialize()` method:

```
$this->loadComponent('Authorization.Authorization');
```

Lastly we'll mark the add, login, and logout actions as not requiring authorization by adding the following to **src/Controller/UsersController.php**:

```
// In the add, login, and logout methods
$this->Authorization->skipAuthorization();
```

The `skipAuthorization()` method should be called in any controller action that should be accessible to all users even those who have not logged in yet.

Creating our First Policy

The Authorization plugin models authorization and permissions as Policy classes. These classes implement the logic to check whether or not a **identity** is allowed to **perform an action** on a given **resource**. Our **identity** is going to be our logged in user, and our **resources** are our ORM entities and queries. Lets use bake to generate a basic policy:

```
bin/cake bake policy --type entity Article
```

This will generate an empty policy class for our `Article` entity. You can find the generated policy in **src/Policy/ArticlePolicy.php**. Next update the policy to look like the following:

```
<?php
namespace App\Policy;
```

(continues on next page)

(continued from previous page)

```
use App\Model\Entity\Article;
use Authorization\IdentityInterface;

class ArticlePolicy
{
    public function canAdd(IdentityInterface $user, Article $article)
    {
        // All logged in users can create articles.
        return true;
    }

    public function canEdit(IdentityInterface $user, Article $article)
    {
        // logged in users can edit their own articles.
        return $this->isAuthor($user, $article);
    }

    public function canDelete(IdentityInterface $user, Article $article)
    {
        // logged in users can delete their own articles.
        return $this->isAuthor($user, $article);
    }

    protected function isAuthor(IdentityInterface $user, Article $article)
    {
        return $article->user_id === $user->getIdentifier();
    }
}
```

While we've defined some very simple rules, you can use as complex logic as your application requires in your policies.

Checking Authorization in the ArticlesController

With our policy created we can start checking authorization in each controller action. If we forget to check or skip authorization in an controller action the Authorization plugin will raise an exception letting us know we forgot to apply authorization. In **src/Controller/ArticlesController.php** add the following to the `add`, `edit` and `delete` methods:

```
public function add()
{
    $article = $this->Articles->newEmptyEntity();
    $this->Authorization->authorize($article);
    // Rest of the method
}

public function edit($slug)
{
    $article = $this->Articles
        ->findBySlug($slug)
        ->contain('Tags') // load associated Tags
        ->firstOrFail();
    $this->Authorization->authorize($article);
```

(continues on next page)

(continued from previous page)

```
    // Rest of the method.
}

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

    $article = $this->Articles->findBySlug($slug)->firstOrFail();
    $this->Authorization->authorize($article);
    // Rest of the method.
}
```

The `AuthorizationComponent::authorize()` method will use the current controller action name to generate the policy method to call. If you'd like to call a different policy method you can call `authorize` with the operation name:

```
$this->Authorization->authorize($article, 'update');
```

Lastly add the following to the `tags`, `view`, and `index` methods on the `ArticlesController`:

```
// View, index and tags actions are public methods
// and don't require authorization checks.
$this->Authorization->skipAuthorization();
```

Fixing the Add & Edit Actions

While we've blocked access to the edit action, we're still open to users changing the `user_id` attribute of articles during edit. We will solve these problems next. First up is the `add` action.

When creating articles, we want to fix the `user_id` to be the currently logged in user. Replace your add action with the following:

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

public function add()
{
    $article = $this->Articles->newEmptyEntity();
    $this->Authorization->authorize($article);

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

        // Changed: Set the user_id from the current user.
        $article->user_id = $this->request->getAttribute('identity')->getIdentifier();

        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.'));
    }
    $tags = $this->Articles->Tags->find('list')->all();
```

(continues on next page)

(continued from previous page)

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

Next we'll update the `edit` action. Replace the edit method with the following:

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

public function edit($slug)
{
    $article = $this->Articles
        ->findBySlug($slug)
        ->contain('Tags') // load associated Tags
        ->firstOrFail();
    $this->Authorization->authorize($article);

    if ($this->request->is(['post', 'put'])) {
        $this->Articles->patchEntity($article, $this->request->getData(), [
            // Added: Disable modification of user_id.
            'accessibleFields' => ['user_id' => false]
        ]);
        if ($this->Articles->save($article)) {
            $this->Flash->success(__('Your article has been updated.'));
            return $this->redirect(['action' => 'index']);
        }
        $this->Flash->error(__('Unable to update your article.'));
    }
    $tags = $this->Articles->Tags->find('list')->all();
    $this->set(compact('article', 'tags'));
}
```

Here we're modifying which properties can be mass-assigned, via the options for `patchEntity()`. See the *Changing Accessible Fields* section for more information. Remember to remove the `user_id` control from **templates/Articles/edit.php** as we no longer need it.

Wrapping Up

We've built a simple CMS application that allows users to login, post articles, tag them, explore posted articles by tag, and applied basic access control to articles. We've also added some nice UX improvements by leveraging the FormHelper and ORM capabilities.

Thank you for taking the time to explore CakePHP. Next, you should learn more about the *Database Access & ORM*, or you peruse the /topics.

CHAPTER 5

Contributing

There are a number of ways you can contribute to CakePHP. The following sections cover the various ways you can contribute to CakePHP:

Documentation

Contributing to the documentation is simple. The files are hosted on https://github.com/cakephp/docs. Feel free to fork the repo, add your changes/improvements/translations and give back by issuing a pull request. You can even edit the docs online with GitHub, without ever downloading the files – the "Improve this Doc" button on any given page will direct you to GitHub's online editor for that page.

CakePHP documentation is continuously integrated[49], and deployed after each pull request is merged.

Translations

Email the docs team (docs at cakephp dot org) or hop on IRC (#cakephp on freenode) to discuss any translation efforts you would like to participate in.

[49] https://en.wikipedia.org/wiki/Continuous_integration

New Translation Language

We want to provide translations that are as complete as possible. However, there may be times where a translation file is not up-to-date. You should always consider the English version as the authoritative version.

If your language is not in the current languages, please contact us through Github and we will consider creating a skeleton folder for it. The following sections are the first one you should consider translating as these files don't change often:

- index.rst
- intro.rst
- quickstart.rst
- installation.rst
- /intro folder
- /tutorials-and-examples folder

Reminder for Docs Administrators

The structure of all language folders should mirror the English folder structure. If the structure changes for the English version, we should apply those changes in the other languages.

For example, if a new English file is created in **en/file.rst**, we should:

- Add the file in all other languages : **fr/file.rst**, **zh/file.rst**, ...
- Delete the content, but keeping the `title`, `meta` information and eventual `toc-tree` elements. The following note will be added while nobody has translated the file:

```
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,...
```

Translator tips

- Browse and edit in the language you want the content to be translated to - otherwise you won't see what has already been translated.
- Feel free to dive right in if your chosen language already exists on the book.
- Use Informal Form[50].
- Translate both the content and the title at the same time.
- Do compare to the English content before submitting a correction (if you correct something, but don't integrate an 'upstream' change your submission won't be accepted).
- If you need to write an English term, wrap it in `<em>` tags. For example, "asdf asdf *Controller* asdf" or "asdf asdf Kontroller (*Controller*) asfd".
- Do not submit partial translations.
- Do not edit a section with a pending change.
- Do not use HTML entities[51] for accented characters, the book uses UTF-8.
- Do not significantly change the markup (HTML) or add new content.
- If the original content is missing some info, submit an edit for that first.

Documentation Formatting Guide

The new CakePHP documentation is written with ReST formatted text[52]. ReST (Re Structured Text) is a plain text markup syntax similar to markdown, or textile. To maintain consistency it is recommended that when adding to the CakePHP documentation you follow the guidelines here on how to format and structure your text.

Line Length

Lines of text should be wrapped at 80 columns. The only exception should be long URLs, and code snippets.

Headings and Sections

Section headers are created by underlining the title with punctuation characters at least the length of the text.

- # Is used to denote page titles.
- = Is used for sections in a page.
- - Is used for subsections.
- ~ Is used for sub-subsections.
- ^ Is used for sub-sub-subsections.

Headings should not be nested more than 5 levels deep. Headings should be preceded and followed by a blank line.

[50] https://en.wikipedia.org/wiki/Register#Linguistics
[51] https://en.wikipedia.org/wiki/List_of_XML_and_HTML_character_entity_references
[52] https://en.wikipedia.org/wiki/ReStructuredText

Paragraphs

Paragraphs are simply blocks of text, with all the lines at the same level of indentation. Paragraphs should be separated by one blank line.

Inline Markup

- One asterisk: *text* for emphasis (italics) We'll use it for general highlighting/emphasis.
  - `*text*`.
- Two asterisks: **text** for strong emphasis (boldface) We'll use it for working directories, bullet list subject, table names and excluding the following word "table".
  - `**/config/Migrations**`, `**articles**`, etc.
- Two backquotes: `text` for code samples We'll use it for names of method options, names of table columns, object names, excluding the following word "object" and for method/function names – include "()".
  - ``` ``cascadeCallbacks`` ```, ``` ``true`` ```, ``` ``id`` ```, ``` ``PagesController`` ```, ``` ``config()`` ```, etc.

If asterisks or backquotes appear in running text and could be confused with inline markup delimiters, they have to be escaped with a backslash.

Inline markup has a few restrictions:

- It **may not** be nested.
- Content may not start or end with whitespace: `* text*` is wrong.
- Content must be separated from surrounding text by non-word characters. Use a backslash escaped space to work around that: `onelong\ *bolded*\ word`.

Lists

List markup is very similar to markdown. Unordered lists are indicated by starting a line with a single asterisk and a space. Numbered lists can be created with either numerals, or # for auto numbering:

```
* This is a bullet
* So is this. But this line
  has two lines.

1. First line
2. Second line

#. Automatic numbering
#. Will save you some time.
```

Indented lists can also be created, by indenting sections and separating them with an empty line:

```
* First line
* Second line

    * Going deeper
    * Whoah

* Back to the first level.
```

Definition lists can be created by doing the following:

```
term
    definition
CakePHP
    An MVC framework for PHP
```

Terms cannot be more than one line, but definitions can be multi-line and all lines should be indented consistently.

Links

There are several kinds of links, each with their own uses.

External Links

Links to external documents can be done with the following:

```
`External Link to php.net <https://php.net>`_
```

The resulting link would look like this: External Link to php.net[53]

Links to Other Pages

:doc:

Other pages in the documentation can be linked to using the `:doc:` role. You can link to the specified document using either an absolute or relative path reference. You should omit the `.rst` extension. For example, if the reference `:doc:`form`` appears in the document `core-helpers/html`, then the link references `core-helpers/form`. If the reference was `:doc:`/core-helpers``, it would always reference `/core-helpers` regardless of where it was used.

Cross Referencing Links

:ref:

You can cross reference any arbitrary title in any document using the `:ref:` role. Link label targets must be unique across the entire documentation. When creating labels for class methods, it's best to use `class-method` as the format for your link label.

The most common use of labels is above a title. Example:

```
.. _label-name:

Section heading
---------------

More content here.
```

Elsewhere you could reference the above section using `:ref:`label-name``. The link's text would be the title that the link preceded. You can also provide custom link text using `:ref:`Link text <label-name>``.

[53] https://php.net

Prevent Sphinx to Output Warnings

Sphinx will output warnings if a file is not referenced in a toc-tree. It's a great way to ensure that all files have a link directed to them, but sometimes, you don't need to insert a link for a file, eg. for our *epub-contents* and *pdf-contents* files. In those cases, you can add `:orphan:` at the top of the file, to suppress warnings that the file is not in the toc-tree.

Describing Classes and their Contents

The CakePHP documentation uses the phpdomain[54] to provide custom directives for describing PHP objects and constructs. Using these directives and roles is required to give proper indexing and cross referencing features.

Describing Classes and Constructs

Each directive populates the index, and or the namespace index.

`.. php:global::` name
: This directive declares a new PHP global variable.

`.. php:function::` name(signature)
: Defines a new global function outside of a class.

`.. php:const::` name
: This directive declares a new PHP constant, you can also use it nested inside a class directive to create class constants.

`.. php:exception::` name
: This directive declares a new Exception in the current namespace. The signature can include constructor arguments.

`.. php:class::` name
: Describes a class. Methods, attributes, and constants belonging to the class should be inside this directive's body:

```
.. php:class:: MyClass

    Class description

    .. php:method:: method($argument)

    Method description
```

Attributes, methods and constants don't need to be nested. They can also just follow the class declaration:

```
.. php:class:: MyClass

    Text about the class

.. php:method:: methodName()

    Text about the method
```

See also:

php:method, *php:attr*, *php:const*

[54] https://pypi.org/project/sphinxcontrib-phpdomain/

.. **php:method::** name(signature)

Describe a class method, its arguments, return value, and exceptions:

```
.. php:method:: instanceMethod($one, $two)

    :param string $one: The first parameter.
    :param string $two: The second parameter.
    :returns: An array of stuff.
    :throws: InvalidArgumentException

   This is an instance method.
```

.. **php:staticmethod::** ClassName::methodName(signature)

Describe a static method, its arguments, return value and exceptions, see *php:method* for options.

.. **php:attr::** name

Describe an property/attribute on a class.

Prevent Sphinx to Output Warnings

Sphinx will output warnings if a function is referenced in multiple files. It's a great way to ensure that you did not add a function two times, but sometimes, you actually want to write a function in two or more files, eg. *debug object* is referenced in */development/debugging* and in */core-libraries/global-constants-and-functions*. In this case, you can add `:noindex:` under the function debug to suppress warnings. Keep only one reference **without** `:no-index:` to still have the function referenced:

```
.. php:function:: debug(mixed $var, boolean $showHtml = null, $showFrom = true)
    :noindex:
```

Cross Referencing

The following roles refer to PHP objects and links are generated if a matching directive is found:

:php:func:

Reference a PHP function.

:php:global:

Reference a global variable whose name has $ prefix.

:php:const:

Reference either a global constant, or a class constant. Class constants should be preceded by the owning class:

```
DateTime has an :php:const:`DateTime::ATOM` constant.
```

:php:class:

Reference a class by name:

```
:php:class:`ClassName`
```

:php:meth:

Reference a method of a class. This role supports both kinds of methods:

```
:php:meth:`DateTime::setDate`
:php:meth:`Classname::staticMethod`
```

:php:attr:

Reference a property on an object:

```
:php:attr:`ClassName::$propertyName`
```

:php:exc:

Reference an exception.

Source Code

Literal code blocks are created by ending a paragraph with `::`. The literal block must be indented, and like all paragraphs be separated by single lines:

```
This is a paragraph::

    while ($i--) {
        doStuff()
    }

This is regular text again.
```

Literal text is not modified or formatted, save that one level of indentation is removed.

Notes and Warnings

There are often times when you want to inform the reader of an important tip, special note or a potential hazard. Admonitions in sphinx are used for just that. There are fives kinds of admonitions.

- `.. tip::` Tips are used to document or re-iterate interesting or important information. The content of the directive should be written in complete sentences and include all appropriate punctuation.
- `.. note::` Notes are used to document an especially important piece of information. The content of the directive should be written in complete sentences and include all appropriate punctuation.
- `.. warning::` Warnings are used to document potential stumbling blocks, or information pertaining to security. The content of the directive should be written in complete sentences and include all appropriate punctuation.
- `.. versionadded:: X.Y.Z` "Version added" admonitions are used to display notes specific to new features added at a specific version, `X.Y.Z` being the version on which the said feature was added.
- `.. deprecated:: X.Y.Z` As opposed to "version added" admonitions, "deprecated" admonition are used to notify of a deprecated feature, `X.Y.Z` being the version on which the said feature was deprecated.

All admonitions are made the same:

```
.. note::

    Indented and preceded and followed by a blank line. Just like a
    paragraph.

This text is not part of the note.
```

Samples

Tip: This is a helpful tid-bit you probably forgot.

Note: You should pay attention here.

Warning: It could be dangerous.

New in version 4.0.0: This awesome feature was added in version 4.0.0

Deprecated since version 4.0.1: This old feature was deprecated on version 4.0.1

Tickets

Getting feedback and help from the community in the form of tickets is an extremely important part of the CakePHP development process. All of CakePHP's tickets are hosted on GitHub[55].

Reporting Bugs

Well written bug reports are very helpful. There are a few steps to help create the best bug report possible:

- **Do**: Please search[56] for a similar existing ticket, and ensure someone hasn't already reported your issue, or that it hasn't already been fixed in the repository.
- **Do**: Please include detailed instructions on **how to reproduce the bug**. This could be in the form of a test-case or a snippet of code that demonstrates the issue. Not having a way to reproduce an issue means it's less likely to get fixed.
- **Do**: Please give as many details as possible about your environment: (OS, PHP version, CakePHP version).
- **Don't**: Please don't use the ticket system to ask support questions. Both the support channel on the CakePHP Slack workspace[57] and the #cakephp IRC channel on Freenode[58] have many developers available to help answer your questions. Also have a look at Stack Overflow[59] or the official CakePHP forum[60].

[55] https://github.com/cakephp/cakephp/issues
[56] https://github.com/cakephp/cakephp/search?q=it+is+broken&ref=cmdform&type=Issues
[57] https://cakesf.herokuapp.com
[58] https://webchat.freenode.net
[59] https://stackoverflow.com/questions/tagged/cakephp
[60] https://discourse.cakephp.org

Reporting Security Issues

If you've found a security issue in CakePHP, please use the following procedure instead of the normal bug reporting system. Instead of using the bug tracker, mailing list or IRC please send an email to **security [at] cakephp.org**. Emails sent to this address go to the CakePHP core team on a private mailing list.

For each report, we try to first confirm the vulnerability. Once confirmed, the CakePHP team will take the following actions:

- Acknowledge to the reporter that we've received the issue, and are working on a fix. We ask that the reporter keep the issue confidential until we announce it.
- Get a fix/patch prepared.
- Prepare a post describing the vulnerability, and the possible exploits.
- Release new versions of all affected versions.
- Prominently feature the problem in the release announcement.

Code

Patches and pull requests are a great way to contribute code back to CakePHP. Pull requests can be created in GitHub, and are preferred over patch files in ticket comments.

Initial Setup

Before working on patches for CakePHP, it's a good idea to get your environment setup. You'll need the following software:

- Git
- PHP 8.1 or greater
- PHPUnit 5.7.0 or greater

Set up your user information with your name/handle and working email address:

```
git config --global user.name 'Bob Barker'
git config --global user.email 'bob.barker@example.com'
```

Note: If you are new to Git, we highly recommend you to read the excellent and free ProGit[61] book.

Get a clone of the CakePHP source code from GitHub:

- If you don't have a GitHub[62] account, create one.
- Fork the CakePHP repository[63] by clicking the **Fork** button.

After your fork is made, clone your fork to your local machine:

```
git clone git@github.com:YOURNAME/cakephp.git
```

[61] https://git-scm.com/book/
[62] https://github.com
[63] https://github.com/cakephp/cakephp

Add the original CakePHP repository as a remote repository. You'll use this later to fetch changes from the CakePHP repository. This will let you stay up to date with CakePHP:

```
cd cakephp
git remote add upstream git://github.com/cakephp/cakephp.git
```

Now that you have CakePHP setup you should be able to define a `$test` *database connection*, and *run all the tests*.

Working on a Patch

Each time you want to work on a bug, feature or enhancement create a topic branch.

The branch you create should be based on the version that your fix/enhancement is for. For example if you are fixing a bug in `3.x` you would want to use the `master` branch as the base for your branch. If your change is a bug fix for the 2.x release series, you should use the `2.x` branch:

```
# fixing a bug on 3.x
git fetch upstream
git checkout -b ticket-1234 upstream/master

# fixing a bug on 2.x
git fetch upstream
git checkout -b ticket-1234 upstream/2.x
```

Tip: Use a descriptive name for your branch. Referencing the ticket or feature name is a good convention. Examples include `ticket-1234` and `feature-awesome`.

The above will create a local branch based on the upstream (CakePHP) 2.x branch. Work on your fix, and make as many commits as you need; but keep in mind the following:

- Follow the *Coding Standards*.
- Add a test case to show the bug is fixed, or that the new feature works.
- Keep your commits logical, and write clear commit messages that provide context on what you changed and why.

Submitting a Pull Request

Once your changes are done and you're ready for them to be merged into CakePHP, you'll want to update your branch:

```
# Rebase fix on top of master
git checkout master
git fetch upstream
git merge upstream/master
git checkout <branch_name>
git rebase master
```

This will fetch + merge in any changes that have happened in CakePHP since you started. It will then rebase - or replay your changes on top of the current code. You might encounter a conflict during the `rebase`. If the rebase quits early you can see which files are conflicted/un-merged with `git status`. Resolve each conflict, and then continue the rebase:

```
git add <filename> # do this for each conflicted file.
git rebase --continue
```

Check that all your tests continue to pass. Then push your branch to your fork:

```
git push origin <branch-name>
```

If you've rebased after pushing your branch, you'll need to use force push:

```
git push --force origin <branch-name>
```

Once your branch is on GitHub, you can submit a pull request on GitHub.

Choosing Where Your Changes will be Merged Into

When making pull requests you should make sure you select the correct base branch, as you cannot edit it once the pull request is created.

- If your change is a **bugfix** and doesn't introduce new functionality and only corrects existing behavior that is present in the current release. Then choose **master** as your merge target.
- If your change is a **new feature** or an addition to the framework, then you should choose the branch with the next version number. For example if the current stable release is `4.0.0`, the branch accepting new features will be `4.next`.
- If your change is a breaks existing functionality, or APIs then you'll have to choose then next major release. For example, if the current release is `4.0.0` then the next time existing behavior can be broken will be in `5.x` so you should target that branch.

Note: Remember that all code you contribute to CakePHP will be licensed under the MIT License, and the Cake Software Foundation[64] will become the owner of any contributed code. Contributors should follow the CakePHP Community Guidelines[65].

All bug fixes merged into a maintenance branch will also be merged into upcoming releases periodically by the core team.

Coding Standards

CakePHP developers will use the PSR-12 coding style guide[66] in addition to the following rules as coding standards.

It is recommended that others developing CakeIngredients follow the same standards.

You can use the CakePHP Code Sniffer[67] to check that your code follows required standards.

[64] https://cakefoundation.org/old
[65] https://cakephp.org/get-involved
[66] https://www.php-fig.org/psr/psr-12/
[67] https://github.com/cakephp/cakephp-codesniffer

Adding New Features

No new features should be added, without having their own tests – which should be passed before committing them to the repository.

IDE Setup

Please make sure your IDE is set up to "trim right" on whitespaces. There should be no trailing spaces per line.

Most modern IDEs also support an `.editorconfig` file. The CakePHP app skeleton ships with it by default. It already contains best practise defaults.

We recommend to use the IdeHelper[68] plugin if you want to maximize IDE compatibility. It will assist to keep the annotations up-to-date which will make the IDE fully understand how all classes work together and provides better type-hinting and auto-completion.

Indentation

Four spaces will be used for indentation.

So, indentation should look like this:

```
// base level
    // level 1
        // level 2
    // level 1
// base level
```

Or:

```
$booleanVariable = true;
$stringVariable = 'moose';
if ($booleanVariable) {
    echo 'Boolean value is true';
    if ($stringVariable === 'moose') {
        echo 'We have encountered a moose';
    }
}
```

In cases where you're using a multi-line function call use the following guidelines:

- Opening parenthesis of a multi-line function call must be the last content on the line.
- Only one argument is allowed per line in a multi-line function call.
- Closing parenthesis of a multi-line function call must be on a line by itself.

As an example, instead of using the following formatting:

```
$matches = array_intersect_key($this->_listeners,
                array_flip(preg_grep($matchPattern,
                    array_keys($this->_listeners), 0)));
```

Use this instead:

[68] https://github.com/dereuromark/cakephp-ide-helper

```
$matches = array_intersect_key(
    $this->_listeners,
    array_flip(
        preg_grep($matchPattern, array_keys($this->_listeners), 0)
    )
);
```

Line Length

It is recommended to keep lines at approximately 100 characters long for better code readability. A limit of 80 or 120 characters makes it necessary to distribute complex logic or expressions by function, as well as give functions and objects shorter, more expressive names. Lines must not be longer than 120 characters.

In short:

- 100 characters is the soft limit.
- 120 characters is the hard limit.

Control Structures

Control structures are for example "`if`", "`for`", "`foreach`", "`while`", "`switch`" etc. Below, an example with "`if`":

```
if ((expr_1) || (expr_2)) {
    // action_1;
} elseif (!(expr_3) && (expr_4)) {
    // action_2;
} else {
    // default_action;
}
```

- In the control structures there should be 1 (one) space before the first parenthesis and 1 (one) space between the last parenthesis and the opening bracket.
- Always use curly brackets in control structures, even if they are not needed. They increase the readability of the code, and they give you fewer logical errors.
- Opening curly brackets should be placed on the same line as the control structure. Closing curly brackets should be placed on new lines, and they should have same indentation level as the control structure. The statement included in curly brackets should begin on a new line, and code contained within it should gain a new level of indentation.
- Inline assignments should not be used inside of the control structures.

```
// wrong = no brackets, badly placed statement
if (expr) statement;

// wrong = no brackets
if (expr)
    statement;

// good
if (expr) {
    statement;
```

(continues on next page)

(continued from previous page)

```
}

// wrong = inline assignment
if ($variable = Class::function()) {
    statement;
}

// good
$variable = Class::function();
if ($variable) {
    statement;
}
```

Ternary Operator

Ternary operators are permissible when the entire ternary operation fits on one line. Longer ternaries should be split into `if else` statements. Ternary operators should not ever be nested. Optionally parentheses can be used around the condition check of the ternary for clarity:

```
// Good, simple and readable
$variable = isset($options['variable']) ? $options['variable'] : true;

// Nested ternaries are bad
$variable = isset($options['variable']) ? isset($options['othervar']) ? true : false :␣
↪false;
```

Template Files

In template files developers should use keyword control structures. Keyword control structures are easier to read in complex template files. Control structures can either be contained in a larger PHP block, or in separate PHP tags:

```
<?php
if ($isAdmin):
    echo '<p>You are the admin user.</p>';
endif;
?>
<p>The following is also acceptable:</p>
<?php if ($isAdmin): ?>
    <p>You are the admin user.</p>
<?php endif; ?>
```

Comparison

Always try to be as strict as possible. If a non-strict test is deliberate it might be wise to comment it as such to avoid confusing it for a mistake.

For testing if a variable is null, it is recommended to use a strict check:

```
if ($value === null) {
    // ...
}
```

The value to check against should be placed on the right side:

```
// not recommended
if (null === $this->foo()) {
    // ...
}

// recommended
if ($this->foo() === null) {
    // ...
}
```

Function Calls

Functions should be called without space between function's name and starting parenthesis. There should be one space between every parameter of a function call:

```
$var = foo($bar, $bar2, $bar3);
```

As you can see above there should be one space on both sides of equals sign (=).

Method Definition

Example of a method definition:

```
public function someFunction($arg1, $arg2 = '')
{
    if (expr) {
        statement;
    }

    return $var;
}
```

Parameters with a default value, should be placed last in function definition. Try to make your functions return something, at least `true` or `false`, so it can be determined whether the function call was successful:

```
public function connection($dns, $persistent = false)
{
    if (is_array($dns)) {
        $dnsInfo = $dns;
```

(continues on next page)

(continued from previous page)

```
    } else {
        $dnsInfo = BD::parseDNS($dns);
    }

    if (!($dnsInfo) || !($dnsInfo['phpType'])) {
        return $this->addError();
    }

    return true;
}
```

There are spaces on both side of the equals sign.

Bail Early

Try to avoid unnecessary nesting by bailing early:

```
public function run(array $data)
{
    ...
    if (!$success) {
        return false;
    }

    ...
}

public function check(array $data)
{
    ...
    if (!$success) {
        throw new RuntimeException(...);
    }

    ...
}
```

This helps to keep the logic sequential which improves readability.

Typehinting

Arguments that expect objects, arrays or callbacks (callable) can be typehinted. We only typehint public methods, though, as typehinting is not cost-free:

```
/**
 * 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.
```

(continues on next page)

(continued from previous page)

```
 */
public function foo(Table $table, array $array, callable $callback, $boolean)
{
}
```

Here `$table` must be an instance of `\Cake\ORM\Table`, `$array` must be an `array` and `$callback` must be of type `callable` (a valid callback).

Note that if you want to allow `$array` to be also an instance of `\ArrayObject` you should not typehint as `array` accepts only the primitive type:

```
/**
 * Some method description.
 *
 * @param array|\ArrayObject $array Some array value.
 */
public function foo($array)
{
}
```

Anonymous Functions (Closures)

Defining anonymous functions follows the PSR-12[69] coding style guide, where they are declared with a space after the *function* keyword, and a space before and after the *use* keyword:

```
$closure = function ($arg1, $arg2) use ($var1, $var2) {
    // code
};
```

Method Chaining

Method chaining should have multiple methods spread across separate lines, and indented with four spaces:

```
$email->from('foo@example.com')
    ->to('bar@example.com')
    ->subject('A great message')
    ->send();
```

Commenting Code

All comments should be written in English, and should in a clear way describe the commented block of code.

Comments can include the following phpDocumentor[70] tags:

- @deprecated[71] Using the `@version <vector> <description>` format, where `version` and `description` are mandatory. Version refers to the one it got deprecated in.
- @example[72]

[69] https://www.php-fig.org/psr/psr-12/
[70] https://phpdoc.org
[71] https://docs.phpdoc.org/latest/guide/references/phpdoc/tags/deprecated.html
[72] https://docs.phpdoc.org/latest/guide/references/phpdoc/tags/example.html

- @ignore[73]
- @internal[74]
- @link[75]
- @see[76]
- @since[77]
- @version[78]

PhpDoc tags are very much like JavaDoc tags in Java. Tags are only processed if they are the first thing in a DocBlock line, for example:

```
/**
 * Tag example.
 *
 * @author this tag is parsed, but this @version is ignored
 * @version 1.0 this tag is also parsed
 */
```

```
/**
 * Example of inline phpDoc tags.
 *
 * This function works hard with foo() to rule the world.
 *
 * @return void
 */
function bar()
{
}

/**
 * Foo function.
 *
 * @return void
 */
function foo()
{
}
```

Comment blocks, with the exception of the first block in a file, should always be preceded by a newline.

[73] https://docs.phpdoc.org/latest/guide/references/phpdoc/tags/ignore.html
[74] https://docs.phpdoc.org/latest/guide/references/phpdoc/tags/internal.html
[75] https://docs.phpdoc.org/latest/guide/references/phpdoc/tags/link.html
[76] https://docs.phpdoc.org/latest/guide/references/phpdoc/tags/see.html
[77] https://docs.phpdoc.org/latest/guide/references/phpdoc/tags/since.html
[78] https://docs.phpdoc.org/latest/guide/references/phpdoc/tags/version.html

Variable Types

Variable types for use in DocBlocks:

Type
: Description

mixed
: A variable with undefined (or multiple) type.

int
: Integer type variable (whole number).

float
: Float type (point number).

bool
: Logical type (true or false).

string
: String type (any value in " " or ' ').

null
: Null type. Usually used in conjunction with another type.

array
: Array type.

object
: Object type. A specific class name should be used if possible.

resource
: Resource type (returned by for example mysql_connect()). Remember that when you specify the type as mixed, you should indicate whether it is unknown, or what the possible types are.

callable
: Callable function.

You can also combine types using the pipe char:

```
int|bool
```

For more than two types it is usually best to just use `mixed`.

When returning the object itself (for example, for chaining), one should use `$this` instead:

```
/**
 * Foo function.
 *
 * @return $this
 */
public function foo()
{
    return $this;
}
```

Including Files

`include`, `require`, `include_once` and `require_once` do not have parentheses:

```
// wrong = parentheses
require_once('ClassFileName.php');
require_once ($class);

// good = no parentheses
require_once 'ClassFileName.php';
require_once $class;
```

When including files with classes or libraries, use only and always the require_once[79] function.

PHP Tags

Always use long tags (`<?php ?>`) instead of short tags (`<? ?>`). The short echo should be used in template files where appropriate.

Short Echo

The short echo should be used in template files in place of `<?php echo`. It should be immediately followed by a single space, the variable or function value to `echo`, a single space, and the php closing tag:

```
// wrong = semicolon, no spaces
<td><?=$name;?></td>

// good = spaces, no semicolon
<td><?= $name ?></td>
```

As of PHP 5.4 the short echo tag (`<?=`) is no longer to be consider a 'short tag' is always available regardless of the `short_open_tag` ini directive.

Naming Convention

Functions

Write all functions in camelBack:

```
function longFunctionName()
{
}
```

[79] https://php.net/require_once

Classes

Class names should be written in CamelCase, for example:

```
class ExampleClass
{
}
```

Variables

Variable names should be as descriptive as possible, but also as short as possible. All variables should start with a lowercase letter, and should be written in camelBack in case of multiple words. Variables referencing objects should in some way associate to the class the variable is an object of. Example:

```
$user = 'John';
$users = ['John', 'Hans', 'Arne'];

$dispatcher = new Dispatcher();
```

Member Visibility

Use PHP's `public`, `protected` and `private` keywords for methods and variables.

Example Addresses

For all example URL and mail addresses use "example.com", "example.org" and "example.net", for example:

- Email: someone@example.com
- WWW: http://www.example.com
- FTP: ftp://ftp.example.com

The "example.com" domain name has been reserved for this (see **RFC 2606**[80]) and is recommended for use in documentation or as examples.

Files

File names which do not contain classes should be lowercased and underscored, for example:

```
long_file_name.php
```

[80] https://datatracker.ietf.org/doc/html/rfc2606.html

Casting

For casting we use:

Type
: Description

(bool)
: Cast to boolean.

(int)
: Cast to integer.

(float)
: Cast to float.

(string)
: Cast to string.

(array)
: Cast to array.

(object)
: Cast to object.

Please use `(int)$var` instead of `intval($var)` and `(float)$var` instead of `floatval($var)` when applicable.

Constants

Constants should be defined in capital letters:

```
define('CONSTANT', 1);
```

If a constant name consists of multiple words, they should be separated by an underscore character, for example:

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

Enums

Enum cases are defined in `CamelCase` style:

```
enum ArticleStatus: string
{
    case Published = 'Y';
    case NotPublishedYet = 'N';
}
```

Careful when using empty()/isset()

While `empty()` often seems correct to use, it can mask errors and cause unintended effects when `'0'` and `0` are given. When variables or 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'])) {
        // ...
```

(continues on next page)

(continued from previous page)

```
    }

    // Recommended
    if ($array['key'] !== null) {
        // ...
    }
}
```

Backwards Compatibility Guide

Ensuring that you can upgrade your applications easily and smoothly is important to us. That's why we only break compatibility at major release milestones. You might be familiar with semantic versioning[81], which is the general guideline we use on all CakePHP projects. In short, semantic versioning means that only major releases (such as 2.0, 3.0, 4.0) can break backwards compatibility. Minor releases (such as 2.1, 3.1, 3.2) may introduce new features, but are not allowed to break compatibility. Bug fix releases (such as 2.1.2, 3.0.1) do not add new features, but fix bugs or enhance performance only.

Note: Deprecations are removed with the next major version of the framework. It is advised that you adapt to deprecations as they are introduced to ensure future upgrades are easier.

To clarify what changes you can expect in each release tier we have more detailed information for developers using CakePHP, and for developers working on CakePHP that helps set expectations of what can be done in minor releases. Major releases can have as many breaking changes as required.

Migration Guides

For each major and minor release, the CakePHP team will provide a migration guide. These guides explain the new features and any breaking changes that are in each release. They can be found in the *Appendices* section of the cookbook.

Using CakePHP

If you are building your application with CakePHP, the following guidelines explain the stability you can expect.

Interfaces

Outside of major releases, interfaces provided by CakePHP will **not** have any existing methods changed. New methods may be added, but no existing methods will be changed.

[81] https://semver.org/

Classes

Classes provided by CakePHP can be constructed and have their public methods and properties used by application code and outside of major releases backwards compatibility is ensured.

Note: Some classes in CakePHP are marked with the `@internal` API doc tag. These classes are **not** stable and do not have any backwards compatibility promises.

In minor releases, new methods may be added to classes, and existing methods may have new arguments added. Any new arguments will have default values, but if you've overridden methods with a differing signature you may see fatal errors. Methods that have new arguments added will be documented in the migration guide for that release.

The following table outlines several use cases and what compatibility you can expect from CakePHP:

If you…	Backwards compatibility?
Typehint against the class	Yes
Create a new instance	Yes
Extend the class	Yes
Access a public property	Yes
Call a public method	Yes
Extend a class and…	
Override a public property	Yes
Access a protected property	No[1]
Override a protected property	No[1]
Override a protected method	No[1]
Call a protected method	No[1]
Add a public property	No
Add a public method	No
Add an argument to an overridden method	No[1]
Add a default argument value to an existing method argument	Yes

Working on CakePHP

If you are helping make CakePHP even better please keep the following guidelines in mind when adding/changing functionality:

In a minor release you can:

[1] Your code *may* be broken by minor releases. Check the migration guide for details.

In a minor release can you...	
Classes	
Remove a class	No
Remove an interface	No
Remove a trait	No
Make final	No
Make abstract	No
Change name	Yes[2]
Properties	
Add a public property	Yes
Remove a public property	No
Add a protected property	Yes
Remove a protected property	Yes[3]
Methods	
Add a public method	Yes
Remove a public method	No
Add a protected method	Yes
Move to parent class	Yes
Remove a protected method	Yes[Page 107, 3]
Reduce visibility	No
Change method name	Yes[2]
Add a new argument with default value	Yes
Add a new required argument to an existing method.	No
Remove a default value from an existing argument	No
Change method type void	Yes

Deprecations

In each minor release, features may be deprecated. If features are deprecated, API documentation and runtime warnings will be added. Runtime errors help you locate code that needs to be updated before it breaks. If you wish to disable runtime warnings you can do so using the `Error.errorLevel` configuration value:

```
// in config/app.php
// ...
'Error' => [
    'errorLevel' => E_ALL ^ E_USER_DEPRECATED,
]
// ...
```

Will disable runtime deprecation warnings.

[2] You can change a class/method name as long as the old name remains available. This is generally avoided unless renaming has significant benefit.

[3] Avoid whenever possible. Any removals need to be documented in the migration guide.

Experimental Features

Experimental features are **not included** in the above backwards compatibility promises. Experimental features can have breaking changes made in minor releases as long as they remain experimental. Experimental features can be identified by the warning in the book and the usage of `@experimental` in the API documentation.

Experimental features are intended to help gather feedback on how a feature works before it becomes stable. Once the interfaces and behavior has been vetted with the community the experimental flags will be removed.

CHAPTER 6

Installation

CakePHP has a few system requirements:

- HTTP Server. For example: Apache. Having mod_rewrite is preferred, but by no means required. You can also use nginx, or Microsoft IIS if you prefer.
- Minimum PHP 8.1 (**8.2** supported).
- mbstring PHP extension
- intl PHP extension
- SimpleXML PHP extension
- PDO PHP extension

Note: In XAMPP, intl extension is included but you have to uncomment `extension=php_intl.dll` (or `extension=intl`) in **php.ini** and restart the server through the XAMPP Control Panel.

In WAMP, the intl extension is "activated" by default but not working. To make it work you have to go to php folder (by default) **C:\wamp\bin\php\php{version}**, copy all the files that looks like **icu*.dll** and paste them into the apache bin directory **C:\wamp\bin\apache\apache{version}\bin**. Then restart all services and it should be OK.

While a database engine isn't required, we imagine that most applications will utilize one. CakePHP supports a variety of database storage engines:

- MySQL (5.7 or higher)
- MariaDB (10.1 or higher)
- PostgreSQL (9.6 or higher)
- Microsoft SQL Server (2012 or higher)
- SQLite 3

The Oracle database is supported through the Driver for Oracle Database[82] community plugin.

Note: All built-in drivers require PDO. You should make sure you have the correct PDO extensions installed.

Installing CakePHP

Before starting you should make sure that your PHP version is up to date:

```
php -v
```

You should have PHP 8.1 (CLI) or higher. Your webserver's PHP version must also be of 8.1 or higher, and should be the same version your command line interface (CLI) uses.

Installing Composer

CakePHP uses Composer[83], a dependency management tool, as the officially supported method for installation.

- Installing Composer on Linux and macOS

  1. Run the installer script as described in the official Composer documentation[84] and follow the instructions to install Composer.
  2. Execute the following command to move the composer.phar to a directory that is in your path:

     ```
     mv composer.phar /usr/local/bin/composer
     ```

- Installing Composer on Windows

  For Windows systems, you can download Composer's Windows installer here[85]. Further instructions for Composer's Windows installer can be found within the README here[86].

Create a CakePHP Project

You can create a new CakePHP application using composer's `create-project` command:

```
composer create-project --prefer-dist cakephp/app:~5.0 my_app_name
```

Once Composer finishes downloading the application skeleton and the core CakePHP library, you should have a functioning CakePHP application installed via Composer. Be sure to keep the composer.json and composer.lock files with the rest of your source code.

You can now visit the path to where you installed your CakePHP application and see the default home page. To change the content of this page, edit **templates/Pages/home.php**.

Although composer is the recommended installation method, there are pre-installed downloads available on Github[87]. Those downloads contain the app skeleton with all vendor packages installed. Also it includes the `composer.phar` so you have everything you need for further use.

[82] https://github.com/CakeDC/cakephp-oracle-driver
[83] https://getcomposer.org
[84] https://getcomposer.org/download/
[85] https://github.com/composer/windows-setup/releases/
[86] https://github.com/composer/windows-setup
[87] https://github.com/cakephp/cakephp/tags

Keeping Up To Date with the Latest CakePHP Changes

By default this is what your application **composer.json** looks like:

```
"require": {
    "cakephp/cakephp": "5.0.*"
}
```

Each time you run `php composer.phar update` you will receive patch releases for this minor version. You can instead change this to `^5.0` to also receive the latest stable minor releases of the `5.x` branch.

Installation using DDEV

Another quick way to install CakePHP is via DDEV[88]. It is an open source tool for launching local web development environments.

If you want to configure a new project, you just need:

```
mkdir my-cakephp-app
cd my-cakephp-app
ddev config --project-type=cakephp --docroot=webroot
ddev composer create --prefer-dist cakephp/app:~5.0
ddev launch
```

If you have an existing project:

```
git clone <your-cakephp-repo>
cd <your-cakephp-project>
ddev config --project-type=cakephp --docroot=webroot
ddev composer install
ddev launch
```

Please check DDEV Docs[89] for details on how to install / update DDEV.

Note: IMPORTANT: This is not a deployment script. It is aimed to help developers to set up a development environment quickly. It is not intended for production environments.

Permissions

CakePHP uses the **tmp** directory for a number of different operations. Model descriptions, cached views, and session information are a few examples. The **logs** directory is used to write log files by the default `FileLog` engine.

As such, make sure the directories **logs**, **tmp** and all its subdirectories in your CakePHP installation are writable by the web server user. Composer's installation process makes **tmp** and its subfolders globally writeable to get things up and running quickly but you can update the permissions for better security and keep them writable only for the web server user.

One common issue is that **logs** and **tmp** directories and subdirectories must be writable both by the web server and the command line user. On a UNIX system, if your web server user is different from your command line user, you can run

[88] https://ddev.com/
[89] https://ddev.readthedocs.io/

the following commands from your application directory just once in your project to ensure that permissions will be setup properly:

```
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
```

In order to use the CakePHP console tools, you need to ensure that `bin/cake` file is executable. On *nix or macOS, you can execute:

```
chmod +x bin/cake
```

On Windows, the **.bat** file should be executable already. If you are using a Vagrant, or any other virtualized environment, any shared directories need to be shared with execute permissions (Please refer to your virtualized environment's documentation on how to do this).

If, for whatever reason, you cannot change the permissions of the `bin/cake` file, you can run the CakePHP console with:

```
php bin/cake.php
```

Development Server

A development installation is the fastest way to setup CakePHP. In this example, we use CakePHP's console to run PHP's built-in web server which will make your application available at **http://host:port**. From the app directory, execute:

```
bin/cake server
```

By default, without any arguments provided, this will serve your application at **http://localhost:8765/**.

If there is conflict with **localhost** or port 8765, you can tell the CakePHP console to run the web server on a specific host and/or port utilizing the following arguments:

```
bin/cake server -H 192.168.13.37 -p 5673
```

This will serve your application at **http://192.168.13.37:5673/**.

That's it! Your CakePHP application is up and running without having to configure a web server.

Note: Try `bin/cake server -H 0.0.0.0` if the server is unreachable from other hosts.

> **Warning:** The development server should *never* be used in a production environment. It is only intended as a basic development server.

If you'd prefer to use a real web server, you should be able to move your CakePHP install (including the hidden files) inside your web server's document root. You should then be able to point your web-browser at the directory you moved the files into and see your application in action.

Production

A production installation is a more flexible way to setup CakePHP. Using this method allows an entire domain to act as a single CakePHP application. This example will help you install CakePHP anywhere on your filesystem and make it available at http://www.example.com. Note that this installation may require the rights to change the `DocumentRoot` on Apache webservers.

After installing your application using one of the methods above into the directory of your choosing - we'll assume you chose /cake_install - your production setup will look like this on the file system:

```
cake_install/
    bin/
    config/
    logs/
    plugins/
    resources/
    src/
    templates/
    tests/
    tmp/
    vendor/
    webroot/ (this directory is set as DocumentRoot)
    .gitignore
    .htaccess
    composer.json
    index.php
    phpunit.xml.dist
    README.md
```

Developers using Apache should set the `DocumentRoot` directive for the domain to:

```
DocumentRoot /cake_install/webroot
```

If your web server is configured correctly, you should now find your CakePHP application accessible at http://www.example.com.

Fire It Up

Alright, let's see CakePHP in action. Depending on which setup you used, you should point your browser to http://example.com/ or http://localhost:8765/. At this point, you'll be presented with CakePHP's default home, and a message that tells you the status of your current database connection.

Congratulations! You are ready to *create your first CakePHP application*.

URL Rewriting

Apache

While CakePHP is built to work with mod_rewrite out of the box–and usually does–we've noticed that a few users struggle with getting everything to play nicely on their systems.

Here are a few things you might try to get it running correctly. First look at your httpd.conf. (Make sure you are editing the system httpd.conf rather than a user- or site-specific httpd.conf.)

These files can vary between different distributions and Apache versions. You may also take a look at https://cwiki.apache.org/confluence/display/httpd/DistrosDefaultLayout for further information.

1. Make sure that an .htaccess override is allowed and that AllowOverride is set to All for the correct DocumentRoot. You should see something similar to:

```
# Each directory to which Apache has access can be configured with respect
# to which services and features are allowed and/or disabled in that
# directory (and its subdirectories).
#
# First, we configure the "default" to be a very restrictive set of
# features.
<Directory />
    Options FollowSymLinks
    AllowOverride All
#    Order deny,allow
#    Deny from all
</Directory>
```

2. Make sure you are loading mod_rewrite correctly. You should see something like:

```
LoadModule rewrite_module libexec/apache2/mod_rewrite.so
```

   In many systems these will be commented out by default, so you may just need to remove the leading # symbols.

   After you make changes, restart Apache to make sure the settings are active.

   Verify that your .htaccess files are actually in the right directories. Some operating systems treat files that start with '.' as hidden and therefore won't copy them.

3. Make sure your copy of CakePHP comes from the downloads section of the site or our Git repository, and has been unpacked correctly, by checking for .htaccess files.

   CakePHP app directory (will be copied to the top directory of your application by bake):

```
<IfModule mod_rewrite.c>
   RewriteEngine on
   RewriteRule    ^$    webroot/    [L]
   RewriteRule    (.*) webroot/$1    [L]
</IfModule>
```

   CakePHP webroot directory (will be copied to your application's web root by bake):

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteCond %{REQUEST_FILENAME} !-f
```

(continues on next page)

(continued from previous page)

```
    RewriteRule ^ index.php [L]
</IfModule>
```

If your CakePHP site still has problems with mod_rewrite, you might want to try modifying settings for Virtual Hosts. On Ubuntu, edit the file **/etc/apache2/sites-available/default** (location is distribution-dependent). In this file, ensure that `AllowOverride None` is changed to `AllowOverride All`, so you have:

```
<Directory />
    Options FollowSymLinks
    AllowOverride All
</Directory>
<Directory /var/www>
    Options FollowSymLinks
    AllowOverride All
    Order Allow,Deny
    Allow from all
</Directory>
```

On macOS, another solution is to use the tool virtualhostx[90] to make a Virtual Host to point to your folder.

For many hosting services (GoDaddy, 1and1), your web server is being served from a user directory that already uses mod_rewrite. If you are installing CakePHP into a user directory (http://example.com/~username/cakephp/), or any other URL structure that already utilizes mod_rewrite, you'll need to add RewriteBase statements to the .htaccess files CakePHP uses (.htaccess, webroot/.htaccess).

This can be added to the same section with the RewriteEngine directive, so for example, your webroot .htaccess file would look like:

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteBase /path/to/app
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^ index.php [L]
</IfModule>
```

The details of those changes will depend on your setup, and can include additional things that are not related to CakePHP. Please refer to Apache's online documentation for more information.

4. (Optional) To improve production setup, you should prevent invalid assets from being parsed by CakePHP. Modify your webroot .htaccess to something like:

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteBase /path/to/app/
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteCond %{REQUEST_URI} !^/(webroot/)?(img|css|js)/(.*)$
    RewriteRule ^ index.php [L]
</IfModule>
```

The above will prevent incorrect assets from being sent to index.php and instead display your web server's 404 page.

Additionally you can create a matching HTML 404 page, or use the default built-in CakePHP 404 by adding an `ErrorDocument` directive:

[90] https://clickontyler.com/virtualhostx/

```
ErrorDocument 404 /404-not-found
```

nginx

nginx does not make use of .htaccess files like Apache, so it is necessary to create those rewritten URLs in the site-available configuration. This is usually found in `/etc/nginx/sites-available/your_virtual_host_conf_file`. Depending on your setup, you will have to modify this, but at the very least, you will need PHP running as a FastCGI instance. The following configuration redirects the request to `webroot/index.php`:

```
location / {
    try_files $uri $uri/ /index.php?$args;
}
```

A sample of the server directive is as follows:

```
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;
        fastcgi_intercept_errors on;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
    }
}
```

Note: Recent configurations of PHP-FPM are set to listen to the unix php-fpm socket instead of TCP port 9000 on address 127.0.0.1. If you get 502 bad gateway errors from the above configuration, try update `fastcgi_pass` to use the unix socket path (eg: fastcgi_pass unix:/var/run/php/php7.1-fpm.sock;) instead of the TCP port.

NGINX Unit

NGINX Unit[91] is dynamically configurable in runtime; the following configuration relies on `webroot/index.php`, also serving other `.php` scripts if present via `cakephp_direct`:

```
{
    "listeners": {
        "*:80": {
            "pass": "routes/cakephp"
        }
    },

    "routes": {
        "cakephp": [
            {
                "match": {
                    "uri": [
                        "*.php",
                        "*.php/*"
                    ]
                },

                "action": {
                    "pass": "applications/cakephp_direct"
                }
            },
            {
                "action": {
                    "share": "/path/to/cakephp/webroot/",
                    "fallback": {
                        "pass": "applications/cakephp_index"
                    }
                }
            }
        ]
    },

    "applications": {
        "cakephp_direct": {
            "type": "php",
            "root": "/path/to/cakephp/webroot/",
            "user": "www-data"
        },

        "cakephp_index": {
            "type": "php",
            "root": "/path/to/cakephp/webroot/",
            "user": "www-data",
            "script": "index.php"
        }
    }
}
```

[91] https://unit.nginx.org

To enable this config (assuming it's saved as `cakephp.json`):

```
# curl -X PUT --data-binary @cakephp.json --unix-socket \
       /path/to/control.unit.sock http://localhost/config
```

IIS7 (Windows hosts)

IIS7 does not natively support .htaccess files. While there are add-ons that can add this support, you can also import htaccess rules into IIS to use CakePHP's native rewrites. To do this, follow these steps:

1. Use Microsoft's Web Platform Installer[92] to install the URL Rewrite Module 2.0[93] or download it directly (32-bit[94] / 64-bit[95]).
2. Create a new file called web.config in your CakePHP root folder.
3. Using Notepad or any XML-safe editor, copy the following code into your new web.config file:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <system.webServer>
        <rewrite>
            <rules>
                <rule name="Exclude direct access to webroot/*"
                  stopProcessing="true">
                    <match url="^webroot/(.*)$" ignoreCase="false" />
                    <action type="None" />
                </rule>
                <rule name="Rewrite routed access to assets(img, css, files, js, favicon)
↪"
                  stopProcessing="true">
                    <match url="^(font|img|css|files|js|favicon.ico)(.*)$" />
                    <action type="Rewrite" url="webroot/{R:1}{R:2}"
                      appendQueryString="false" />
                </rule>
                <rule name="Rewrite requested file/folder to index.php"
                  stopProcessing="true">
                    <match url="^(.*)$" ignoreCase="false" />
                    <action type="Rewrite" url="index.php"
                      appendQueryString="true" />
                </rule>
            </rules>
        </rewrite>
    </system.webServer>
</configuration>
```

Once the web.config file is created with the correct IIS-friendly rewrite rules, CakePHP's links, CSS, JavaScript, and rerouting should work correctly.

[92] https://www.microsoft.com/web/downloads/platform.aspx
[93] https://www.iis.net/downloads/microsoft/url-rewrite
[94] https://download.microsoft.com/download/D/8/1/D81E5DD6-1ABB-46B0-9B4B-21894E18B77F/rewrite_x86_en-US.msi
[95] https://download.microsoft.com/download/1/2/8/128E2E22-C1B9-44A4-BE2A-5859ED1D4592/rewrite_amd64_en-US.msi

Lighttpd

Lighttpd does not make use of **.htaccess** files like Apache, so it is necessary to add a `url.rewrite-once` configuration in **conf/lighttpd.conf**. Ensure the following is present in your lighthttpd configuration:

```
server.modules += (
    "mod_alias",
    "mod_cgi",
    "mod_rewrite"
)

# Directory Alias
alias.url       = ( "/TestCake" => "C:/Users/Nicola/Documents/TestCake" )

# CGI Php
cgi.assign      = ( ".php" => "c:/php/php-cgi.exe" )

# Rewrite Cake Php (on /TestCake path)
url.rewrite-once = (
    "^/TestCake/(css|files|img|js|stats)/(.*)$" => "/TestCake/webroot/$1/$2",
    "^/TestCake/(.*)$" => "/TestCake/webroot/index.php/$1"
)
```

The above lines include PHP CGI configuration and example application configuration for an application on the `/TestCake` path.

I Can't Use URL Rewriting

If you don't want or can't get mod_rewrite (or some other compatible module) running on your server, you will need to use CakePHP's built in pretty URLs. In **config/app.php**, uncomment the line that looks like:

```
'App' => [
    // ...
    // 'baseUrl' => env('SCRIPT_NAME'),
]
```

Also remove these .htaccess files:

```
/.htaccess
webroot/.htaccess
```

This will make your URLs look like www.example.com/index.php/controllername/actionname/param rather than www.example.com/controllername/actionname/param.

CHAPTER 7

Configuration

While conventions remove the need to configure all of CakePHP, you'll still need to configure a few things like your database credentials.

Additionally, there are optional configuration options that allow you to swap out default values & implementations with ones tailored to your application.

Configuring your Application

Configuration is generally stored in either PHP or INI files, and loaded during the application bootstrap. CakePHP comes with one configuration file by default, but if required you can add additional configuration files and load them in your application's bootstrap code. *Cake\Core\Configure* is used for global configuration, and classes like `Cache` provide `setConfig()` methods to make configuration simple and transparent.

The application skeleton features a **config/app.php** file which should contain configuration that doesn't vary across the various environments your application is deployed in. The **config/app_local.php** file should contain the configuration data that varies between environments and should be managed by configuration management, or your deployment tooling. Both of these files reference environment variables through the `env()` function that enables configuration values to set though the server environment.

Loading Additional Configuration Files

If your application has many configuration options it can be helpful to split configuration into multiple files. After creating each of the files in your **config/** directory you can load them in **bootstrap.php**:

```
use Cake\Core\Configure;
use Cake\Core\Configure\Engine\PhpConfig;

Configure::setConfig('default', new PhpConfig());
Configure::load('app', 'default', false);
Configure::load('other_config', 'default');
```

Environment Variables

Many modern cloud providers, like Heroku, let you define environment variables for configuration data. You can configure your CakePHP through environment variables in the 12factor app style[96]. Environment variables allow your application to require less state making your application easier to manage when it is deployed across a number of environments.

As you can see in your **app.php**, the `env()` function is used to read configuration from the environment, and build the application configuration. CakePHP uses *DSN* strings for databases, logs, email transports and cache configurations allowing you to easily vary these libraries in each environment.

For local development, CakePHP leverages dotenv[97] to make local development automatically reload environment variables. Use composer to require this library and then there is a block of code in `bootstrap.php` that needs to be uncommented to harness it.

You will see a `config/.env.example` in your application. By copying this file into `config/.env` and customizing the values you can configure your application.

You should avoid committing the `config/.env` file to your repository and instead use the `config/.env.example` as a template with placeholder values so everyone on your team knows what environment variables are in use and what should go in each one.

Once your environment variables have been set, you can use `env()` to read data from the environment:

```
$debug = env('APP_DEBUG', false);
```

The second value passed to the env function is the default value. This value will be used if no environment variable exists for the given key.

General Configuration

Below is a description of the variables and how they affect your CakePHP application.

debug
: Changes CakePHP debugging output. `false` = Production mode. No error messages, errors, or warnings shown. `true` = Errors and warnings shown.

App.namespace
: The namespace to find app classes under.

[96] https://12factor.net/
[97] https://github.com/josegonzalez/php-dotenv

Note: When changing the namespace in your configuration, you will also need to update your **composer.json** file to use this namespace as well. Additionally, create a new autoloader by running `php composer.phar dumpautoload`.

App.baseUrl
Un-comment this definition if you **don't** plan to use Apache's mod_rewrite with CakePHP. Don't forget to remove your .htaccess files too.

App.base
The base directory the app resides in. If `false` this will be auto detected. If not `false`, ensure your string starts with a `/` and does NOT end with a `/`. For example, */basedir* is a valid App.base.

App.encoding
Define what encoding your application uses. This encoding is used to generate the charset in the layout, and encode entities. It should match the encoding values specified for your database.

App.webroot
The webroot directory.

App.wwwRoot
The file path to webroot.

App.fullBaseUrl
The fully qualified domain name (including protocol) to your application's root. This is used when generating absolute URLs. By default this value is generated using the `$_SERVER` environment. However, you should define it manually to optimize performance or if you are concerned about people manipulating the `Host` header. In a CLI context (from command) the *fullBaseUrl* cannot be read from $_SERVER, as there is no webserver involved. You do need to specify it yourself if you do need to generate URLs from a shell (for example, when sending emails).

App.imageBaseUrl
Web path to the public images directory under webroot. If you are using a *CDN* you should set this value to the CDN's location.

App.cssBaseUrl
Web path to the public css directory under webroot. If you are using a *CDN* you should set this value to the CDN's location.

App.jsBaseUrl
Web path to the public js directory under webroot. If you are using a *CDN* you should set this value to the CDN's location.

App.paths
Configure paths for non class based resources. Supports the `plugins`, `templates`, `locales` subkeys, which allow the definition of paths for plugins, view templates and locale files respectively.

App.uploadedFilesAsObjects
Defines whether uploaded files are being represented as objects (`true`), or arrays (`false`). This option is being treated as enabled by default. See the *File Uploads section* in the Request & Response Objects chapter for more information.

Security.salt
A random string used in hashing. This value is also used as the HMAC salt when doing symmetric encryption.

Asset.timestamp
Appends a timestamp which is last modified time of the particular file at the end of asset files URLs (CSS, JavaScript, Image) when using proper helpers. Valid values:

- (bool) `false` - Doesn't do anything (default)

- (bool) `true` - Appends the timestamp when debug is `true`
- (string) 'force' - Always appends the timestamp.

Asset.cacheTime
Sets the asset cache time. This determines the http header `Cache-Control`'s `max-age`, and the http header's `Expire`'s time for assets. This can take anything that you version of PHP's strtotime function[98] can take. The default is `+1 day`.

Using a CDN

To use a CDN for loading your static assets, change `App.imageBaseUrl`, `App.cssBaseUrl`, `App.jsBaseUrl` to point the CDN URI, for example: `https://mycdn.example.com/` (note the trailing `/`).

All images, scripts and styles loaded via HtmlHelper will prepend the absolute CDN path, matching the same relative path used in the application. Please note there is a specific use case when using plugin based assets: plugins will not use the plugin's prefix when absolute `...BaseUrl` URI is used, for example By default:

- `$this->Helper->assetUrl('TestPlugin.logo.png')` resolves to `test_plugin/logo.png`

If you set `App.imageBaseUrl` to `https://mycdn.example.com/`:

- `$this->Helper->assetUrl('TestPlugin.logo.png')` resolves to `https://mycdn.example.com/logo.png`.

Database Configuration

See the *Database Configuration* for information on configuring your database connections.

Caching Configuration

See the *Caching Configuration* for information on configuring caching in CakePHP.

Error and Exception Handling Configuration

See the *Error and Exception Configuration* for information on configuring error and exception handlers.

Logging Configuration

See the *Logging Configuration* for information on configuring logging in CakePHP.

[98] https://php.net/manual/en/function.strtotime.php

Email Configuration

See the *Email Configuration* for information on configuring email presets in CakePHP.

Session Configuration

See the *Session Configuration* for information on configuring session handling in CakePHP.

Routing configuration

See the *Routes Configuration* for more information on configuring routing and creating routes for your application.

Additional Class Paths

Additional class paths are setup through the autoloaders your application uses. When using `composer` to generate your autoloader, you could do the following, to provide fallback paths for controllers in your application:

```
"autoload": {
    "psr-4": {
        "App\\Controller\\": "/path/to/directory/with/controller/folders/",
        "App\\": "src/"
    }
}
```

The above would setup paths for both the `App` and `App\Controller` namespace. The first key will be searched, and if that path does not contain the class/file the second key will be searched. You can also map a single namespace to multiple directories with the following:

```
"autoload": {
    "psr-4": {
        "App\\": ["src/", "/path/to/directory/"]
    }
}
```

Plugin, View Template and Locale Paths

Since plugins, view templates and locales are not classes, they cannot have an autoloader configured. CakePHP provides three Configure variables to setup additional paths for these resources. In your **config/app.php** you can set these variables:

```
return [
    // More configuration
    'App' => [
        'paths' => [
            'plugins' => [
                ROOT . DS . 'plugins' . DS,
                '/path/to/other/plugins/',
            ],
            'templates' => [
```

(continues on next page)

(continued from previous page)

```
                ROOT . DS . 'templates' . DS,
                ROOT . DS . 'templates2' . DS,
            ],
            'locales' => [
                ROOT . DS . 'resources' . DS . 'locales' . DS,
            ],
        ],
    ],
];
```

Paths should end with a directory separator, or they will not work properly.

Inflection Configuration

See the *Inflection Configuration* docs for more information.

Configure Class

class Cake\Core**Configure**

CakePHP's Configure class can be used to store and retrieve application or runtime specific values. Be careful, this class allows you to store anything in it, then use it in any other part of your code: a sure temptation to break the MVC pattern CakePHP was designed for. The main goal of Configure class is to keep centralized variables that can be shared between many objects. Remember to try to live by "convention over configuration" and you won't end up breaking the MVC structure CakePHP provides.

Writing Configuration data

static Cake\Core\Configure::**write**(*$key*, *$value*)

Use `write()` to store data in the application's configuration:

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

Note: The *dot notation* used in the `$key` parameter can be used to organize your configuration settings into logical groups.

The above example could also be written in a single call:

```
Configure::write('Company', [
    'name' => 'Pizza, Inc.',
    'slogan' => 'Pizza for your body and soul'
]);
```

You can use `Configure::write('debug', $bool)` to switch between debug and production modes on the fly.

Note: Any configuration changes done using `Configure::write()` are in memory and will not persist across requests.

Reading Configuration Data

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

Used to read configuration data from the application. If a key is supplied, the data is returned. Using our examples from write() above, we can read that data back:

```
// Returns 'Pizza Inc.'
Configure::read('Company.name');

// Returns 'Pizza for your body and soul'
Configure::read('Company.slogan');

Configure::read('Company');
// Returns:
['name' => 'Pizza, Inc.', 'slogan' => 'Pizza for your body and soul'];

// Returns 'fallback' as Company.nope is undefined.
Configure::read('Company.nope', 'fallback');
```

If `$key` is left null, all values in Configure will be returned.

static Cake\Core\Configure::**readOrFail**(*$key*)

Reads configuration data just like *Cake\Core\Configure::read* but expects to find a key/value pair. In case the requested pair does not exist, a `RuntimeException` will be thrown:

```
Configure::readOrFail('Company.name');    // Yields: 'Pizza, Inc.'
Configure::readOrFail('Company.geolocation');  // Will throw an exception

Configure::readOrFail('Company');

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

Checking to see if Configuration Data is Defined

static Cake\Core\Configure::**check**(*$key*)

Used to check if a key/path exists and has non-null value:

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

Deleting Configuration Data

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

Used to delete information from the application's configuration:

```
Configure::delete('Company.name');
```

Reading & Deleting Configuration Data

static Cake\Core\Configure::**consume**(*$key*)

Read and delete a key from Configure. This is useful when you want to combine reading and deleting values in a single operation.

static Cake\Core\Configure::**consumeOrFail**(*$key*)

Consumes configuration data just like *Cake\Core\Configure::consume* but expects to find a key/value pair. In case the requested pair does not exist, a `RuntimeException` will be thrown:

```
Configure::consumeOrFail('Company.name');    // Yields: 'Pizza, Inc.'
Configure::consumeOrFail('Company.geolocation');  // Will throw an exception

Configure::consumeOrFail('Company');

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

Reading and writing configuration files

static Cake\Core\Configure::**setConfig**(*$name*, *$engine*)

CakePHP comes with two built-in configuration file engines. `Cake\Core\Configure\Engine\PhpConfig` is able to read PHP config files, in the same format that Configure has historically read. `Cake\Core\Configure\Engine\IniConfig` is able to read ini config files. See the PHP documentation[99] for more information on the specifics of ini files. To use a core config engine, you'll need to attach it to Configure using `Configure::config()`:

```
use Cake\Core\Configure\Engine\PhpConfig;

// Read config files from config
Configure::config('default', new PhpConfig());

// Read config files from another path.
Configure::config('default', new PhpConfig('/path/to/your/config/files/'));
```

You can have multiple engines attached to Configure, each reading different kinds or sources of configuration files. You can interact with attached engines using a few other methods on Configure. To check which engine aliases are attached you can use `Configure::configured()`:

[99] https://php.net/parse_ini_file

```
// Get the array of aliases for attached engines.
Configure::configured();

// Check if a specific engine is attached
Configure::configured('default');
```

static Cake\Core\Configure::**drop**(*$name*)

You can also remove attached engines. `Configure::drop('default')` would remove the default engine alias. Any future attempts to load configuration files with that engine would fail:

```
Configure::drop('default');
```

Loading Configuration Files

static Cake\Core\Configure::**load**(*$key*, *$config = 'default'*, *$merge = true*)

Once you've attached a config engine to Configure you can load configuration files:

```
// Load my_file.php using the 'default' engine object.
Configure::load('my_file', 'default');
```

Loaded configuration files merge their data with the existing runtime configuration in Configure. This allows you to overwrite and add new values into the existing runtime configuration. By setting `$merge` to `true`, values will not ever overwrite the existing configuration.

Warning: When merging configuration files with *$merge = true*, dot notation in keys is not expanded:

```
// config1.php
'Key1' => [
    'Key2' => [
        'Key3' => ['NestedKey1' => 'Value'],
    ],
],

// config2.php
'Key1.Key2' => [
    'Key3' => ['NestedKey2' => 'Value2'],
]

Configure::load('config1', 'default');
Configure::load('config2', 'default', true);

// Now Key1.Key2.Key3 has the value ['NestedKey2' => 'Value2']
// instead of ['NestedKey1' => 'Value', 'NestedKey2' => 'Value2']
```

Creating or Modifying Configuration Files

static Cake\Core\Configure::**dump**(*$key*, *$config = 'default'*, *$keys = []*)

Dumps all or some of the data in Configure into a file or storage system supported by a config engine. The serialization format is decided by the config engine attached as $config. For example, if the 'default' engine is a `Cake\Core\Configure\Engine\PhpConfig`, the generated file will be a PHP configuration file loadable by the `Cake\Core\Configure\Engine\PhpConfig`

Given that the 'default' engine is an instance of PhpConfig. Save all data in Configure to the file *my_config.php*:

```
Configure::dump('my_config', 'default');
```

Save only the error handling configuration:

```
Configure::dump('error', 'default', ['Error', 'Exception']);
```

`Configure::dump()` can be used to either modify or overwrite configuration files that are readable with *Configure::load()*

Storing Runtime Configuration

static Cake\Core\Configure::**store**(*$name*, *$cacheConfig = 'default'*, *$data = null*)

You can also store runtime configuration values for use in a future request. Since configure only remembers values for the current request, you will need to store any modified configuration information if you want to use it in subsequent requests:

```
// Store the current configuration in the 'user_1234' key in the 'default' cache.
Configure::store('user_1234', 'default');
```

Stored configuration data is persisted in the named cache configuration. See the *Caching* documentation for more information on caching.

Restoring Runtime Configuration

static Cake\Core\Configure::**restore**(*$name*, *$cacheConfig = 'default'*)

Once you've stored runtime configuration, you'll probably need to restore it so you can access it again. `Configure::restore()` does exactly that:

```
// Restore runtime configuration from the cache.
Configure::restore('user_1234', 'default');
```

When restoring configuration information it's important to restore it with the same key, and cache configuration as was used to store it. Restored information is merged on top of the existing runtime configuration.

Configuration Engines

CakePHP provides the ability to load configuration files from a number of different sources, and features a pluggable system for creating your own configuration engines[100]. The built in configuration engines are:

- JsonConfig[101]
- IniConfig[102]
- PhpConfig[103]

By default your application will use `PhpConfig`.

Disabling Generic Tables

While utilizing generic table classes - also called auto-tables - when quickly creating new applications and baking models is useful, generic table class can make debugging more difficult in some scenarios.

You can check if any query was emitted from a generic table class via DebugKit via the SQL panel in DebugKit. If you're still having trouble diagnosing an issue that could be caused by auto-tables, you can throw an exception when CakePHP implicitly uses a generic `Cake\ORM\Table` instead of your concrete class like so:

```
// In your bootstrap.php
use Cake\Event\EventManager;
use Cake\Http\Exception\InternalErrorException;

$isCakeBakeShellRunning = (PHP_SAPI === 'cli' && isset($argv[1]) && $argv[1] === 'bake');
if (!$isCakeBakeShellRunning) {
    EventManager::instance()->on('Model.initialize', function($event) {
        $subject = $event->getSubject();
        if (get_class($subject) === 'Cake\ORM\Table') {
            $msg = sprintf(
                'Missing table class or incorrect alias when registering table class for␣
↪database table %s.',
                $subject->getTable());
            throw new InternalErrorException($msg);
        }
    });
}
```

[100] https://api.cakephp.org/5.x/interface-Cake.Core.Configure.ConfigEngineInterface.html
[101] https://api.cakephp.org/5.x/class-Cake.Core.Configure.Engine.JsonConfig.html
[102] https://api.cakephp.org/5.x/class-Cake.Core.Configure.Engine.IniConfig.html
[103] https://api.cakephp.org/5.x/class-Cake.Core.Configure.Engine.PhpConfig.html

CHAPTER 8

Routing

class Cake\Routing**RouterBuilder**

Routing provides you tools that map URLs to controller actions. By defining routes, you can separate how your application is implemented from how its URLs are structured.

Routing in CakePHP also encompasses the idea of reverse routing, where an array of parameters can be transformed into a URL string. By using reverse routing, you can re-factor your application's URL structure without having to update all your code.

Quick Tour

This section will teach you by example the most common uses of the CakePHP Router. Typically you want to display something as a landing page, so you add this to your **config/routes.php** file:

```
/** @var \Cake\Routing\RouteBuilder $routes */
$routes->connect('/', ['controller' => 'Articles', 'action' => 'index']);
```

This will execute the index method in the `ArticlesController` when the homepage of your site is visited. Sometimes you need dynamic routes that will accept multiple parameters, this would be the case, for example of a route for viewing an article's content:

```
$routes->connect('/articles/*', ['controller' => 'Articles', 'action' => 'view']);
```

The above route will accept any URL looking like `/articles/15` and invoke the method `view(15)` in the `ArticlesController`. This will not, though, prevent people from trying to access URLs looking like `/articles/foobar`. If you wish, you can restrict some parameters to conform to a regular expression:

```
// Using fluent interface
$routes->connect(
```

(continues on next page)

(continued from previous page)

```
    '/articles/{id}',
    ['controller' => 'Articles', 'action' => 'view'],
)
->setPatterns(['id' => '\d+'])
->setPass(['id']);

// Using options array
$routes->connect(
    '/articles/{id}',
    ['controller' => 'Articles', 'action' => 'view'],
    ['id' => '\d+', 'pass' => ['id']]
);
```

The previous example changed the star matcher by a new placeholder `{id}`. Using placeholders allows us to validate parts of the URL, in this case we used the `\d+` regular expression so that only digits are matched. Finally, we told the Router to treat the `id` placeholder as a function argument to the `view()` function by specifying the `pass` option. More on using this option later.

The CakePHP Router can also reverse match routes. That means that from an array containing matching parameters, it is capable of generating a URL string:

```
use Cake\Routing\Router;

echo Router::url(['controller' => 'Articles', 'action' => 'view', 'id' => 15]);
// Will output
/articles/15
```

Routes can also be labelled with a unique name, this allows you to quickly reference them when building links instead of specifying each of the routing parameters:

```
// In routes.php
$routes->connect(
    '/upgrade',
    ['controller' => 'Subscriptions', 'action' => 'create'],
    ['_name' => 'upgrade']
);

use Cake\Routing\Router;

echo Router::url(['_name' => 'upgrade']);
// Will output
/upgrade
```

To help keep your routing code DRY, the Router has the concept of 'scopes'. A scope defines a common path segment, and optionally route defaults. Any routes connected inside a scope will inherit the path/defaults from their wrapping scopes:

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

The above route would match `/blog/` and send it to `Blog\Controller\ArticlesController::index()`.

The application skeleton comes with a few routes to get you started. Once you've added your own routes, you can

remove the default routes if you don't need them.

Connecting Routes

To keep your code *DRY* you should use 'routing scopes'. Routing scopes not only let you keep your code DRY, they also help Router optimize its operation. This method defaults to the / scope. To create a scope and connect some routes we'll use the `scope()` method:

```
// In config/routes.php
use Cake\Routing\RouteBuilder;
use Cake\Routing\Route\DashedRoute;

$routes->scope('/', function (RouteBuilder $routes) {
    // Connect the generic fallback routes.
    $routes->fallbacks(DashedRoute::class);
});
```

The `connect()` method takes up to three parameters: the URL template you wish to match, the default values for your route elements, and the options for the route. Options frequently include regular expression rules to help the router match elements in the URL.

The basic format for a route definition is:

```
$routes->connect(
    '/url/template',
    ['targetKey' => 'targetValue'],
    ['option' => 'matchingRegex']
);
```

The first parameter is used to tell the router what sort of URL you're trying to control. The URL is a normal slash delimited string, but can also contain a wildcard (*) or *Route Elements*. Using a wildcard tells the router that you are willing to accept any additional arguments supplied. Routes without a * only match the exact template pattern supplied.

Once you've specified a URL, you use the last two parameters of `connect()` to tell CakePHP what to do with a request once it has been matched. The second parameter defines the route 'target'. This can be defined either as an array, or as a destination string. A few examples of route targets are:

```
// Array target to an application controller
$routes->connect(
    '/users/view/*',
    ['controller' => 'Users', 'action' => 'view']
);
$routes->connect('/users/view/*', 'Users::view');

// Array target to a prefixed plugin controller
$routes->connect(
    '/admin/cms/articles',
    ['prefix' => 'Admin', 'plugin' => 'Cms', 'controller' => 'Articles', 'action' =>
↪'index']
);
$routes->connect('/admin/cms/articles', 'Cms.Admin/Articles::index');
```

The first route we connect matches URLs starting with `/users/view` and maps those requests to the `UsersController->view()`. The trailing `/*` tells the router to pass any additional segments as method arguments.

For example, `/users/view/123` would map to `UsersController->view(123)`.

The above example also illustrates string targets. String targets provide a compact way to define a route's destination. String targets have the following syntax:

```
[Plugin].[Prefix]/[Controller]::[action]
```

Some example string targets are:

```
// Application controller
'Articles::view'

// Application controller with prefix
Admin/Articles::view

// Plugin controller
Cms.Articles::edit

// Prefixed plugin controller
Vendor/Cms.Management/Admin/Articles::view
```

Earlier we used the greedy star (`/*`) to capture additional path segments, there is also the trailing star (`/**`). Using a trailing double star, will capture the remainder of a URL as a single passed argument. This is useful when you want to use an argument that included a `/` in it:

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

The incoming URL of `/pages/the-example-/-and-proof` would result in a single passed argument of `the-example-/-and-proof`.

The second parameter of `connect()` can define any parameters that compose the default route parameters:

```
$routes->connect(
    '/government',
    ['controller' => 'Pages', 'action' => 'display', 5]
);
```

This example uses the second parameter of `connect()` to define default parameters. If you built an application that features products for different categories of customers, you might consider creating a route. This allows you to link to `/government` rather than `/pages/display/5`.

A common use for routing is to rename controllers and their actions. Instead of accessing our users controller at `/users/some-action/5`, we'd like to be able to access it through `/cooks/some-action/5`. The following route takes care of that:

```
$routes->connect(
    '/cooks/{action}/*', ['controller' => 'Users']
);
```

This is telling the Router that any URL beginning with `/cooks/` should be sent to the `UsersController`. The action called will depend on the value of the `{action}` parameter. By using *Route Elements*, you can create variable routes, that accept user input or variables. The above route also uses the greedy star. The greedy star indicates that this route should accept any additional positional arguments given. These arguments will be made available in the *Passed Arguments* array.

When generating URLs, routes are used too. Using `['controller' => 'Users', 'action' => 'some-action', 5]` as a URL will output `/cooks/some-action/5` if the above route is the first match found.

The routes we've connected so far will match any HTTP verb. If you are building a REST API you'll often want to map HTTP actions to different controller methods. The `RouteBuilder` provides helper methods that make defining routes for specific HTTP verbs simpler:

```
// Create a route that only responds to GET requests.
$routes->get(
    '/cooks/{id}',
    ['controller' => 'Users', 'action' => 'view'],
    'users:view'
);

// Create a route that only responds to PUT requests
$routes->put(
    '/cooks/{id}',
    ['controller' => 'Users', 'action' => 'update'],
    'users:update'
);
```

The above routes map the same URL to different controller actions based on the HTTP verb used. GET requests will go to the 'view' action, while PUT requests will go to the 'update' action. There are HTTP helper methods for:

- GET
- POST
- PUT
- PATCH
- DELETE
- OPTIONS
- HEAD

All of these methods return the route instance allowing you to leverage the *fluent setters* to further configure your route.

Route Elements

You can specify your own route elements and doing so gives you the power to define places in the URL where parameters for controller actions should lie. When a request is made, the values for these route elements are found in `$this->request->getParam()` in the controller. When you define a custom route element, you can optionally specify a regular expression - this tells CakePHP how to know if the URL is correctly formed or not. If you choose to not provide a regular expression, any non / character will be treated as part of the parameter:

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

$routes->connect(
    '/{controller}/{id}',
    ['action' => 'view'],
```

(continues on next page)

(continued from previous page)

```
    ['id' => '[0-9]+']
);
```

The above example illustrates how to create a quick way to view models from any controller by crafting a URL that looks like `/controllername/{id}`. The URL provided to `connect()` specifies two route elements: `{controller}` and `{id}`. The `{controller}` element is a CakePHP default route element, so the router knows how to match and identify controller names in URLs. The `{id}` element is a custom route element, and must be further clarified by specifying a matching regular expression in the third parameter of `connect()`.

CakePHP does not automatically produce lowercased and dashed URLs when using the `{controller}` parameter. If you need this, the above example could be rewritten like so:

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

// Create a builder with a different route class.
$routes->scope('/', function (RouteBuilder $routes) {
    $routes->setRouteClass(DashedRoute::class);
    $routes->connect('/{controller}/{id}', ['action' => 'view'])
        ->setPatterns(['id' => '[0-9]+']);

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

The `DashedRoute` class will make sure that the `{controller}` and `{plugin}` parameters are correctly lowercased and dashed.

Note: Patterns used for route elements must not contain any capturing groups. If they do, Router will not function correctly.

Once this route has been defined, requesting `/apples/5` would call the `view()` method of the ApplesController. Inside the `view()` method, you would need to access the passed ID at `$this->request->getParam('id')`.

If you have a single controller in your application and you do not want the controller name to appear in the URL, you can map all URLs to actions in your controller. For example, to map all URLs to actions of the `home` controller, e.g have URLs like `/demo` instead of `/home/demo`, you can do the following:

```
$routes->connect('/{action}', ['controller' => 'Home']);
```

If you would like to provide a case insensitive URL, you can use regular expression inline modifiers:

```
$routes->connect(
    '/{userShortcut}',
    ['controller' => 'Teachers', 'action' => 'profile', 1],
)->setPatterns(['userShortcut' => '(?i:principal)']);
```

One more example, and you'll be a routing pro:

```
$routes->connect(
    '/{controller}/{year}/{month}/{day}',
```

(continues on next page)

(continued from previous page)

```
    ['action' => 'index']
)->setPatterns([
    'year' => '[12][0-9]{3}',
    'month' => '0[1-9]|1[012]',
    'day' => '0[1-9]|[12][0-9]|3[01]'
]);
```

This is rather involved, but shows how powerful routes can be. The URL supplied has four route elements. The first is familiar to us: it's a default route element that tells CakePHP to expect a controller name.

Next, we specify some default values. Regardless of the controller, we want the `index()` action to be called.

Finally, we specify some regular expressions that will match years, months and days in numerical form. Note that parenthesis (capturing groups) are not supported in the regular expressions. You can still specify alternates, as above, but not grouped with parenthesis.

Once defined, this route will match `/articles/2007/02/01`, `/articles/2004/11/16`, handing the requests to the `index()` actions of their respective controllers, with the date parameters in `$this->request->getParam()`.

Reserved Route Elements

There are several route elements that have special meaning in CakePHP, and should not be used unless you want the special meaning

- `controller` Used to name the controller for a route.
- `action` Used to name the controller action for a route.
- `plugin` Used to name the plugin a controller is located in.
- `prefix` Used for *Prefix Routing*
- `_ext` Used for *File extentions routing*.
- `_base` Set to `false` to remove the base path from the generated URL. If your application is not in the root directory, this can be used to generate URLs that are 'cake relative'.
- `_scheme` Set to create links on different schemes like *webcal* or *ftp*. Defaults to the current scheme.
- `_host` Set the host to use for the link. Defaults to the current host.
- `_port` Set the port if you need to create links on non-standard ports.
- `_full` If `true` the value of `App.fullBaseUrl` mentioned in *General Configuration* will be prepended to generated URLs.
- `#` Allows you to set URL hash fragments.
- `_https` Set to `true` to convert the generated URL to https or `false` to force http. Prior to 4.5.0 use `_ssl`.
- `_method` Define the HTTP verb/method to use. Useful when working with *RESTful Routing*.
- `_name` Name of route. If you have setup named routes, you can use this key to specify it.

Configuring Route Options

There are a number of route options that can be set on each route. After connecting a route you can use its fluent builder methods to further configure the route. These methods replace many of the keys in the `$options` parameter of `connect()`:

```
$routes->connect(
    '/{lang}/articles/{slug}',
    ['controller' => 'Articles', 'action' => 'view'],
)
// Allow GET and POST requests.
->setMethods(['GET', 'POST'])

// Only match on the blog subdomain.
->setHost('blog.example.com')

// Set the route elements that should be converted to passed arguments
->setPass(['slug'])

// Set the matching patterns for route elements
->setPatterns([
    'slug' => '[a-z0-9-_]+',
    'lang' => 'en|fr|es',
])

// Also allow JSON file extensions
->setExtensions(['json'])

// Set lang to be a persistent parameter
->setPersist(['lang']);
```

Passing Parameters to Action

When connecting routes using *Route Elements* you may want to have routed elements be passed arguments instead. The `pass` option indicates which route elements should also be made available as arguments passed into the controller functions:

```
// src/Controller/BlogsController.php
public function view($articleId = null, $slug = null)
{
    // Some code here...
}

// routes.php
$routes->scope('/', function (RouteBuilder $routes) {
    $routes->connect(
        '/blog/{id}-{slug}', // For example, /blog/3-CakePHP_Rocks
        ['controller' => 'Blogs', 'action' => 'view']
    )
    // Define the route elements in the route template
    // to prepend as function arguments. Order matters as this
    // will pass the `$id` and `$slug` elements as the first and
```

(continues on next page)

(continued from previous page)

```
    // second parameters. Any additional passed parameters in your
    // route will be added after the setPass() arguments.
    ->setPass(['id', 'slug'])
    // Define a pattern that `id` must match.
    ->setPatterns([
        'id' => '[0-9]+',
    ]);
});
```

Now thanks to the reverse routing capabilities, you can pass in the URL array like below and CakePHP will know how to form the URL as defined in the routes:

```
// view.php
// This will return a link to /blog/3-CakePHP_Rocks
echo $this->Html->link('CakePHP Rocks', [
    'controller' => 'Blog',
    'action' => 'view',
    'id' => 3,
    'slug' => 'CakePHP_Rocks'
]);

// You can also used numerically indexed parameters.
echo $this->Html->link('CakePHP Rocks', [
    'controller' => 'Blog',
    'action' => 'view',
    3,
    'CakePHP_Rocks'
]);
```

Using Path Routing

We talked about string targets above. The same also works for URL generation using `Router::pathUrl()`:

```
echo Router::pathUrl('Articles::index');
// outputs: /articles

echo Router::pathUrl('MyBackend.Admin/Articles::view', [3]);
// outputs: /admin/my-backend/articles/view/3
```

Tip: IDE support for Path Routing autocomplete can be enabled with CakePHP IdeHelper Plugin[104].

[104] https://github.com/dereuromark/cakephp-ide-helper

Using Named Routes

Sometimes you'll find typing out all the URL parameters for a route too verbose, or you'd like to take advantage of the performance improvements that named routes have. When connecting routes you can specify a `_name` option, this option can be used in reverse routing to identify the route you want to use:

```
// Connect a route with a name.
$routes->connect(
    '/login',
    ['controller' => 'Users', 'action' => 'login'],
    ['_name' => 'login']
);

// Name a verb specific route
$routes->post(
    '/logout',
    ['controller' => 'Users', 'action' => 'logout'],
    'logout'
);

// Generate a URL using a named route.
$url = Router::url(['_name' => 'logout']);

// Generate a URL using a named route,
// with some query string args.
$url = Router::url(['_name' => 'login', 'username' => 'jimmy']);
```

If your route template contains any route elements like `{controller}` you'll need to supply those as part of the options to `Router::url()`.

Note: Route names must be unique across your entire application. The same `_name` cannot be used twice, even if the names occur inside a different routing scope.

When building named routes, you will probably want to stick to some conventions for the route names. CakePHP makes building up route names easier by allowing you to define name prefixes in each scope:

```
$routes->scope('/api', ['_namePrefix' => 'api:'], function (RouteBuilder $routes) {
    // This route's name will be `api:ping`
    $routes->get('/ping', ['controller' => 'Pings'], 'ping');
});
// Generate a URL for the ping route
Router::url(['_name' => 'api:ping']);

// Use namePrefix with plugin()
$routes->plugin('Contacts', ['_namePrefix' => 'contacts:'], function (RouteBuilder
↪$routes) {
    // Connect routes.
});

// Or with prefix()
$routes->prefix('Admin', ['_namePrefix' => 'admin:'], function (RouteBuilder $routes) {
    // Connect routes.
});
```

You can also use the `_namePrefix` option inside nested scopes and it works as you'd expect:

```
$routes->plugin('Contacts', ['_namePrefix' => 'contacts:'], function (RouteBuilder
↪$routes) {
    $routes->scope('/api', ['_namePrefix' => 'api:'], function (RouteBuilder $routes) {
        // This route's name will be `contacts:api:ping`
        $routes->get('/ping', ['controller' => 'Pings'], 'ping');
    });
});

// Generate a URL for the ping route
Router::url(['_name' => 'contacts:api:ping']);
```

Routes connected in named scopes will only have names added if the route is also named. Nameless routes will not have the `_namePrefix` applied to them.

Prefix Routing

static Cake\Routing\RouterBuilder::**prefix**(*$name*, *$callback*)

Many applications require an administration section where privileged users can make changes. This is often done through a special URL such as `/admin/users/edit/5`. In CakePHP, prefix routing can be enabled by using the `prefix` scope method:

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

$routes->prefix('Admin', function (RouteBuilder $routes) {
    // All routes here will be prefixed with `/admin`, and
    // have the `'prefix' => 'Admin'` route element added that
    // will be required when generating URLs for these routes
    $routes->fallbacks(DashedRoute::class);
});
```

Prefixes are mapped to sub-namespaces in your application's `Controller` namespace. By having prefixes as separate controllers you can create smaller and simpler controllers. Behavior that is common to the prefixed and non-prefixed controllers can be encapsulated using inheritance, *Components*, or traits. Using our users example, accessing the URL `/admin/users/edit/5` would call the `edit()` method of our **src/Controller/Admin/UsersController.php** passing 5 as the first parameter. The view file used would be **templates/Admin/Users/edit.php**

You can map the URL /admin to your `index()` action of pages controller using following route:

```
$routes->prefix('Admin', function (RouteBuilder $routes) {
    // Because you are in the admin scope,
    // you do not need to include the /admin prefix
    // or the Admin route element.
    $routes->connect('/', ['controller' => 'Pages', 'action' => 'index']);
});
```

When creating prefix routes, you can set additional route parameters using the `$options` argument:

```
$routes->prefix('Admin', ['param' => 'value'], function (RouteBuilder $routes) {
    // Routes connected here are prefixed with '/admin' and
    // have the 'param' routing key set.
```

(continues on next page)

(continued from previous page)

```
    $routes->connect('/{controller}');
});
```

Multi word prefixes are by default converted using dasherize inflection, ie `MyPrefix` would be mapped to `my-prefix` in the URL. Make sure to set a path for such prefixes if you want to use a different format like for example underscoring:

```
$routes->prefix('MyPrefix', ['path' => '/my_prefix'], function (RouteBuilder $routes) {
    // Routes connected here are prefixed with '/my_prefix'
    $routes->connect('/{controller}');
});
```

You can define prefixes inside plugin scopes as well:

```
$routes->plugin('DebugKit', function (RouteBuilder $routes) {
    $routes->prefix('Admin', function (RouteBuilder $routes) {
        $routes->connect('/{controller}');
    });
});
```

The above would create a route template like `/debug-kit/admin/{controller}`. The connected route would have the `plugin` and `prefix` route elements set.

When defining prefixes, you can nest multiple prefixes if necessary:

```
$routes->prefix('Manager', function (RouteBuilder $routes) {
    $routes->prefix('Admin', function (RouteBuilder $routes) {
        $routes->connect('/{controller}/{action}');
    });
});
```

The above would create a route template like `/manager/admin/{controller}/{action}`. The connected route would have the `prefix` route element set to `Manager/Admin`.

The current prefix will be available from the controller methods through `$this->request->getParam('prefix')`

When using prefix routes it's important to set the `prefix` option, and to use the same CamelCased format that is used in the `prefix()` method. Here's how to build this link using the HTML helper:

```
// Go into a prefixed route.
echo $this->Html->link(
    'Manage articles',
    ['prefix' => 'Manager/Admin', 'controller' => 'Articles', 'action' => 'add']
);

// Leave a prefix
echo $this->Html->link(
    'View Post',
    ['prefix' => false, 'controller' => 'Articles', 'action' => 'view', 5]
);
```

Creating Links to Prefix Routes

You can create links that point to a prefix, by adding the prefix key to your URL array:

```
echo $this->Html->link(
    'New admin todo',
    ['prefix' => 'Admin', 'controller' => 'TodoItems', 'action' => 'create']
);
```

When using nesting, you need to chain them together:

```
echo $this->Html->link(
    'New todo',
    ['prefix' => 'Admin/MyPrefix', 'controller' => 'TodoItems', 'action' => 'create']
);
```

This would link to a controller with the namespace `App\\Controller\\Admin\\MyPrefix` and the file path `src/Controller/Admin/MyPrefix/TodoItemsController.php`.

Note: The prefix is always CamelCased here, even if the routing result is dashed. The route itself will do the inflection if necessary.

Plugin Routing

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

Routes for *Plugins* should be created using the `plugin()` method. This method creates a new routing scope for the plugin's routes:

```
$routes->plugin('DebugKit', function (RouteBuilder $routes) {
    // Routes connected here are prefixed with '/debug-kit' and
    // have the plugin route element set to 'DebugKit'.
    $routes->connect('/{controller}');
});
```

When creating plugin scopes, you can customize the path element used with the `path` option:

```
$routes->plugin('DebugKit', ['path' => '/debugger'], function (RouteBuilder $routes) {
    // Routes connected here are prefixed with '/debugger' and
    // have the plugin route element set to 'DebugKit'.
    $routes->connect('/{controller}');
});
```

When using scopes you can nest plugin scopes within prefix scopes:

```
$routes->prefix('Admin', function (RouteBuilder $routes) {
    $routes->plugin('DebugKit', function (RouteBuilder $routes) {
        $routes->connect('/{controller}');
    });
});
```

The above would create a route that looks like `/admin/debug-kit/{controller}`. It would have the `prefix`, and `plugin` route elements set. The *Plugin Routes* section has more information on building plugin routes.

Creating Links to Plugin Routes

You can create links that point to a plugin, by adding the plugin key to your URL array:

```
echo $this->Html->link(
    'New todo',
    ['plugin' => 'Todo', 'controller' => 'TodoItems', 'action' => 'create']
);
```

Conversely if the active request is a plugin request and you want to create a link that has no plugin you can do the following:

```
echo $this->Html->link(
    'New todo',
    ['plugin' => null, 'controller' => 'Users', 'action' => 'profile']
);
```

By setting `'plugin' => null` you tell the Router that you want to create a link that is not part of a plugin.

SEO-Friendly Routing

Some developers prefer to use dashes in URLs, as it's perceived to give better search engine rankings. The `DashedRoute` class can be used in your application with the ability to route plugin, controller, and camelized action names to a dashed URL.

For example, if we had a `ToDo` plugin, with a `TodoItems` controller, and a `showItems()` action, it could be accessed at `/to-do/todo-items/show-items` with the following router connection:

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

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

Matching Specific HTTP Methods

Routes can match specific HTTP methods using the HTTP verb helper methods:

```
$routes->scope('/', function (RouteBuilder $routes) {
    // This route only matches on POST requests.
    $routes->post(
        '/reviews/start',
        ['controller' => 'Reviews', 'action' => 'start']
    );

    // Match multiple verbs
    $routes->connect(
        '/reviews/start',
        [
            'controller' => 'Reviews',
            'action' => 'start',
        ]
```

(continues on next page)

(continued from previous page)

```
    )->setMethods(['POST', 'PUT']);
});
```

You can match multiple HTTP methods by using an array. Because the `_method` parameter is a routing key, it participates in both URL parsing and URL generation. To generate URLs for method specific routes you'll need to include the `_method` key when generating the URL:

```
$url = Router::url([
    'controller' => 'Reviews',
    'action' => 'start',
    '_method' => 'POST',
]);
```

Matching Specific Hostnames

Routes can use the `_host` option to only match specific hosts. You can use the `*.` wildcard to match any subdomain:

```
$routes->scope('/', function (RouteBuilder $routes) {
    // This route only matches on http://images.example.com
    $routes->connect(
        '/images/default-logo.png',
        ['controller' => 'Images', 'action' => 'default']
    )->setHost('images.example.com');

    // This route only matches on http://*.example.com
    $routes->connect(
        '/images/old-log.png',
        ['controller' => 'Images', 'action' => 'oldLogo']
    )->setHost('*.example.com');
});
```

The `_host` option is also used in URL generation. If your `_host` option specifies an exact domain, that domain will be included in the generated URL. However, if you use a wildcard, then you will need to provide the `_host` parameter when generating URLs:

```
// If you have this route
$routes->connect(
    '/images/old-log.png',
    ['controller' => 'Images', 'action' => 'oldLogo']
)->setHost('images.example.com');

// You need this to generate a url
echo Router::url([
    'controller' => 'Images',
    'action' => 'oldLogo',
    '_host' => 'images.example.com',
]);
```

Routing File Extensions

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

To handle different file extensions in your URLs, you can define the extensions using the `Cake\Routing\RouteBuilder::setExtensions()` method:

```
$routes->scope('/', function (RouteBuilder $routes) {
    $routes->setExtensions(['json', 'xml']);
});
```

This will enable the named extensions for all routes that are being connected in that scope **after** the `setExtensions()` call, including those that are being connected in nested scopes.

Note: Setting the extensions should be the first thing you do in a scope, as the extensions will only be applied to routes connected **after** the extensions are set.

Also be aware that re-opened scopes will **not** inherit extensions defined in previously opened scopes.

By using extensions, you tell the router to remove any matching file extensions from the URL, and then parse what remains. If you want to create a URL such as /page/title-of-page.html you would create your route using:

```
$routes->scope('/page', function (RouteBuilder $routes) {
    $routes->setExtensions(['json', 'xml', 'html']);
    $routes->connect(
        '/{title}',
        ['controller' => 'Pages', 'action' => 'view']
    )->setPass(['title']);
});
```

Then to create links which map back to the routes simply use:

```
$this->Html->link(
    'Link title',
    ['controller' => 'Pages', 'action' => 'view', 'title' => 'super-article', '_ext' =>
↪'html']
);
```

Route Scoped Middleware

While Middleware can be applied to your entire application, applying middleware to specific routing scopes offers more flexibility, as you can apply middleware only where it is needed allowing your middleware to not concern itself with how/where it is being applied.

Note: Applied scoped middleware will be run by RoutingMiddleware, normally at the end of your application's middleware queue.

Before middleware can be applied to a scope, it needs to be registered into the route collection:

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

$routes->registerMiddleware('csrf', new CsrfProtectionMiddleware());
$routes->registerMiddleware('cookies', new EncryptedCookieMiddleware());
```

Once registered, scoped middleware can be applied to specific scopes:

```
$routes->scope('/cms', function (RouteBuilder $routes) {
    // Enable CSRF & cookies middleware
    $routes->applyMiddleware('csrf', 'cookies');
    $routes->get('/articles/{action}/*', ['controller' => 'Articles']);
});
```

In situations where you have nested scopes, inner scopes will inherit the middleware applied in the containing scope:

```
$routes->scope('/api', function (RouteBuilder $routes) {
    $routes->applyMiddleware('ratelimit', 'auth.api');
    $routes->scope('/v1', function (RouteBuilder $routes) {
        $routes->applyMiddleware('v1compat');
        // Define routes here.
    });
});
```

In the above example, the routes defined in `/v1` will have 'ratelimit', 'auth.api', and 'v1compat' middleware applied. If you re-open a scope, the middleware applied to routes in each scope will be isolated:

```
$routes->scope('/blog', function (RouteBuilder $routes) {
    $routes->applyMiddleware('auth');
    // Connect the authenticated actions for the blog here.
});
$routes->scope('/blog', function (RouteBuilder $routes) {
    // Connect the public actions for the blog here.
});
```

In the above example, the two uses of the `/blog` scope do not share middleware. However, both of these scopes will inherit middleware defined in their enclosing scopes.

Grouping Middleware

To help keep your route code DRY (Do not Repeat Yourself) middleware can be combined into groups. Once combined groups can be applied like middleware can:

```
$routes->registerMiddleware('cookie', new EncryptedCookieMiddleware());
$routes->registerMiddleware('auth', new AuthenticationMiddleware());
$routes->registerMiddleware('csrf', new CsrfProtectionMiddleware());
$routes->middlewareGroup('web', ['cookie', 'auth', 'csrf']);

// Apply the group
$routes->applyMiddleware('web');
```

RESTful Routing

Router helps generate RESTful routes for your controllers. RESTful routes are helpful when you are creating API endpoints for your application. If we wanted to allow REST access to a recipe controller, we'd do something like this:

```
// In config/routes.php...

$routes->scope('/', function (RouteBuilder $routes) {
    $routes->setExtensions(['json']);
    $routes->resources('Recipes');
});
```

The first line sets up a number of default routes for REST access where method specifies the desired result format, for example, xml, json and rss. These routes are HTTP Request Method sensitive.

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)

Note: The default for pattern for resource IDs only matches integers or UUIDs. If your IDs are different you will have to supply a regular expression pattern via the `id` option, for example, `$builder->resources('Recipes', ['id' => '.*'])`.

The HTTP method being used is detected from a few different sources. The sources in order of preference are:

1. The `_method` POST variable
2. The `X_HTTP_METHOD_OVERRIDE` header.
3. The `REQUEST_METHOD` header

The `_method` POST variable is helpful in using a browser as a REST client (or anything else that can do POST). Just set the value of `_method` to the name of the HTTP request method you wish to emulate.

Creating Nested Resource Routes

Once you have connected resources in a scope, you can connect routes for sub-resources as well. Sub-resource routes will be prepended by the original resource name and a id parameter. For example:

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

Will generate resource routes for both `articles` and `comments`. The comments routes will look like:

```
/api/articles/{article_id}/comments
/api/articles/{article_id}/comments/{id}
```

You can get the `article_id` in `CommentsController` by:

```
$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:

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

The above would map the 'Comments' resource to the `App\Controller\Articles\CommentsController`. Having separate controllers lets you keep your controller logic simpler. The prefixes created this way are compatible with *Prefix Routing*.

Note: While you can nest resources as deeply as you require, it is not recommended to nest more than 2 resources together.

Limiting the Routes Created

By default CakePHP will connect 6 routes for each resource. If you'd like to only connect specific resource routes you can use the `only` option:

```
$routes->resources('Articles', [
    'only' => ['index', 'view']
]);
```

Would create read only resource routes. The route names are `create`, `update`, `view`, `index`, and `delete`.

The default **route name and controller action used** are as follows:

Route name	Controller action used
create	add
update	edit
view	view
index	index
delete	delete

Changing the Controller Actions Used

You may need to change the controller action names that are used when connecting routes. For example, if your `edit()` action is called `put()` you can use the `actions` key to rename the actions used:

```
$routes->resources('Articles', [
    'actions' => ['update' => 'put', 'create' => 'add']
]);
```

The above would use `put()` for the `edit()` action, and `add()` instead of `create()`.

Mapping Additional Resource Routes

You can map additional resource methods using the `map` option:

```
$routes->resources('Articles', [
   'map' => [
       'deleteAll' => [
           'action' => 'deleteAll',
           'method' => 'DELETE'
       ]
   ]
]);
// This would connect /articles/deleteAll
```

In addition to the default routes, this would also connect a route for */articles/delete-all*. By default the path segment will match the key name. You can use the 'path' key inside the resource definition to customize the path name:

```
$routes->resources('Articles', [
    'map' => [
        'updateAll' => [
            'action' => 'updateAll',
            'method' => 'PUT',
            'path' => '/update-many',
        ],
    ],
]);
// This would connect /articles/update-many
```

If you define 'only' and 'map', make sure that your mapped methods are also in the 'only' list.

Prefixed Resource Routing

Resource routes can be connected to controllers in routing prefixes by connecting routes within a prefixed scope or by using the `prefix` option:

```
$routes->resources('Articles', [
    'prefix' => 'Api',
]);
```

Custom Route Classes for Resource Routes

You can provide `connectOptions` key in the `$options` array for `resources()` to provide custom setting used by `connect()`:

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

URL Inflection for Resource Routes

By default, multi-worded controllers' URL fragments are the dashed form of the controller's name. For example, `BlogPostsController`'s URL fragment would be **/blog-posts**.

You can specify an alternative inflection type using the `inflect` option:

```
$routes->scope('/', function (RouteBuilder $routes) {
    $routes->resources('BlogPosts', [
        'inflect' => 'underscore' // Will use ``Inflector::underscore()``
    ]);
});
```

The above will generate URLs styled like: **/blog_posts**.

Changing the Path Element

By default resource routes use an inflected form of the resource name for the URL segment. You can set a custom URL segment with the `path` option:

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

Passed Arguments

Passed arguments are additional arguments or path segments that are used when making a request. They are often used to pass parameters to your controller methods.

```
http://localhost/calendars/view/recent/mark
```

In the above example, both `recent` and `mark` are passed arguments to `CalendarsController::view()`. Passed arguments are given to your controllers in three ways. First as arguments to the action method called, and secondly they are available in `$this->request->getParam('pass')` as a numerically indexed array. When using custom routes you can force particular parameters to go into the passed arguments as well.

If you were to visit the previously mentioned URL, and you had a controller action that looked like:

```
class CalendarsController extends AppController
{
    public function view($arg1, $arg2)
    {
        debug(func_get_args());
    }
}
```

You would get the following output:

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

This same data is also available at `$this->request->getParam('pass')` in your controllers, views, and helpers. The values in the pass array are numerically indexed based on the order they appear in the called URL:

```
debug($this->request->getParam('pass'));
```

Either of the above would output:

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

When generating URLs, using a *routing array* you add passed arguments as values without string keys in the array:

```
['controller' => 'Articles', 'action' => 'view', 5]
```

Since `5` has a numeric key, it is treated as a passed argument.

Generating URLs

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

static Cake\Routing\RouterBuilder::**reverse**(*$params*, *$full = false*)

Generating URLs or Reverse routing is a feature in CakePHP that is used to allow you to change your URL structure without having to modify all your code.

If you create URLs using strings like:

```
$this->Html->link('View', '/articles/view/' . $id);
```

And then later decide that `/articles` should really be called 'posts' instead, you would have to go through your entire application renaming URLs. However, if you defined your link like:

```
//`link()` uses Router::url() internally and accepts a routing array
```

(continues on next page)

(continued from previous page)

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

or:

```
//'Router::reverse()' operates on the request parameters array
//and will produce a url string, valid input for `link()`

$requestParams = Router::getRequest()->getAttribute('params');
$this->Html->link('View', Router::reverse($requestParams));
```

Then when you decided to change your URLs, you could do so by defining a route. This would change both the incoming URL mapping, as well as the generated URLs.

The choice of technique is determined by how well you can predict the routing array elements.

Using `Router::url()`

`Router::url()` allows you to use *routing arrays* in situations where the array elements required are fixed or easily deduced.

It will provide reverse routing when the destination url is well defined:

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

It is also useful when the destination is unknown but follows a well defined pattern:

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

Elements with numeric keys are treated as *Passed Arguments*.

When using routing arrays, you can define both query string parameters and document fragments using special keys:

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

// Will generate a URL like.
/articles/index?page=1#top
```

You can also use any of the special route elements when generating URLs:

- `_ext` Used for *Routing File Extensions* routing.

- `_base` Set to `false` to remove the base path from the generated URL. If your application is not in the root directory, this can be used to generate URLs that are 'cake relative'.
- `_scheme` Set to create links on different schemes like `webcal` or `ftp`. Defaults to the current scheme.
- `_host` Set the host to use for the link. Defaults to the current host.
- `_port` Set the port if you need to create links on non-standard ports.
- `_method` Define the HTTP verb the URL is for.
- `_full` If `true` the value of `App.fullBaseUrl` mentioned in *General Configuration* will be prepended to generated URLs.
- `_https` Set to `true` to convert the generated URL to https or `false` to force http. Prior to 4.5.0 use `_ssl`
- `_name` Name of route. If you have setup named routes, you can use this key to specify it.

Using `Router::reverse()`

`Router::reverse()` allows you to use the *Request Parameters* in cases where the current URL with some modification is the basis for the destination and the elements of the current URL are unpredictable.

As an example, imagine a blog that allowed users to create **Articles** and **Comments**, and to mark both as either *published* or *draft*. Both the index page URLs might include the user id. The **Comments** URL might also include an article id to identify what article the comment refers to.

Here are urls for this scenario:

```
/articles/index/42
/comments/index/42/18
```

When the author uses these pages, it would be convenient to include links that allow the page to be displayed with all results, published only, or draft only.

To keep the code DRY, it would be best to include the links through an element:

```
// element/filter_published.php

$params = $this->getRequest()->getAttribute('params');

/* prepare url for Draft */
$params = Hash::insert($params, '?.published', 0);
echo $this->Html->link(__('Draft'), Router::reverse($params));

/* Prepare url for Published */
$params = Hash::insert($params, '?.published', 1);
echo $this->Html->link(__('Published'), Router::reverse($params));

/* Prepare url for All */
$params = Hash::remove($params, '?.published');
echo $this->Html->link(__('All'), Router::reverse($params));
```

The links generated by these method calls would include one or two pass parameters depending on the structure of the current URL. And the code would work for any future URL, for example, if you started using pathPrefixes or if you added more pass parameters.

Routing Arrays vs Request Parameters

The significant difference between the two arrays and their use in these reverse routing methods is in the way they include pass parameters.

Routing arrays include pass parameters as un-keyed values in the array:

```
$url = [
    'controller' => 'Articles',
    'action' => 'View',
    $id, //a pass parameter
    'page' => 3, //a query argument
];
```

Request parameters include pass parameters on the 'pass' key of the array:

```
$url = [
    'controller' => 'Articles',
    'action' => 'View',
    'pass' => [$id], //the pass parameters
    '?' => ['page' => 3], //the query arguments
];
```

So it is possible, if you wish, to convert the request parameters into a routing array or vice versa.

Generating Asset URLs

The `Asset` class provides methods for generating URLs to your application's css, javascript, images and other static asset files:

```
use Cake\Routing\Asset;

// Generate a URL to APP/webroot/js/app.js
$js = Asset::scriptUrl('app.js');

// Generate a URL to APP/webroot/css/app.css
$css = Asset::cssUrl('app.css');

// Generate a URL to APP/webroot/image/logo.png
$img = Asset::imageUrl('logo.png');

// Generate a URL to APP/webroot/files/upload/photo.png
$file = Asset::url('files/upload/photo.png');
```

The above methods also accept an array of options as their second parameter:

- `fullBase` Append the full URL with domain name.
- `pathPrefix` Path prefix for relative URLs.
- `plugin` You can provide `false` to prevent paths from being treated as a plugin asset.
- `timestamp` Overrides the value of `Asset.timestamp` in Configure. Set to `false` to skip timestamp generation. Set to `true` to apply timestamps when debug is true. Set to `'force'` to always enable timestamping regardless of debug value.

```
// Generates http://example.org/img/logo.png
$img = Asset::url('logo.png', ['fullBase' => true]);

// Generates /img/logo.png?1568563625
// Where the timestamp is the last modified time of the file.
$img = Asset::url('logo.png', ['timestamp' => true]);
```

To generate asset URLs for files in plugins use *plugin syntax*:

```
// Generates `/debug_kit/img/cake.png`
$img = Asset::imageUrl('DebugKit.cake.png');
```

Redirect Routing

Redirect routing allows you to issue HTTP status 30x redirects for incoming routes, and point them at different URLs. This is useful when you want to inform client applications that a resource has moved and you don't want to expose two URLs for the same content.

Redirection routes are different from normal routes as they perform an actual header redirection if a match is found. The redirection can occur to a destination within your application or an outside location:

```
$routes->scope('/', function (RouteBuilder $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.
    );
})
```

Redirects `/home/*` to `/articles/view` and passes the parameters to `/articles/view`. Using an array as the redirect destination allows you to use other routes to define where a URL string should be redirected to. You can redirect to external locations using string URLs as the destination:

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

This would redirect `/articles/*` to `http://google.com` with a HTTP status of 302.

Entity Routing

Entity routing allows you to use an entity, an array or object implement `ArrayAccess` as the source of routing parameters. This allows you to refactor routes more easily, and generate URLs with less code. For example, if you start off with a route that looks like:

```
$routes->get(
    '/view/{id}',
    ['controller' => 'Articles', 'action' => 'view'],
```

(continues on next page)

(continued from previous page)

```
    'articles:view'
);
```

You can generate URLs to this route using:

```
// $article is an entity in the local scope.
Router::url(['_name' => 'articles:view', 'id' => $article->id]);
```

Later on, you may want to expose the article slug in the URL for SEO purposes. In order to do this you would need to update everywhere you generate a URL to the `articles:view` route, which could take some time. If we use entity routes we pass the entire article entity into URL generation allowing us to skip any rework when URLs require more parameters:

```
use Cake\Routing\Route\EntityRoute;

// Create entity routes for the rest of this scope.
$routes->setRouteClass(EntityRoute::class);

// Create the route just like before.
$routes->get(
    '/view/{id}/{slug}',
    ['controller' => 'Articles', 'action' => 'view'],
    'articles:view'
);
```

Now we can generate URLs using the `_entity` key:

```
Router::url(['_name' => 'articles:view', '_entity' => $article]);
```

This will extract both the `id` property and the `slug` property out of the provided entity.

Custom Route Classes

Custom route classes allow you to extend and change how individual routes parse requests and handle reverse routing. Route classes have a few conventions:

- Route classes are expected to be found in the `Routing\\Route` namespace of your application or plugin.
- Route classes should extend `Cake\Routing\Route\Route`.
- Route classes should implement one or both of `match()` and/or `parse()`.

The `parse()` method is used to parse an incoming URL. It should generate an array of request parameters that can be resolved into a controller & action. Return `null` from this method to indicate a match failure.

The `match()` method is used to match an array of URL parameters and create a string URL. If the URL parameters do not match the route `false` should be returned.

You can use a custom route class when making a route by using the `routeClass` option:

```
$routes->connect(
    '/{slug}',
    ['controller' => 'Articles', 'action' => 'view'],
    ['routeClass' => 'SlugRoute']
```

(continues on next page)

(continued from previous page)

```
);

// Or by setting the routeClass in your scope.
$routes->scope('/', function (RouteBuilder $routes) {
    $routes->setRouteClass('SlugRoute');
    $routes->connect(
        '/{slug}',
        ['controller' => 'Articles', 'action' => 'view']
    );
});
```

This route would create an instance of `SlugRoute` and allow you to implement custom parameter handling. You can use plugin route classes using standard *plugin syntax*.

Default Route Class

static Cake\Routing\RouterBuilder::**setRouteClass**(*$routeClass = null*)

If you want to use an alternate route class for your routes besides the default `Route`, you can do so by calling `RouterBuilder::setRouteClass()` before setting up any routes and avoid having to specify the `routeClass` option for each route. For example using:

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

$routes->setRouteClass(DashedRoute::class);
```

will cause all routes connected after this to use the `DashedRoute` route class. Calling the method without an argument will return current default route class.

Fallbacks Method

Cake\Routing\RouterBuilder::**fallbacks**(*$routeClass = null*)

The fallbacks method is a simple shortcut for defining default routes. The method uses the passed routing class for the defined rules or if no class is provided the class returned by `RouterBuilder::setRouteClass()` is used.

Calling fallbacks like so:

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

$routes->fallbacks(DashedRoute::class);
```

Is equivalent to the following explicit calls:

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

$routes->connect('/{controller}', ['action' => 'index'], ['routeClass' =>␣
↪DashedRoute::class]);
$routes->connect('/{controller}/{action}/*', [], ['routeClass' => DashedRoute::class]);
```

Note: Using the default route class (`Route`) with fallbacks, or any route with `{plugin}` and/or `{controller}` route elements will result in inconsistent URL case.

Warning: Fallback route templates are very generic and allow URLs to be generated and parsed for controllers & actions that do not exist. Fallback URLs can also introduce ambiguity and duplication in your URLs.

As your application grows, it is recommended to move away from fallback URLs and explicitly define the routes in your application.

Creating Persistent URL Parameters

You can hook into the URL generation process using URL filter functions. Filter functions are called *before* the URLs are matched against the routes, this allows you to prepare URLs before routing.

Callback filter functions should expect the following parameters:

- `$params` The URL parameter array being processed.
- `$request` The current request (`Cake\Http\ServerRequest` instance).

The URL filter function should *always* return the parameters even if unmodified.

URL filters allow you to implement features like persistent parameters:

```
Router::addUrlFilter(function (array $params, ServerRequest $request) {
    if ($request->getParam('lang') && !isset($params['lang'])) {
        $params['lang'] = $request->getParam('lang');
    }
    return $params;
});
```

Filter functions are applied in the order they are connected.

Another use case is changing a certain route on runtime (plugin routes for example):

```
Router::addUrlFilter(function (array $params, ServerRequest $request) {
    if (empty($params['plugin']) || $params['plugin'] !== 'MyPlugin' || empty($params[
↪'controller'])) {
        return $params;
    }
    if ($params['controller'] === 'Languages' && $params['action'] === 'view') {
        $params['controller'] = 'Locations';
        $params['action'] = 'index';
        $params['language'] = $params[0];
        unset($params[0]);
    }
    return $params;
});
```

This will alter the following route:

```
Router::url(['plugin' => 'MyPlugin', 'controller' => 'Languages', 'action' => 'view', 'es
↪']);
```

into this:

```
Router::url(['plugin' => 'MyPlugin', 'controller' => 'Locations', 'action' => 'index',
↪'language' => 'es']);
```

Warning: If you are using the caching features of routing-middleware you must define the URL filters in your application `bootstrap()` as filters are not part of the cached data.

CHAPTER 9

Request & Response Objects

The request and response objects provide an abstraction around HTTP requests and responses. The request object in CakePHP allows you to introspect an incoming request, while the response object allows you to effortlessly create HTTP responses from your controllers.

Request

class Cake\Http**ServerRequest**

`ServerRequest` is the default request object used in CakePHP. It centralizes a number of features for interrogating and interacting with request data. On each request one Request is created and then passed by reference to the various layers of an application that use request data. By default the request is assigned to `$this->request`, and is available in Controllers, Cells, Views and Helpers. You can also access it in Components using the controller reference.

Changed in version 4.4.0: The `ServerRequest` is available via DI. So you can get it from container or use it as a dependency for your service.

Some of the duties `ServerRequest` performs include:

- Processing the GET, POST, and FILES arrays into the data structures you are familiar with.
- Providing environment introspection pertaining to the request. Information like the headers sent, the client's IP address, and the subdomain/domain names the server your application is running on.
- Providing access to request parameters both as array indexes and object properties.

CakePHP's request object implements the PSR-7 ServerRequestInterface[105] making it easier to use libraries from outside of CakePHP.

[105] https://www.php-fig.org/psr/psr-7/

Request Parameters

The request exposes routing parameters through the `getParam()` method:

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

To get all routing parameters as an array use `getAttribute()`:

```
$parameters = $this->request->getAttribute('params');
```

All *Route Elements* are accessed through this interface.

In addition to *Route Elements*, you also often need access to *Passed Arguments*. These are both available on the request object as well:

```
// Passed arguments
$passedArgs = $this->request->getParam('pass');
```

Will all provide you access to the passed arguments. There are several important/useful parameters that CakePHP uses internally, these are also all found in the routing parameters:

- `plugin` The plugin handling the request. Will be null when there is no plugin.
- `controller` The controller handling the current request.
- `action` The action handling the current request.
- `prefix` The prefix for the current action. See *Prefix Routing* for more information.

Query String Parameters

`Cake\Http\ServerRequest::getQuery`(*$name*, *$default = null*)

Query string parameters can be read using the `getQuery()` method:

```
// URL is /posts/index?page=1&sort=title
$page = $this->request->getQuery('page');
```

You can either directly access the query property, or you can use `getQuery()` method to read the URL query array in an error-free manner. Any keys that do not exist will return `null`:

```
$foo = $this->request->getQuery('value_that_does_not_exist');
// $foo === null

// You can also provide default values
$foo = $this->request->getQuery('does_not_exist', 'default val');
```

If you want to access all the query parameters you can use `getQueryParams()`:

```
$query = $this->request->getQueryParams();
```

Request Body Data

Cake\Http\ServerRequest::**getData**(*$name*, *$default = null*)

All POST data normally available through PHP's `$_POST` global variable can be accessed using *Cake\Http\ServerRequest::getData()*. For example:

```
// An input with a name attribute equal to 'title' is accessible at
$title = $this->request->getData('title');
```

You can use a dot separated names to access nested data. For example:

```
$value = $this->request->getData('address.street_name');
```

For non-existent names the `$default` value will be returned:

```
$foo = $this->request->getData('value.that.does.not.exist');
// $foo == null
```

You can also use body-parser-middleware to parse request body of different content types into an array, so that it's accessible through `ServerRequest::getData()`.

If you want to access all the data parameters you can use `getParsedBody()`:

```
$data = $this->request->getParsedBody();
```

File Uploads

Uploaded files can be accessed through the request body data, using the *Cake\Http\ServerRequest::getData()* method described above. For example, a file from an input element with a name attribute of `attachment`, can be accessed like this:

```
$attachment = $this->request->getData('attachment');
```

By default file uploads are represented in the request data as objects that implement \Psr\Http\Message\UploadedFileInterface[106]. In the current implementation, the `$attachment` variable in the above example would by default hold an instance of `\Laminas\Diactoros\UploadedFile`.

Accessing the uploaded file details is fairly simple, here's how you can obtain the same data as provided by the old style file upload array:

```
$name = $attachment->getClientFilename();
$type = $attachment->getClientMediaType();
$size = $attachment->getSize();
$tmpName = $attachment->getStream()->getMetadata('uri');
$error = $attachment->getError();
```

Moving the uploaded file from its temporary location to the desired target location, doesn't require manually accessing the temporary file, instead it can be easily done by using the objects `moveTo()` method:

```
$attachment->moveTo($targetPath);
```

[106] https://www.php-fig.org/psr/psr-7/#16-uploaded-files

In an HTTP environment, the `moveTo()` method will automatically validate whether the file is an actual uploaded file, and throw an exception in case necessary. In an CLI environment, where the concept of uploading files doesn't exist, it will allow to move the file that you've referenced irrespective of its origins, which makes testing file uploads possible.

`Cake\Http\ServerRequest::getUploadedFile(`*$path*`)`

Returns the uploaded file at a specific path. The path uses the same dot syntax as the *Cake\Http\ServerRequest::getData()* method:

```
$attachment = $this->request->getUploadedFile('attachment');
```

Unlike *Cake\Http\ServerRequest::getData()*, *Cake\Http\ServerRequest::getUploadedFile()* would only return data when an actual file upload exists for the given path, if there is regular, non-file request body data present at the given path, then this method will return `null`, just like it would for any non-existent path.

`Cake\Http\ServerRequest::getUploadedFiles()`

Returns all uploaded files in a normalized array structure. For the above example with the file input name of `attachment`, the structure would look like:

```
[
      'attachment' => object(Laminas\Diactoros\UploadedFile) {
          // ...
      }
]
```

`Cake\Http\ServerRequest::withUploadedFiles(`*array $files*`)`

This method sets the uploaded files of the request object, it accepts an array of objects that implement \Psr\Http\Message\UploadedFileInterface[107]. It will replace all possibly existing uploaded files:

```
$files = [
    'MyModel' => [
        'attachment' => new \Laminas\Diactoros\UploadedFile(
            $streamOrFile,
            $size,
            $errorStatus,
            $clientFilename,
            $clientMediaType
        ),
        'anotherAttachment' => new \Laminas\Diactoros\UploadedFile(
            '/tmp/hfz6dbn.tmp',
            123,
            \UPLOAD_ERR_OK,
            'attachment.txt',
            'text/plain'
        ),
    ],
];

$this->request = $this->request->withUploadedFiles($files);
```

Note: Uploaded files that have been added to the request via this method, will *not* be available in the request body data, ie you cannot retrieve them via *Cake\Http\ServerRequest::getData()*! If you need them in the

[107] https://www.php-fig.org/psr/psr-7/#16-uploaded-files

request data (too), then you have to set them via `Cake\Http\ServerRequest::withData()` or `Cake\Http\ServerRequest::withParsedBody()`.

PUT, PATCH or DELETE Data

`Cake\Http\ServerRequest::input`(*$callback*[, *$options*])

When building REST services, you often accept request data on `PUT` and `DELETE` requests. Any `application/x-www-form-urlencoded` request body data will automatically be parsed and set to `$this->data` for `PUT` and `DELETE` requests. If you are accepting JSON or XML data, see below for how you can access those request bodies.

When accessing the input data, you can decode it with an optional function. This is useful when interacting with XML or JSON request body content. Additional parameters for the decoding function can be passed as arguments to `input()`:

```
$jsonData = $this->request->input('json_decode');
```

Environment Variables (from $_SERVER and $_ENV)

`Cake\Http\ServerRequest::putenv`(*$key*, *$value* = *null*)

`ServerRequest::getEnv()` is a wrapper for `getenv()` global function and acts as a getter/setter for environment variables without having to modify globals `$_SERVER` and `$_ENV`:

```
// Get the host
$host = $this->request->getEnv('HTTP_HOST');

// Set a value, generally helpful in testing.
$this->request->withEnv('REQUEST_METHOD', 'POST');
```

To access all the environment variables in a request use `getServerParams()`:

```
$env = $this->request->getServerParams();
```

XML or JSON Data

Applications employing *REST* often exchange data in non-URL-encoded post bodies. You can read input data in any format using *input()*. By providing a decoding function, you can receive the content in a deserialized format:

```
// Get JSON encoded data submitted to a PUT/POST action
$jsonData = $this->request->input('json_decode');
```

Some deserializing methods require additional parameters when called, such as the 'as array' parameter on `json_decode`. If you want XML converted into a DOMDocument object, *input()* supports passing in additional parameters as well:

```
// Get XML encoded data submitted to a PUT/POST action
$data = $this->request->input('Cake\Utility\Xml::build', ['return' => 'domdocument']);
```

Path Information

The request object also provides useful information about the paths in your application. The `base` and `webroot` attributes are useful for generating URLs, and determining whether or not your application is in a subdirectory. The attributes you can use are:

```
// Assume the current request URL is /subdir/articles/edit/1?page=1

// Holds /subdir/articles/edit/1?page=1
$here = $request->getRequestTarget();

// Holds /subdir
$base = $request->getAttribute('base');

// Holds /subdir/
$base = $request->getAttribute('webroot');
```

Checking Request Conditions

`Cake\Http\ServerRequest::is`(*$type*, *$args...*)

The request object provides a way to inspect certain conditions in a given request. By using the `is()` method you can check a number of common conditions, as well as inspect other application specific request criteria:

```
$isPost = $this->request->is('post');
```

You can also extend the request detectors that are available, by using *Cake\Http\ServerRequest::addDetector()* to create new kinds of detectors. There are different types of detectors that you can create:

- Environment value comparison - Compares a value fetched from `env()` for equality with the provided value.
- Header value comparison - If the specified header exists with the specified value, or if the callable returns true.
- Pattern value comparison - Pattern value comparison allows you to compare a value fetched from `env()` to a regular expression.
- Option based comparison - Option based comparisons use a list of options to create a regular expression. Subsequent calls to add an already defined options detector will merge the options.
- Callback detectors - Callback detectors allow you to provide a 'callback' type to handle the check. The callback will receive the request object as its only parameter.

`Cake\Http\ServerRequest::addDetector`(*$name*, *$options*)

Some examples would be:

```
// Add an environment detector.
$this->request->addDetector(
    'post',
    ['env' => 'REQUEST_METHOD', 'value' => 'POST']
);

// Add a pattern value detector.
$this->request->addDetector(
    'iphone',
    ['env' => 'HTTP_USER_AGENT', 'pattern' => '/iPhone/i']
```

(continues on next page)

(continued from previous page)

```
);

// Add an option detector
$this->request->addDetector('internalIp', [
    'env' => 'CLIENT_IP',
    'options' => ['192.168.0.101', '192.168.0.100']
]);


// Add a header detector with value comparison
$this->request->addDetector('fancy', [
    'env' => 'CLIENT_IP',
    'header' => ['X-Fancy' => 1]
]);

// Add a header detector with callable comparison
$this->request->addDetector('fancy', [
    'env' => 'CLIENT_IP',
    'header' => ['X-Fancy' => function ($value, $header) {
        return in_array($value, ['1', '0', 'yes', 'no'], true);
    }]
]);

// Add a callback detector. Must be a valid callable.
$this->request->addDetector(
    'awesome',
    function ($request) {
        return $request->getParam('awesome');
    }
);

// Add a detector that uses additional arguments.
$this->request->addDetector(
    'csv',
    [
        'accept' => ['text/csv'],
        'param' => '_ext',
        'value' => 'csv',
    ]
);
```

There are several built-in detectors that you can use:

- `is('get')` Check to see whether the current request is a GET.
- `is('put')` Check to see whether the current request is a PUT.
- `is('patch')` Check to see whether the current request is a PATCH.
- `is('post')` Check to see whether the current request is a POST.
- `is('delete')` Check to see whether the current request is a DELETE.
- `is('head')` Check to see whether the current request is HEAD.
- `is('options')` Check to see whether the current request is OPTIONS.

- `is('ajax')` Check to see whether the current request came with X-Requested-With = XMLHttpRequest.
- `is('ssl')` Check to see whether the request is via SSL.
- `is('flash')` Check to see whether the request has a User-Agent of Flash.
- `is('json')` Check to see whether the request URL has 'json' extension or the *Accept* header is set to 'application/json'.
- `is('xml')` Check to see whether the request URL has 'xml' extension or the *Accept* header is set to 'application/xml' or 'text/xml'.

`ServerRequest` also includes methods like *Cake\Http\ServerRequest::domain()*, *Cake\Http\ServerRequest::subdomains()* and *Cake\Http\ServerRequest::host()* to make applications that use subdomains simpler.

Session Data

To access the session for a given request use the `getSession()` method or use the `session` attribute:

```
$session = $this->request->getSession();
$session = $this->request->getAttribute('session');

$data = $session->read('sessionKey');
```

For more information, see the *Sessions* documentation for how to use the session object.

Host and Domain Name

`Cake\Http\ServerRequest::`**`domain`**`(`*$tldLength = 1*`)`

Returns the domain name your application is running on:

```
// Prints 'example.org'
echo $request->domain();
```

`Cake\Http\ServerRequest::`**`subdomains`**`(`*$tldLength = 1*`)`

Returns the subdomains your application is running on as an array:

```
// Returns ['my', 'dev'] for 'my.dev.example.org'
$subdomains = $request->subdomains();
```

`Cake\Http\ServerRequest::`**`host`**`()`

Returns the host your application is on:

```
// Prints 'my.dev.example.org'
echo $request->host();
```

Reading the HTTP Method

`Cake\Http\ServerRequest::getMethod()`

Returns the HTTP method the request was made with:

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

Restricting Which HTTP method an Action Accepts

`Cake\Http\ServerRequest::allowMethod($methods)`

Set allowed HTTP methods. If not matched, will throw `MethodNotAllowedException`. The 405 response will include the required `Allow` header with the passed methods:

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

Reading HTTP Headers

Allows you to access any of the `HTTP_*` headers that were used for the request. For example:

```
// Get the header as a string
$userAgent = $this->request->getHeaderLine('User-Agent');

// Get an array of all values.
$acceptHeader = $this->request->getHeader('Accept');

// Check if a header exists
$hasAcceptHeader = $this->request->hasHeader('Accept');
```

While some apache installs don't make the `Authorization` header accessible, CakePHP will make it available through apache specific methods as required.

`Cake\Http\ServerRequest::referer($local = true)`

Returns the referring address for the request.

`Cake\Http\ServerRequest::clientIp()`

Returns the current visitor's IP address.

Trusting Proxy Headers

If your application is behind a load balancer or running on a cloud service, you will often get the load balancer host, port and scheme in your requests. Often load balancers will also send `HTTP-X-Forwarded-*` headers with the original values. The forwarded headers will not be used by CakePHP out of the box. To have the request object use these headers set the `trustProxy` property to `true`:

```
$this->request->trustProxy = true;

// These methods will now use the proxied headers.
$port = $this->request->port();
$host = $this->request->host();
$scheme = $this->request->scheme();
$clientIp = $this->request->clientIp();
```

Once proxies are trusted the `clientIp()` method will use the *last* IP address in the `X-Forwarded-For` header. If your application is behind multiple proxies, you can use `setTrustedProxies()` to define the IP addresses of proxies in your control:

```
$request->setTrustedProxies(['127.1.1.1', '127.8.1.3']);
```

After proxies are trusted `clientIp()` will use the first IP address in the `X-Forwarded-For` header providing it is the only value that isn't from a trusted proxy.

Checking Accept Headers

`Cake\Http\ServerRequest::accepts`(*$type = null*)

Find out which content types the client accepts, or check whether it accepts a particular type of content.

Get all types:

```
$accepts = $this->request->accepts();
```

Check for a single type:

```
$acceptsJson = $this->request->accepts('application/json');
```

`Cake\Http\ServerRequest::acceptLanguage`(*$language = null*)

Get all the languages accepted by the client, or check whether a specific language is accepted.

Get the list of accepted languages:

```
$acceptsLanguages = $this->request->acceptLanguage();
```

Check whether a specific language is accepted:

```
$acceptsSpanish = $this->request->acceptLanguage('es-es');
```

Reading Cookies

Request cookies can be read through a number of methods:

```
// Get the cookie value, or null if the cookie is missing.
$rememberMe = $this->request->getCookie('remember_me');

// Read the value, or get the default of 0
$rememberMe = $this->request->getCookie('remember_me', 0);

// Get all cookies as an hash
$cookies = $this->request->getCookieParams();

// Get a CookieCollection instance
$cookies = $this->request->getCookieCollection()
```

See the *Cake\Http\Cookie\CookieCollection* documentation for how to work with cookie collection.

Uploaded Files

Requests expose the uploaded file data in `getData()` or `getUploadedFiles()` as `UploadedFileInterface` objects:

```
// Get a list of UploadedFile objects
$files = $request->getUploadedFiles();

// Read the file data.
$files[0]->getStream();
$files[0]->getSize();
$files[0]->getClientFileName();

// Move the file.
$files[0]->moveTo($targetPath);
```

Manipulating URIs

Requests contain a URI object, which contains methods for interacting with the requested URI:

```
// Get the URI
$uri = $request->getUri();

// Read data out of the URI.
$path = $uri->getPath();
$query = $uri->getQuery();
$host = $uri->getHost();
```

Response

class Cake\Http**Response**

Cake\Http\Response is the default response class in CakePHP. It encapsulates a number of features and functionality for generating HTTP responses in your application. It also assists in testing, as it can be mocked/stubbed allowing you to inspect headers that will be sent.

`Response` provides an interface to wrap the common response-related tasks such as:

- Sending headers for redirects.
- Sending content type headers.
- Sending any header.
- Sending the response body.

Dealing with Content Types

Cake\Http\Response::**withType**(*$contentType = null*)

You can control the Content-Type of your application's responses with *Cake\Http\Response::withType()*. If your application needs to deal with content types that are not built into Response, you can map them with `setTypeMap()` as well:

```
// Add a vCard type
$this->response->setTypeMap('vcf', ['text/v-card']);

// Set the response Content-Type to vcard.
$this->response = $this->response->withType('vcf');
```

Usually, you'll want to map additional content types in your controller's `beforeFilter()` callback, so you can leverage the automatic view switching features of `RequestHandlerComponent` if you are using it.

Sending Files

Cake\Http\Response::**withFile**(*string $path*, *array $options = []*)

There are times when you want to send files as responses for your requests. You can accomplish that by using *Cake\Http\Response::withFile()*:

```
public function sendFile($id)
{
    $file = $this->Attachments->getFile($id);
    $response = $this->response->withFile($file['path']);
    // Return the response to prevent controller from trying to render
    // a view.
    return $response;
}
```

As shown in the above example, you must pass the file path to the method. CakePHP will send a proper content type header if it's a known file type listed in *Cake\Http\Response::$_mimeTypes*. You can add new types prior to calling *Cake\Http\Response::withFile()* by using the *Cake\Http\Response::withType()* method.

If you want, you can also force a file to be downloaded instead of displayed in the browser by specifying the options:

```
$response = $this->response->withFile(
    $file['path'],
    ['download' => true, 'name' => 'foo']
);
```

The supported options are:

name
: The name allows you to specify an alternate file name to be sent to the user.

download
: A boolean value indicating whether headers should be set to force download.

Sending a String as File

You can respond with a file that does not exist on the disk, such as a pdf or an ics generated on the fly from a string:

```
public function sendIcs()
{
    $icsString = $this->Calendars->generateIcs();
    $response = $this->response;

    // Inject string content into response body
    $response = $response->withStringBody($icsString);

    $response = $response->withType('ics');

    // Optionally force file download
    $response = $response->withDownload('filename_for_download.ics');

    // Return response object to prevent controller from trying to render
    // a view.
    return $response;
}
```

Setting Headers

Cake\Http\Response::**withHeader**(*$header*, *$value*)

Setting headers is done with the *Cake\Http\Response::withHeader()* method. Like all of the PSR-7 interface methods, this method returns a *new* instance with the new header:

```
// Add/replace a header
$response = $response->withHeader('X-Extra', 'My header');

// Set multiple headers
$response = $response->withHeader('X-Extra', 'My header')
    ->withHeader('Location', 'http://example.com');

// Append a value to an existing header
$response = $response->withAddedHeader('Set-Cookie', 'remember_me=1');
```

Headers are not sent when set. Instead, they are held until the response is emitted by `Cake\Http\Server`.

You can now use the convenience method `Cake\Http\Response::withLocation()` to directly set or get the redirect location header.

Setting the Body

`Cake\Http\Response::**withStringBody**(*$string*)`

To set a string as the response body, do the following:

```
// Set a string into the body
$response = $response->withStringBody('My Body');

// If you want a json response
$response = $response->withType('application/json')
    ->withStringBody(json_encode(['Foo' => 'bar']));
```

`Cake\Http\Response::**withBody**(*$body*)`

To set the response body, use the `withBody()` method, which is provided by the `Laminas\Diactoros\MessageTrait`:

```
$response = $response->withBody($stream);
```

Be sure that `$stream` is a `Psr\Http\Message\StreamInterface` object. See below on how to create a new stream.

You can also stream responses from files using `Laminas\Diactoros\Stream` streams:

```
// To stream from a file
use Laminas\Diactoros\Stream;

$stream = new Stream('/path/to/file', 'rb');
$response = $response->withBody($stream);
```

You can also stream responses from a callback using the `CallbackStream`. This is useful when you have resources like images, CSV files or PDFs you need to stream to the client:

```
// Streaming from a callback
use Cake\Http\CallbackStream;

// Create an image.
$img = imagecreate(100, 100);
// ...

$stream = new CallbackStream(function () use ($img) {
    imagepng($img);
});
$response = $response->withBody($stream);
```

Setting the Character Set

Cake\Http\Response::**withCharset**(*$charset*)

Sets the charset that will be used in the response:

```
$this->response = $this->response->withCharset('UTF-8');
```

Interacting with Browser Caching

Cake\Http\Response::**withDisabledCache**()

You sometimes need to force browsers not to cache the results of a controller action. *Cake\Http\Response::withDisabledCache()* is intended for just that:

```
public function index()
{
    // Disable caching
    $this->response = $this->response->withDisabledCache();
}
```

Warning: Disabling caching from SSL domains while trying to send files to Internet Explorer can result in errors.

Cake\Http\Response::**withCache**(*$since*, *$time = '+1 day'*)

You can also tell clients that you want them to cache responses. By using *Cake\Http\Response::withCache()*:

```
public function index()
{
    // Enable caching
    $this->response = $this->response->withCache('-1 minute', '+5 days');
}
```

The above would tell clients to cache the resulting response for 5 days, hopefully speeding up your visitors' experience. The `withCache()` method sets the `Last-Modified` value to the first argument. `Expires` header and the `max-age` directive are set based on the second parameter. Cache-Control's `public` directive is set as well.

Fine Tuning HTTP Cache

One of the best and easiest ways of speeding up your application is to use HTTP cache. Under this caching model, you are only required to help clients decide if they should use a cached copy of the response by setting a few headers such as modified time and response entity tag.

Rather than forcing you to code the logic for caching and for invalidating (refreshing) it once the data has changed, HTTP uses two models, expiration and validation, which usually are much simpler to use.

Apart from using *Cake\Http\Response::withCache()*, you can also use many other methods to fine-tune HTTP cache headers to take advantage of browser or reverse proxy caching.

The Cache Control Header

Cake\Http\Response::**withSharable**(*$public*, *$time = null*)

Used under the expiration model, this header contains multiple indicators that can change the way browsers or proxies use the cached content. A `Cache-Control` header can look like this:

```
Cache-Control: private, max-age=3600, must-revalidate
```

`Response` class helps you set this header with some utility methods that will produce a final valid `Cache-Control` header. The first is the `withSharable()` method, which indicates whether a response is to be considered sharable across different users or clients. This method actually controls the `public` or `private` part of this header. Setting a response as private indicates that all or part of it is intended for a single user. To take advantage of shared caches, the control directive must be set as public.

The second parameter of this method is used to specify a `max-age` for the cache, which is the number of seconds after which the response is no longer considered fresh:

```
public function view()
{
    // ...
    // Set the Cache-Control as public for 3600 seconds
    $this->response = $this->response->withSharable(true, 3600);
}

public function my_data()
{
    // ...
    // Set the Cache-Control as private for 3600 seconds
    $this->response = $this->response->withSharable(false, 3600);
}
```

`Response` exposes separate methods for setting each of the directives in the `Cache-Control` header.

The Expiration Header

Cake\Http\Response::**withExpires**(*$time*)

You can set the `Expires` header to a date and time after which the response is no longer considered fresh. This header can be set using the `withExpires()` method:

```
public function view()
{
    $this->response = $this->response->withExpires('+5 days');
}
```

This method also accepts a `DateTime` instance or any string that can be parsed by the `DateTime` class.

The Etag Header

Cake\Http\Response::**withEtag**(*$tag*, *$weak = false*)

Cache validation in HTTP is often used when content is constantly changing, and asks the application to only generate the response contents if the cache is no longer fresh. Under this model, the client continues to store pages in the cache, but it asks the application every time whether the resource has changed, instead of using it directly. This is commonly used with static resources such as images and other assets.

The `withEtag()` method (called entity tag) is a string that uniquely identifies the requested resource, as a checksum does for a file, in order to determine whether it matches a cached resource.

To take advantage of this header, you must either call the `checkNotModified()` method manually or include the *Checking HTTP Cache* in your controller:

```
public function index()
{
    $articles = $this->Articles->find('all')->all();

    // Simple checksum of the article contents.
    // You should use a more efficient implementation
    // in a real world application.
    $checksum = md5(json_encode($articles));

    $response = $this->response->withEtag($checksum);
    if ($response->checkNotModified($this->request)) {
        return $response;
    }

    $this->response = $response;
    // ...
}
```

Note: Most proxy users should probably consider using the Last Modified Header instead of Etags for performance and compatibility reasons.

The Last Modified Header

Cake\Http\Response::**withModified**(*$time*)

Also, under the HTTP cache validation model, you can set the `Last-Modified` header to indicate the date and time at which the resource was modified for the last time. Setting this header helps CakePHP tell caching clients whether the response was modified or not based on their cache.

To take advantage of this header, you must either call the `checkNotModified()` method manually or include the *Checking HTTP Cache* in your controller:

```
public function view()
{
    $article = $this->Articles->find()->first();
    $response = $this->response->withModified($article->modified);
    if ($response->checkNotModified($this->request)) {
        return $response;
```

(continues on next page)

(continued from previous page)

```
    }
    $this->response;
    // ...
}
```

The Vary Header

Cake\Http\Response::**withVary**(*$header*)

In some cases, you might want to serve different content using the same URL. This is often the case if you have a multilingual page or respond with different HTML depending on the browser. Under such circumstances you can use the `Vary` header:

```
$response = $this->response->withVary('User-Agent');
$response = $this->response->withVary('Accept-Encoding', 'User-Agent');
$response = $this->response->withVary('Accept-Language');
```

Sending Not-Modified Responses

Cake\Http\Response::**checkNotModified**(*Request $request*)

Compares the cache headers for the request object with the cache header from the response and determines whether it can still be considered fresh. If so, deletes the response content, and sends the *304 Not Modified* header:

```
// In a controller action.
if ($this->response->checkNotModified($this->request)) {
    return $this->response;
}
```

Setting Cookies

Cookies can be added to response using either an array or a *Cake\Http\Cookie\Cookie* object:

```
use Cake\Http\Cookie\Cookie;
use DateTime;

// Add a cookie
$this->response = $this->response->withCookie(Cookie::create(
    'remember_me',
    'yes',
    // All keys are optional
    [
        'expires' => new DateTime('+1 year'),
        'path' => '',
        'domain' => '',
        'secure' => false,
        'httponly' => false,
        'samesite' => null // Or one of CookieInterface::SAMESITE_* constants
    ]
));
```

See the *Creating Cookies* section for how to use the cookie object. You can use `withExpiredCookie()` to send an expired cookie in the response. This will make the browser remove its local cookie:

```
$this->response = $this->response->withExpiredCookie(new Cookie('remember_me'));
```

Setting Cross Origin Request Headers (CORS)

The `cors()` method is used to define HTTP Access Control[108] related headers with a fluent interface:

```
$this->response = $this->response->cors($this->request)
    ->allowOrigin(['*.cakephp.org'])
    ->allowMethods(['GET', 'POST'])
    ->allowHeaders(['X-CSRF-Token'])
    ->allowCredentials()
    ->exposeHeaders(['Link'])
    ->maxAge(300)
    ->build();
```

CORS related headers will only be applied to the response if the following criteria are met:

1. The request has an `Origin` header.
2. The request's `Origin` value matches one of the allowed Origin values.

Tip: CakePHP has no built-in CORS middleware because dealing with CORS requests is very application specific. We recommend you build your own `CORSMiddleware` if you need one and adjust the response object as desired.

Common Mistakes with Immutable Responses

Response objects offer a number of methods that treat responses as immutable objects. Immutable objects help prevent difficult to track accidental side-effects, and reduce mistakes caused by method calls caused by refactoring that change ordering. While they offer a number of benefits, immutable objects can take some getting used to. Any method that starts with `with` operates on the response in an immutable fashion, and will **always** return a **new** instance. Forgetting to retain the modified instance is the most frequent mistake people make when working with immutable objects:

```
$this->response->withHeader('X-CakePHP', 'yes!');
```

In the above code, the response will be lacking the `X-CakePHP` header, as the return value of the `withHeader()` method was not retained. To correct the above code you would write:

```
$this->response = $this->response->withHeader('X-CakePHP', 'yes!');
```

[108] https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS

Cookie Collections

class Cake\Http\Cookie**CookieCollection**

`CookieCollection` objects are accessible from the request and response objects. They let you interact with groups of cookies using immutable patterns, which allow the immutability of the request and response to be preserved.

Creating Cookies

class Cake\Http\Cookie**Cookie**

`Cookie` objects can be defined through constructor objects, or by using the fluent interface that follows immutable patterns:

```
use Cake\Http\Cookie\Cookie;

// All arguments in the constructor
$cookie = new Cookie(
    'remember_me', // name
    1, // value
    new DateTime('+1 year'), // expiration time, if applicable
    '/', // path, if applicable
    'example.com', // domain, if applicable
    false, // secure only?
    true // http only ?
);

// Using the builder methods
$cookie = (new Cookie('remember_me'))
    ->withValue('1')
    ->withExpiry(new DateTime('+1 year'))
    ->withPath('/')
    ->withDomain('example.com')
    ->withSecure(false)
    ->withHttpOnly(true);
```

Once you have created a cookie, you can add it to a new or existing `CookieCollection`:

```
use Cake\Http\Cookie\CookieCollection;

// Create a new collection
$cookies = new CookieCollection([$cookie]);

// Add to an existing collection
$cookies = $cookies->add($cookie);

// Remove a cookie by name
$cookies = $cookies->remove('remember_me');
```

Note: Remember that collections are immutable and adding cookies into, or removing cookies from a collection, creates a *new* collection object.

Cookie objects can be added to responses:

```
// Add one cookie
$response = $this->response->withCookie($cookie);

// Replace the entire cookie collection
$response = $this->response->withCookieCollection($cookies);
```

Cookies set to responses can be encrypted using the encrypted-cookie-middleware.

Reading Cookies

Once you have a `CookieCollection` instance, you can access the cookies it contains:

```
// Check if a cookie exists
$cookies->has('remember_me');

// Get the number of cookies in the collection
count($cookies);

// Get a cookie instance. Will throw an error if the cookie is not found
$cookie = $cookies->get('remember_me');

// Get a cookie or null
$cookie = $cookies->remember_me;

// Check if a cookie exists
$exists = isset($cookies->remember_me)
```

Once you have a `Cookie` object you can interact with it's state and modify it. Keep in mind that cookies are immutable, so you'll need to update the collection if you modify a cookie:

```
// Get the value
$value = $cookie->getValue()

// Access data inside a JSON value
$id = $cookie->read('User.id');

// Check state
$cookie->isHttpOnly();
$cookie->isSecure();
```

CHAPTER 10

Controllers

class Cake\Controller**Controller**

Controllers are the 'C' in MVC. After routing has been applied and the correct controller has been found, your controller's action is called. Your controller should handle interpreting the request data, making sure the correct models are called, and the right response or view is rendered. Controllers can be thought of as middle layer between the Model and View. You want to keep your controllers thin, and your models fat. This will help you reuse your code and makes your code easier to test.

Commonly, a controller is used to manage the logic around a single model. For example, if you were building a site for an online bakery, you might have a RecipesController managing your recipes and an IngredientsController managing your ingredients. However, it's also possible to have controllers work with more than one model. In CakePHP, a controller is named after the primary model it handles.

Your application's controllers extend the `AppController` class, which in turn extends the core *`Controller`* class. The `AppController` class can be defined in **src/Controller/AppController.php** and it should contain methods that are shared between all of your application's controllers.

Controllers provide a number of methods that handle requests. These are called *actions*. By default, each public method in a controller is an action, and is accessible from a URL. An action is responsible for interpreting the request and creating the response. Usually responses are in the form of a rendered view, but there are other ways to create responses as well.

The App Controller

As stated in the introduction, the `AppController` class is the parent class to all of your application's controllers. `AppController` itself extends the *`Cake\Controller\Controller`* class included in CakePHP. `AppController` is defined in **src/Controller/AppController.php** as follows:

```
namespace App\Controller;

use Cake\Controller\Controller;

class AppController extends Controller
{
}
```

Controller attributes and methods created in your `AppController` will be available in all controllers that extend it. Components (which you'll learn about later) are best used for code that is used in many (but not necessarily all) controllers.

You can use your `AppController` to load components that will be used in every controller in your application. CakePHP provides a `initialize()` method that is invoked at the end of a Controller's constructor for this kind of use:

```
namespace App\Controller;

use Cake\Controller\Controller;

class AppController extends Controller
{
    public function initialize(): void
    {
        // Always enable the FormProtection component.
        $this->loadComponent('FormProtection');
    }
}
```

Request Flow

When a request is made to a CakePHP application, CakePHP's `Cake\Routing\Router` and `Cake\Routing\Dispatcher` classes use *Connecting Routes* to find and create the correct controller instance. The request data is encapsulated in a request object. CakePHP puts all of the important request information into the `$this->request` property. See the section on *Request* for more information on the CakePHP request object.

Controller Actions

Controller actions are responsible for converting the request parameters into a response for the browser/user making the request. CakePHP uses conventions to automate this process and remove some boilerplate code you would otherwise need to write.

By convention, CakePHP renders a view with an inflected version of the action name. Returning to our online bakery example, our RecipesController might contain the `view()`, `share()`, and `search()` actions. The controller would be found in **src/Controller/RecipesController.php** and contain:

```
// src/Controller/RecipesController.php

class RecipesController extends AppController
{
    public function view($id)
    {
        // Action logic goes here.
    }

    public function share($customerId, $recipeId)
    {
        // Action logic goes here.
    }

    public function search($query)
    {
        // Action logic goes here.
    }
}
```

The template files for these actions would be **templates/Recipes/view.php**, **templates/Recipes/share.php**, and **templates/Recipes/search.php**. The conventional view file name is the lowercased and underscored version of the action name.

Controller actions generally use `Controller::set()` to create a context that `View` uses to render the view layer. Because of the conventions that CakePHP uses, you don't need to create and render the view manually. Instead, once a controller action has completed, CakePHP will handle rendering and delivering the View.

If for some reason you'd like to skip the default behavior, you can return a *Cake\Http\Response* object from the action with the fully created response.

In order for you to use a controller effectively in your own application, we'll cover some of the core attributes and methods provided by CakePHP's controllers.

Interacting with Views

Controllers interact with views in a number of ways. First, they are able to pass data to the views, using `Controller::set()`. You can also decide which view class to use, and which view file should be rendered from the controller.

Setting View Variables

Cake\Controller\Controller::**set**(*string $var*, *mixed $value*)

The `Controller::set()` method is the main way to send data from your controller to your view. Once you've used `Controller::set()`, the variable can be accessed in your view:

```
// First you pass data from the controller:

$this->set('color', 'pink');

// Then, in the view, you can utilize the data:
?>

You have selected <?= h($color) ?> icing for the cake.
```

The `Controller::set()` method also takes an associative array as its first parameter. This can often be a quick way to assign a set of information to the view:

```
$data = [
    'color' => 'pink',
    'type' => 'sugar',
    'base_price' => 23.95,
];

// Make $color, $type, and $base_price
// available to the view:

$this->set($data);
```

Keep in mind that view vars are shared among all parts rendered by your view. They will be available in all parts of the view: the template, the layout and all elements inside the former two.

Setting View Options

If you want to customize the view class, layout/template paths, helpers or the theme that will be used when rendering the view, you can use the `viewBuilder()` method to get a builder. This builder can be used to define properties of the view before it is created:

```
$this->viewBuilder()
    ->addHelper('MyCustom')
    ->setTheme('Modern')
    ->setClassName('Modern.Admin');
```

The above shows how you can load custom helpers, set the theme and use a custom view class.

Rendering a View

Cake\Controller\Controller::**render**(*string $view*, *string $layout*)

The `Controller::render()` method is automatically called at the end of each requested controller action. This method performs all the view logic (using the data you've submitted using the `Controller::set()` method), places the view inside its `View::$layout`, and serves it back to the end user.

The default view file used by render is determined by convention. If the `search()` action of the RecipesController is requested, the view file in **templates/Recipes/search.php** will be rendered:

```
namespace App\Controller;

class RecipesController extends AppController
{
// ...
    public function search()
    {
        // Render the view in templates/Recipes/search.php
        return $this->render();
    }
// ...
}
```

Although CakePHP will automatically call it after every action's logic (unless you've called `$this->disableAutoRender()`), you can use it to specify an alternate view file by specifying a view file name as first argument of `Controller::render()` method.

If `$view` starts with '/', it is assumed to be a view or element file relative to the **templates** folder. This allows direct rendering of elements, very useful in AJAX calls:

```
// Render the element in templates/element/ajaxreturn.php
$this->render('/element/ajaxreturn');
```

The second parameter `$layout` of `Controller::render()` allows you to specify the layout with which the view is rendered.

Rendering a Specific Template

In your controller, you may want to render a different view than the conventional one. You can do this by calling `Controller::render()` directly. Once you have called `Controller::render()`, CakePHP will not try to re-render the view:

```
namespace App\Controller;

class PostsController extends AppController
{
    public function my_action()
    {
        $this->render('custom_file');
    }
}
```

This would render **templates/Posts/custom_file.php** instead of **templates/Posts/my_action.php**.

You can also render views inside plugins using the following syntax: `$this->render('PluginName.PluginController/custom_file')`. For example:

```
namespace App\Controller;

class PostsController extends AppController
{
    public function myAction()
    {
        $this->render('Users.UserDetails/custom_file');
    }
}
```

This would render **plugins/Users/templates/UserDetails/custom_file.php**

Content Type Negotiation

`Cake\Controller\Controller::addViewClasses()`

Controllers can define a list of view classes they support. After the controller's action is complete CakePHP will use the view list to perform content-type negotiation with either *Routing File Extensions* or `Accept` headers. This enables your application to re-use the same controller action to render an HTML view or render a JSON or XML response. To define the list of supported view classes for a controller is done with the `addViewClasses()` method:

```
namespace App\Controller;

use Cake\View\JsonView;
use Cake\View\XmlView;

class PostsController extends AppController
{
    public function initialize(): void
    {
        parent::initialize();

        $this->addViewClasses([JsonView::class, XmlView::class]);
    }
}
```

The application's `View` class is automatically used as a fallback when no other view can be selected based on the request's `Accept` header or routing extension. If your application only supports content types for a specific actions, you can call `addClasses()` within your action too:

```
public function export(): void
{
    // Use a custom CSV view for data exports.
    $this->addViewClasses([CsvView::class]);

    // Rest of the action code
}
```

If within your controller actions you need to process the request or load data differently based on the content type you can use *Checking Request Conditions*:

```
// In a controller action

// Load additional data when preparing JSON responses
if ($this->request->is('json')) {
    $query->contain('Authors');
}
```

In case your app need more complex logic to decide which view classes to use then you can override the `Controller::viewClasses()` method and return an array of view classes as required.

Note: View classes must implement the static `contentType()` hook method to participate in content-type negotiation.

Content Type Negotiation Fallbacks

If no View can be matched with the request's content type preferences, CakePHP will use the base `View` class. If you want to require content-type negotiation, you can use the `NegotiationRequiredView` which sets a `406` status code:

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

    // Require Accept header negotiation or return a 406 response.
    $this->addViewClasses([JsonView::class, NegotiationRequiredView::class]);
}
```

You can use the `TYPE_MATCH_ALL` content type value to build your own fallback view logic:

```
namespace App\View;

use Cake\View\View;

class CustomFallbackView extends View
{
    public static function contentType(): string
    {
        return static::TYPE_MATCH_ALL;
    }

}
```

It is important to remember that match-all views are applied only *after* content-type negotiation is attempted.

Using AjaxView

In applications that use hypermedia or AJAX clients, you often need to render view contents without the wrapping layout. You can use the `AjaxView` that is bundled with the application skeleton:

```
// In a controller action, or in beforeRender.
if ($this->request->is('ajax')) {
    $this->viewBuilder()->setClassName('Ajax');
}
```

`AjaxView` will respond as `text/html` and use the `ajax` layout. Generally this layout is minimal or contains client specific markup. This replaces usage of `RequestHandlerComponent` automatically using the `AjaxView` in 4.x.

Redirecting to Other Pages

`Cake\Controller\Controller::`**`redirect`**(*string|array $url*, *integer $status*)

The `redirect()` method adds a `Location` header and sets the status code of a response and returns it. You should return the response created by `redirect()` to have CakePHP send the redirect instead of completing the controller action and rendering a view.

You can redirect using *routing array* values:

```
return $this->redirect([
    'controller' => 'Orders',
    'action' => 'confirm',
    $order->id,
    '?' => [
        'product' => 'pizza',
        'quantity' => 5
    ],
    '#' => 'top'
]);
```

Or using a relative or absolute URL:

```
return $this->redirect('/orders/confirm');
return $this->redirect('http://www.example.com');
```

Or to the referer page:

```
return $this->redirect($this->referer());
```

By using the second parameter you can define a status code for your redirect:

```
// Do a 301 (moved permanently)
return $this->redirect('/order/confirm', 301);

// Do a 303 (see other)
return $this->redirect('/order/confirm', 303);
```

See the *Using Redirects in Component Events* section for how to redirect out of a life-cycle handler.

Loading Additional Tables/Models

Cake\Controller\Controller::**fetchTable**(*string $alias*, *array $config = []*)

The `fetchTable()` method comes handy when you need to use an ORM table that is not the controller's default one:

```
// In a controller method.
$recentArticles = $this->fetchTable('Articles')->find('all',
        limit: 5,
        order: 'Articles.created DESC'
    )
    ->all();
```

Cake\Controller\Controller::**fetchModel**(*string|null $modelClass = null*, *string|null $modelType = null*)

The `fetchModel()` method is useful to load non ORM models or ORM tables that are not the controller's default:

```
// ModelAwareTrait need to be explicity added to your controler first for fetchModel()␣
↪to work.
use ModelAwareTrait;

// Get an ElasticSearch model
$articles = $this->fetchModel('Articles', 'Elastic');

// Get a webservices model
$github = $this->fetchModel('GitHub', 'Webservice');

// If you skip the 2nd argument it will by default try to load a ORM table.
$authors = $this->fetchModel('Authors');
```

New in version 4.5.0.

Paginating a Model

Cake\Controller\Controller::**paginate**()

This method is used for paginating results fetched by your models. You can specify page sizes, model find conditions and more. See the *pagination* section for more details on how to use `paginate()`.

The `$paginate` attribute gives you a way to customize how `paginate()` behaves:

```
class ArticlesController extends AppController
{
    protected array $paginate = [
        'Articles' => [
            'conditions' => ['published' => 1],
        ],
    ];
}
```

Configuring Components to Load

Cake\Controller\Controller::**loadComponent**(*$name*, *$config = []*)

In your Controller's `initialize()` method you can define any components you want loaded, and any configuration data for them:

```
public function initialize(): void
{
    parent::initialize();
    $this->loadComponent('Flash');
    $this->loadComponent('Comments', Configure::read('Comments'));
}
```

Request Life-cycle Callbacks

CakePHP controllers trigger several events/callbacks that you can use to insert logic around the request life-cycle:

Event List

- `Controller.initialize`
- `Controller.startup`
- `Controller.beforeRedirect`
- `Controller.beforeRender`
- `Controller.shutdown`

Controller Callback Methods

By default the following callback methods are connected to related events if the methods are implemented by your controllers

Cake\Controller\Controller::**beforeFilter**(*EventInterface $event*)

Called during the `Controller.initialize` event which occurs before every action in the controller. It's a handy place to check for an active session or inspect user permissions.

Note: The beforeFilter() method will be called for missing actions.

Returning a response from a `beforeFilter` method will not prevent other listeners of the same event from being called. You must explicitly *stop the event*.

Cake\Controller\Controller::**beforeRender**(*EventInterface $event*)

Called during the `Controller.beforeRender` event which occurs after controller action logic, but before the view is rendered. This callback is not used often, but may be needed if you are calling *`render()`* manually before the end of a given action.

`Cake\Controller\Controller::afterFilter`(*EventInterface $event*)

Called during the `Controller.shutdown` event which is triggered after every controller action, and after rendering is complete. This is the last controller method to run.

In addition to controller life-cycle callbacks, *Components* also provide a similar set of callbacks.

Remember to call `AppController`'s callbacks within child controller callbacks for best results:

```
//use Cake\Event\EventInterface;
public function beforeFilter(EventInterface $event)
{
    parent::beforeFilter($event);
}
```

Controller Middleware

`Cake\Controller\Controller::middleware`(*$middleware*, *array $options = []*)

Middleware can be defined globally, in a routing scope or within a controller. To define middleware for a specific controller use the `middleware()` method from your controller's `initialize()` method:

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

    $this->middleware(function ($request, $handler) {
        // Do middleware logic.

        // Make sure you return a response or call handle()
        return $handler->handle($request);
    });
}
```

Middleware defined by a controller will be called **before** `beforeFilter()` and action methods are called.

More on Controllers

The Pages Controller

CakePHP's official skeleton app ships with a default controller **PagesController.php**. This is a simple and optional controller for serving up static content. The home page you see after installation is generated using this controller and the view file **templates/Pages/home.php**. If you make the view file **templates/Pages/about_us.php** you can access it using the URL **http://example.com/pages/about_us**. You are free to modify the Pages Controller to meet your needs.

When you "bake" an app using Composer the Pages Controller is created in your **src/Controller/** folder.

Components

Components are packages of logic that are shared between controllers. CakePHP comes with a fantastic set of core components you can use to aid in various common tasks. You can also create your own components. If you find yourself wanting to copy and paste things between controllers, you should consider creating your own component to contain the functionality. Creating components keeps controller code clean and allows you to reuse code between different controllers.

For more information on the components included in CakePHP, check out the chapter for each component:

Flash

class Cake\Controller\Component**FlashComponent**(*ComponentCollection $collection*, *array $config = []*)

FlashComponent provides a way to set one-time notification messages to be displayed after processing a form or acknowledging data. CakePHP refers to these messages as "flash messages". FlashComponent writes flash messages to `$_SESSION`, to be rendered in a View using *FlashHelper*.

Setting Flash Messages

FlashComponent provides two ways to set flash messages: its `__call()` magic method and its `set()` method. To furnish your application with verbosity, FlashComponent's `__call()` magic method allows you use a method name that maps to an element located under the **templates/element/flash** directory. By convention, camelcased methods will map to the lowercased and underscored element name:

```
// Uses templates/element/flash/success.php
$this->Flash->success('This was successful');

// Uses templates/element/flash/great_success.php
$this->Flash->greatSuccess('This was greatly successful');
```

Alternatively, to set a plain-text message without rendering an element, you can use the `set()` method:

```
$this->Flash->set('This is a message');
```

Flash messages are appended to an array internally. Successive calls to `set()` or `__call()` with the same key will append the messages in the `$_SESSION`. If you want to overwrite existing messages when setting a flash message, set the `clear` option to `true` when configuring the Component.

FlashComponent's `__call()` and `set()` methods optionally take a second parameter, an array of options:

- `key` Defaults to 'flash'. The array key found under the `Flash` key in the session.
- `element` Defaults to `null`, but will automatically be set when using the `__call()` magic method. The element name to use for rendering.
- `params` An optional array of keys/values to make available as variables within an element.
- `clear` expects a `bool` and allows you to delete all messages in the current stack and start a new one.

An example of using these options:

```
// In your Controller
$this->Flash->success('The user has been saved', [
    'key' => 'positive',
    'clear' => true,
```

(continues on next page)

(continued from previous page)

```
    'params' => [
        'name' => $user->name,
        'email' => $user->email,
    ],
]);

// In your View
<?= $this->Flash->render('positive') ?>

<!-- In templates/element/flash/success.php -->
<div id="flash-<?= h($key) ?>" class="message-info success">
    <?= h($message) ?>: <?= h($params['name']) ?>, <?= h($params['email']) ?>.
</div>
```

Note that the parameter `element` will be always overridden while using `__call()`. In order to retrieve a specific element from a plugin, you should set the `plugin` parameter. For example:

```
// In your Controller
$this->Flash->warning('My message', ['plugin' => 'PluginName']);
```

The code above will use the **warning.php** element under **plugins/PluginName/templates/element/flash** for rendering the flash message.

Note: By default, CakePHP escapes the content in flash messages to prevent cross site scripting. User data in your flash messages will be HTML encoded and safe to be printed. If you want to include HTML in your flash messages, you need to pass the `escape` option and adjust your flash message templates to allow disabling escaping when the escape option is passed.

HTML in Flash Messages

It is possible to output HTML in flash messages by using the `'escape'` option key:

```
$this->Flash->info(sprintf('<b>%s</b> %s', h($highlight), h($message)), ['escape' =>␣
↪false]);
```

Make sure that you escape the input manually, then. In the above example `$highlight` and `$message` are non-HTML input and therefore escaped.

For more information about rendering your flash messages, please refer to the *FlashHelper* section.

FormProtection

class FormProtection(*ComponentCollection $collection*, *array $config = []*)

The FormProtection Component provides protection against form data tampering.

Like all components it is configured through several configurable parameters. All of these properties can be set directly or through setter methods of the same name in your controller's `initialize()` or `beforeFilter()` methods.

If you are using other components that process form data in their `startup()` callbacks, be sure to place FormProtection Component before those components in your `initialize()` method.

Note: When using the FormProtection Component you **must** use the FormHelper to create your forms. In addition, you must **not** override any of the fields' "name" attributes. The FormProtection Component looks for certain indicators that are created and managed by the FormHelper (especially those created in *create()* and *end()*). Dynamically altering the fields that are submitted in a POST request, such as disabling, deleting or creating new fields via JavaScript, is likely to cause the form token validation to fail.

Form tampering prevention

By default the `FormProtectionComponent` prevents users from tampering with forms in specific ways. It will prevent the following things:

- Form's action (URL) cannot be modified.
- Unknown fields cannot be added to the form.
- Fields cannot be removed from the form.
- Values in hidden inputs cannot be modified.

Preventing these types of tampering is accomplished by working with the `FormHelper` and tracking which fields are in a form. The values for hidden fields are tracked as well. All of this data is combined and turned into a hash and hidden token fields are automatically be inserted into forms. When a form is submitted, the `FormProtectionComponent` will use the POST data to build the same structure and compare the hash.

Note: The FormProtectionComponent will **not** prevent select options from being added/changed. Nor will it prevent radio options from being added/changed.

Usage

Configuring the form protection component is generally done in the controller's `initialize()` or `beforeFilter()` callbacks

Available options are:

validate
: Set to `false` to completely skip the validation of POST requests, essentially turning off form validation.

unlockedFields
: Set to a list of form fields to exclude from POST validation. Fields can be unlocked either in the Component, or with `FormHelper::unlockField()`. Fields that have been unlocked are not required to be part of the POST and hidden unlocked fields do not have their values checked.

unlockedActions
: Actions to exclude from POST validation checks.

validationFailureCallback
: Callback to call in case of validation failure. Must be a valid Closure. Unset by default in which case exception is thrown on validation failure.

Disabling form tampering checks

```
namespace App\Controller;

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

class WidgetsController extends AppController
{
    public function initialize(): void
    {
        parent::initialize();

        $this->loadComponent('FormProtection');
    }

    public function beforeFilter(EventInterface $event)
    {
        parent::beforeFilter($event);

        if ($this->request->getParam('prefix') === 'Admin') {
            $this->FormProtection->setConfig('validate', false);
        }
    }
}
```

The above example would disable form tampering prevention for admin prefixed routes.

Disabling form tampering for specific actions

There may be cases where you want to disable form tampering prevention for an action (ex. AJAX requests). You may "unlock" these actions by listing them in `$this->FormProtection->setConfig('unlockedActions', ['edit']);` in your `beforeFilter()`:

```
namespace App\Controller;

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

class WidgetController extends AppController
{
    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('FormProtection');
    }

    public function beforeFilter(EventInterface $event)
    {
        parent::beforeFilter($event);

        $this->FormProtection->setConfig('unlockedActions', ['edit']);
```

(continues on next page)

(continued from previous page)

```
    }
}
```

This example would disable all security checks for the edit action.

Handling validation failure through callbacks

If form protection validation fails it will result in a 400 error by default. You can configure this behavior by setting the `validationFailureCallback` configuration option to a callback function in the controller.

By configuring a callback method you can customize how the failure handling process works:

```
public function beforeFilter(EventInterface $event)
{
    parent::beforeFilter($event);

    $this->FormProtection->setConfig(
        'validationFailureCallback',
        function (BadRequestException $exception) {
            // You can either return a response instance or throw the exception
            // received as argument.
        }
    );
}
```

Checking HTTP Cache

class CheckHttpCacheComponent(*ComponentCollection $collection*, *array $config = []*)

The HTTP cache validation model is one of the processes used for cache gateways, also known as reverse proxies, to determine if they can serve a stored copy of a response to the client. Under this model, you mostly save bandwidth, but when used correctly you can also save some CPU processing, reducing response times:

```
// in a Controller
public function initialize(): void
{
    parent::initialize();

    $this->addComponent('CheckHttpCache');
}
```

Enabling the `CheckHttpCacheComponent` in your controller automatically activates a `beforeRender` check. This check compares caching headers set in the response object to the caching headers sent in the request to determine whether the response was not modified since the last time the client asked for it. The following request headers are used:

- `If-None-Match` is compared with the response's `Etag` header.
- `If-Modified-Since` is compared with the response's `Last-Modified` header.

If response headers match the request header criteria, then view rendering is skipped. This saves your application generating a view, saving bandwidth and time. When response headers match, an empty response is returned with a `304 Not Modified` status code.

Configuring Components

Many of the core components require configuration. One example would be the *FormProtection*. Configuration for these components, and for components in general, is usually done via `loadComponent()` in your Controller's `initialize()` method or via the `$components` array:

```
class PostsController extends AppController
{
    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('FormProtection', [
            'unlockedActions' => ['index'],
        ]);
        $this->loadComponent('Flash');
    }
}
```

You can configure components at runtime using the `setConfig()` method. Often, this is done in your controller's `beforeFilter()` method. The above could also be expressed as:

```
public function beforeFilter(EventInterface $event)
{
    $this->FormProtection->setConfig('unlockedActions', ['index']);
}
```

Like helpers, components implement `getConfig()` and `setConfig()` methods to read and write configuration data:

```
// Read config data.
$this->FormProtection->getConfig('unlockedActions');

// Set config
$this->Flash->setConfig('key', 'myFlash');
```

As with helpers, components will automatically merge their `$_defaultConfig` property with constructor configuration to create the `$_config` property which is accessible with `getConfig()` and `setConfig()`.

Aliasing Components

One common setting to use is the `className` option, which allows you to alias components. This feature is useful when you want to replace `$this->Flash` or another common Component reference with a custom implementation:

```
// src/Controller/PostsController.php
class PostsController extends AppController
{
    public function initialize(): void
    {
        $this->loadComponent('Flash', [
            'className' => 'MyFlash',
        ]);
    }
}
```

(continues on next page)

(continued from previous page)

```
// src/Controller/Component/MyFlashComponent.php
use Cake\Controller\Component\FlashComponent;

class MyFlashComponent extends FlashComponent
{
    // Add your code to override the core FlashComponent
}
```

The above would *alias* `MyFlashComponent` to `$this->Flash` in your controllers.

Note: Aliasing a component replaces that instance anywhere that component is used, including inside other Components.

Loading Components on the Fly

You might not need all of your components available on every controller action. In situations like this you can load a component at runtime using the `loadComponent()` method in your controller:

```
// In a controller action
$this->loadComponent('OneTimer');
$time = $this->OneTimer->getTime();
```

Note: Keep in mind that components loaded on the fly will not have missed callbacks called. If you rely on the `beforeFilter` or `startup` callbacks being called, you may need to call them manually depending on when you load your component.

Using Components

Once you've included some components in your controller, using them is pretty simple. Each component you use is exposed as a property on your controller. If you had loaded up the *Cake\Controller\Component\FlashComponent* in your controller, you could access it like so:

```
class PostsController extends AppController
{
    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('Flash');
    }

    public function delete()
    {
        if ($this->Post->delete($this->request->getData('Post.id')) {
            $this->Flash->success('Post deleted.');
            return $this->redirect(['action' => 'index']);
        }
    }
```

Note: Since both Models and Components are added to Controllers as properties they share the same 'namespace'. Be sure to not give a component and a model the same name.

Warning: Component methods **don't** have access to /development/dependency-injection like Controller actions have. Use a service class inside your controller actions instead of a component if you need this functionality.

Creating a Component

Suppose our application needs to perform a complex mathematical operation in many different parts of the application. We could create a component to house this shared logic for use in many different controllers.

The first step is to create a new component file and class. Create the file in **src/Controller/Component/MathComponent.php**. The basic structure for the component would look something like this:

```
namespace App\Controller\Component;

use Cake\Controller\Component;

class MathComponent extends Component
{
    public function doComplexOperation($amount1, $amount2)
    {
        return $amount1 + $amount2;
    }
}
```

Note: All components must extend `Cake\Controller\Component`. Failing to do this will trigger an exception.

Including your Component in your Controllers

Once our component is finished, we can use it in the application's controllers by loading it during the controller's `initialize()` method. Once loaded, the controller will be given a new attribute named after the component, through which we can access an instance of it:

```
// In a controller
// Make the new component available at $this->Math,
// as well as the standard $this->Flash
public function initialize(): void
{
    parent::initialize();
    $this->loadComponent('Math');
    $this->loadComponent('Flash');
}
```

When including Components in a Controller you can also declare a set of parameters that will be passed on to the Component's constructor. These parameters can then be handled by the Component:

```
// In your controller.
public function initialize(): void
{
    parent::initialize();
    $this->loadComponent('Math', [
        'precision' => 2,
        'randomGenerator' => 'srand',
    ]);
    $this->loadComponent('Flash');
}
```

The above would pass the array containing precision and randomGenerator to `MathComponent::initialize()` in the `$config` parameter.

Using Other Components in your Component

Sometimes one of your components may need to use another component. You can load other components by adding them to the *$components* property:

```
// src/Controller/Component/CustomComponent.php
namespace App\Controller\Component;

use Cake\Controller\Component;

class CustomComponent extends Component
{
    // The other component your component uses
    protected array $components = ['Existing'];

    // Execute any other additional setup for your component.
    public function initialize(array $config): void
    {
        $this->Existing->foo();
    }

    public function bar()
    {
        // ...
    }
}

// src/Controller/Component/ExistingComponent.php
namespace App\Controller\Component;

use Cake\Controller\Component;

class ExistingComponent extends Component
{
    public function foo()
    {
        // ...
    }
```

(continues on next page)

(continued from previous page)

```
}
```

Note: In contrast to a component included in a controller no callbacks will be triggered on a component's component.

Accessing a Component's Controller

From within a Component you can access the current controller through the registry:

```
$controller = $this->getController();
```

Component Callbacks

Components also offer a few request life-cycle callbacks that allow them to augment the request cycle.

`beforeFilter`(*EventInterface $event*)
: Is called before the controller's beforeFilter() method, but *after* the controller's initialize() method.

`startup`(*EventInterface $event*)
: Is called after the controller's beforeFilter() method but before the controller executes the current action handler.

`beforeRender`(*EventInterface $event*)
: Is called after the controller executes the requested action's logic, but before the controller renders views and layout.

`afterFilter`(*EventInterface $event*)
: Is called during the `Controller.shutdown` event, before output is sent to the browser.

`beforeRedirect`(*EventInterface $event*, *$url*, *Response $response*)
: Is invoked when the controller's redirect method is called but before any further action. If this method returns `false` the controller will not continue on to redirect the request. The $url, and $response parameters allow you to inspect and modify the location or any other headers in the response.

Using Redirects in Component Events

To redirect from within a component callback method you can use the following:

```
public function beforeFilter(EventInterface $event)
{
    $event->stopPropagation();
    return $this->getController()->redirect('/');
}
```

By stopping the event you let CakePHP know that you don't want any other component callbacks to run, and that the controller should not handle the action any further. As of 4.1.0 you can raise a `RedirectException` to signal a redirect:

```
use Cake\Http\Exception\RedirectException;
use Cake\Routing\Router;
```

(continues on next page)

(continued from previous page)

```
public function beforeFilter(EventInterface $event)
{
    throw new RedirectException(Router::url('/'))
}
```

Raising an exception will halt all other event listeners and create a new response that doesn't retain or inherit any of the current response's headers. When raising a `RedirectException` you can include additional headers:

```
throw new RedirectException(Router::url('/'), 302, [
    'Header-Key' => 'value',
]);
```

CHAPTER 11

Views

class Cake\View**View**

Views are the **V** in MVC. Views are responsible for generating the specific output required for the request. Often this is in the form of HTML, XML, or JSON, but streaming files and creating PDFs that users can download are also responsibilities of the View Layer.

CakePHP comes with a few built-in View classes for handling the most common rendering scenarios:

- To create XML or JSON webservices you can use the *JSON and XML views*.
- To serve protected files, or dynamically generated files, you can use *Sending Files*.
- To create multiple themed views, you can use *Themes*.

The App View

`AppView` is your application's default View class. `AppView` itself extends the `Cake\View\View` class included in CakePHP and is defined in **src/View/AppView.php** as follows:

```
<?php
namespace App\View;

use Cake\View\View;

class AppView extends View
{
}
```

You can use your `AppView` to load helpers that will be used for every view rendered in your application. CakePHP provides an `initialize()` method that is invoked at the end of a View's constructor for this kind of use:

```php
<?php
namespace App\View;

use Cake\View\View;

class AppView extends View
{
    public function initialize(): void
    {
        // Always enable the MyUtils Helper
        $this->addHelper('MyUtils');
    }
}
```

View Templates

The view layer of CakePHP is how you speak to your users. Most of the time your views will be rendering HTML/XHTML documents to browsers, but you might also need to reply to a remote application via JSON, or output a CSV file for a user.

CakePHP template files are regular PHP files and utilize the alternative PHP syntax[109] for control structures and output. These files contain the logic necessary to prepare the data received from the controller into a presentation format that is ready for your audience.

Alternative Echos

Echo, or print a variable in your template:

```php
<?php echo $variable; ?>
```

Using Short Tag support:

```php
<?= $variable ?>
```

Alternative Control Structures

Control structures, like `if`, `for`, `foreach`, `switch`, and `while` can be written in a simplified format. Notice that there are no braces. Instead, the end brace for the `foreach` is replaced with `endforeach`. Each of the control structures listed above has a similar closing syntax: `endif`, `endfor`, `endforeach`, and `endwhile`. Also notice that instead of using a `semicolon` after each structure (except the last one), there is a `colon`.

The following is an example using `foreach`:

```php
<ul>
<?php foreach ($todo as $item): ?>
  <li><?= $item ?></li>
<?php endforeach; ?>
</ul>
```

[109] https://php.net/manual/en/control-structures.alternative-syntax.php

Another example, using if/elseif/else. Notice the colons:

```
<?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; ?>
```

If you'd prefer to use a templating language like Twig[110], checkout the CakePHP Twig Plugin[111]

Template files are stored in **templates/**, in a folder named after the controller that uses the files, and named after the action it corresponds to. For example, the view file for the `Products` controller's `view()` action, would normally be found in **templates/Products/view.php**.

The view layer in CakePHP can be made up of a number of different parts. Each part has different uses, and will be covered in this chapter:

- **templates**: Templates are the part of the page that is unique to the action being run. They form the meat of your application's response.
- **elements**: small, reusable bits of view code. Elements are usually rendered inside views.
- **layouts**: template files that contain presentational code that wraps many interfaces in your application. Most views are rendered inside a layout.
- **helpers**: these classes encapsulate view logic that is needed in many places in the view layer. Among other things, helpers in CakePHP can help you build forms, build AJAX functionality, paginate model data, or serve RSS feeds.
- **cells**: these classes provide miniature controller-like features for creating self contained UI components. See the *View Cells* documentation for more information.

View Variables

Any variables you set in your controller with `set()` will be available in both the view and the layout your action renders. In addition, any set variables will also be available in any element. If you need to pass additional variables from the view to the layout you can either call `set()` in the view template, or use *View Blocks*.

You should remember to **always** escape any user data before outputting it as CakePHP does not automatically escape output. You can escape user content with the `h()` function:

```
<?= h($user->bio); ?>
```

[110] https://twig.symfony.com
[111] https://github.com/cakephp/twig-view

Setting View Variables

Cake\View\View::**set**(*string $var*, *mixed $value*)

Views have a `set()` method that is analogous to the `set()` found in Controller objects. Using set() from your view file will add the variables to the layout and elements that will be rendered later. See *Setting View Variables* for more information on using `set()`.

In your view file you can do:

```
$this->set('activeMenuButton', 'posts');
```

Then, in your layout, the `$activeMenuButton` variable will be available and contain the value 'posts'.

Extending Views

View extending allows you to wrap one view in another. Combining this with *view blocks* gives you a powerful way to keep your views *DRY*. For example, your application has a sidebar that needs to change depending on the specific view being rendered. By extending a common view file, you can avoid repeating the common markup for your sidebar, and only define the parts that change:

```
<!-- templates/Common/view.php -->
<h1><?= h($this->fetch('title')) ?></h1>
<?= $this->fetch('content') ?>

<div class="actions">
    <h3>Related actions</h3>
    <ul>
    <?= $this->fetch('sidebar') ?>
    </ul>
</div>
```

The above view file could be used as a parent view. It expects that the view extending it will define the `sidebar` and `title` blocks. The `content` block is a special block that CakePHP creates. It will contain all the uncaptured content from the extending view. Assuming our view file has a `$post` variable with the data about our post, the view could look like:

```
<!-- templates/Posts/view.php -->
<?php
$this->extend('/Common/view');

$this->assign('title', $post->title);

$this->start('sidebar');
?>
<li>
<?php
echo $this->Html->link('edit', [
    'action' => 'edit',
    $post->id,
]);
?>
</li>
<?php $this->end(); ?>
```

(continues on next page)

(continued from previous page)

```
// The remaining content will be available as the 'content' block
// In the parent view.
<?= h($post->body) ?>
```

The post view above shows how you can extend a view, and populate a set of blocks. Any content not already in a defined block will be captured and put into a special block named `content`. When a view contains a call to `extend()`, execution continues to the bottom of the current view file. Once it is complete, the extended view will be rendered. Calling `extend()` more than once in a view file will override the parent view that will be processed next:

```
$this->extend('/Common/view');
$this->extend('/Common/index');
```

The above will result in **/Common/index.php** being rendered as the parent view to the current view.

You can nest extended views as many times as necessary. Each view can extend another view if desired. Each parent view will get the previous view's content as the `content` block.

Note: You should avoid using `content` as a block name in your application. CakePHP uses this for uncaptured content in extended views.

Extending Layouts

Just like views, layouts can also be extended. Like views, you use `extend()` to extend layouts. Layout extensions can update or replace blocks, and update or replace the content rendered by the child layout. For example if we wanted to wrap a block with additional markup you could do:

```
// Our layout extends the application layout.
$this->extend('application');
$this->prepend('content', '<main class="nosidebar">');
$this->append('content', '</main>');

// Output more markup.

// Remember to echo the contents of the previous layout.
echo $this->fetch('content');
```

Using View Blocks

View blocks provide a flexible API that allows you to define slots or blocks in your views/layouts that will be defined elsewhere. For example, blocks are ideal for implementing things such as sidebars, or regions to load assets at the bottom/top of the layout. Blocks can be defined in two ways: either as a capturing block, or by direct assignment. The `start()`, `append()`, `prepend()`, `assign()`, `fetch()`, and `end()` methods allow you to work with capturing blocks:

```
// Create the sidebar block.
$this->start('sidebar');
echo $this->element('sidebar/recent_topics');
echo $this->element('sidebar/recent_comments');
```

(continues on next page)

(continued from previous page)

```
$this->end();

// Append into the sidebar later on.
$this->start('sidebar');
echo $this->fetch('sidebar');
echo $this->element('sidebar/popular_topics');
$this->end();
```

You can also append into a block using `append()`:

```
$this->append('sidebar');
echo $this->element('sidebar/popular_topics');
$this->end();

// The same as the above.
$this->append('sidebar', $this->element('sidebar/popular_topics'));
```

If you need to clear or overwrite a block there are a couple of alternatives. The `reset()` method will clear or overwrite a block at any time. The `assign()` method with an empty content string can also be used to clear the specified block.:

```
// Clear the previous content from the sidebar block.
$this->reset('sidebar');

// Assigning an empty string will also clear the sidebar block.
$this->assign('sidebar', '');
```

Assigning a block's content is often useful when you want to convert a view variable into a block. For example, you may want to use a block for the page title, and sometimes assign the title as a view variable in the controller:

```
// In view file or layout above $this->fetch('title')
$this->assign('title', $title);
```

The `prepend()` method allows you to prepend content to an existing block:

```
// Prepend to sidebar
$this->prepend('sidebar', 'this content goes on top of sidebar');
```

Displaying Blocks

You can display blocks using the `fetch()` method. `fetch()` will output a block, returning '' if a block does not exist:

```
<?= $this->fetch('sidebar') ?>
```

You can also use fetch to conditionally show content that should surround a block should it exist. This is helpful in layouts, or extended views where you want to conditionally show headings or other markup:

```
// In templates/layout/default.php
<?php if ($this->fetch('menu')): ?>
<div class="menu">
    <h3>Menu options</h3>
    <?= $this->fetch('menu') ?>
```

(continues on next page)

(continued from previous page)

```
</div>
<?php endif; ?>
```

You can also provide a default value for a block if it does not exist. This allows you to add placeholder content when a block does not exist. You can provide a default value using the second argument:

```
<div class="shopping-cart">
    <h3>Your Cart</h3>
    <?= $this->fetch('cart', 'Your cart is empty') ?>
</div>
```

Using Blocks for Script and CSS Files

The `HtmlHelper` ties into view blocks, and its `script()`, `css()`, and `meta()` methods each update a block with the same name when used with the `block = true` option:

```
<?php
// In your view file
$this->Html->script('carousel', ['block' => true]);
$this->Html->css('carousel', ['block' => true]);
?>

// In your layout file.
<!DOCTYPE html>
<html lang="en">
    <head>
    <title><?= h($this->fetch('title')) ?></title>
    <?= $this->fetch('script') ?>
    <?= $this->fetch('css') ?>
    </head>
    // Rest of the layout follows
```

The *Cake\View\Helper\HtmlHelper* also allows you to control which block the scripts and CSS go to:

```
// In your view
$this->Html->script('carousel', ['block' => 'scriptBottom']);

// In your layout
<?= $this->fetch('scriptBottom') ?>
```

Layouts

A layout contains presentation code that wraps around a view. Anything you want to see in all of your views should be placed in a layout.

CakePHP's default layout is located at **templates/layout/default.php**. If you want to change the overall look of your application, then this is the right place to start, because controller-rendered view code is placed inside of the default layout when the page is rendered.

Other layout files should be placed in **templates/layout**. When you create a layout, you need to tell CakePHP where to place the output of your views. To do so, make sure your layout includes a place for `$this->fetch('content')`

Here's an example of what a default layout might look like:

```
<!DOCTYPE html>
<html lang="en">
<head>
<title><?= h($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>

<!-- If you'd like some sort of menu to
show up on all of your views, include it here -->
<div id="header">
    <div id="menu">...</div>
</div>

<!-- Here's where I want my views to be displayed -->
<?= $this->fetch('content') ?>

<!-- Add a footer to each displayed page -->
<div id="footer">...</div>

</body>
</html>
```

The `script`, `css` and `meta` blocks contain any content defined in the views using the built-in HTML helper. Useful for including JavaScript and CSS files from views.

Note: When using `HtmlHelper::css()` or `HtmlHelper::script()` in template files, specify `'block' => true` to place the HTML source in a block with the same name. (See API for more details on usage).

The `content` block contains the contents of the rendered view.

You can set the `title` block content from inside your view file:

```
$this->assign('title', 'View Active Users');
```

Empty values for the `title` block will be automatically replaced with a representation of the current template path, such as `'Admin/Articles'`.

You can create as many layouts as you wish: just place them in the **templates/layout** directory, and switch between them inside of your controller actions using the controller or view's `$layout` property:

```
// From a controller
public function view()
{
    // Set the layout.
    $this->viewBuilder()->setLayout('admin');
```

(continues on next page)

(continued from previous page)

```
}

// From a view file
$this->layout = 'loggedin';
```

For example, if a section of my site included a smaller ad banner space, I might create a new layout with the smaller advertising space and specify it as the layout for all controllers' actions using something like:

```
namespace App\Controller;

class UsersController extends AppController
{
    public function viewActive()
    {
        $this->set('title', 'View Active Users');
        $this->viewBuilder()->setLayout('default_small_ad');
    }

    public function viewImage()
    {
        $this->viewBuilder()->setLayout('image');

        // Output user image
    }
}
```

Besides a default layout CakePHP's official skeleton app also has an 'ajax' layout. The Ajax layout is handy for crafting AJAX responses - it's an empty layout. (Most AJAX calls only require a bit of markup in return, rather than a fully-rendered interface.)

The skeleton app also has a default layout to help generate RSS.

Using Layouts from Plugins

If you want to use a layout that exists in a plugin, you can use *plugin syntax*. For example, to use the contact layout from the Contacts plugin:

```
namespace App\Controller;

class UsersController extends AppController
{
    public function viewActive()
    {
        $this->viewBuilder()->setLayout('Contacts.contact');
    }
}
```

Elements

Cake\View\View::**element**(*string $elementPath*, *array $data*, *array $options = []*)

Many applications have small blocks of presentation code that need to be repeated from page to page, sometimes in different places in the layout. CakePHP can help you repeat parts of your website that need to be reused. These reusable parts are called Elements. Ads, help boxes, navigational controls, extra menus, login forms, and callouts are often implemented in CakePHP as elements. An element is basically a mini-view that can be included in other views, in layouts, and even within other elements. Elements can be used to make a view more readable, placing the rendering of repeating elements in its own file. They can also help you re-use content fragments in your application.

Elements live in the **templates/element/** folder, and have the .php filename extension. They are output using the element method of the view:

```
echo $this->element('helpbox');
```

Passing Variables into an Element

You can pass data to an element through the element's second argument:

```
echo $this->element('helpbox', [
    'helptext' => 'Oh, this text is very helpful.',
]);
```

Inside the element file, all the passed variables are available as members of the parameter array (in the same way that `Controller::set()` in the controller works with template files). In the above example, the **templates/element/helpbox.php** file can use the `$helptext` variable:

```
// Inside templates/element/helpbox.php
echo $helptext; // Outputs `Oh, this text is very helpful.`
```

Keep in mind that in those view vars are merged with the view vars from the view itself. So all view vars set using `Controller::set()` in the controller and `View::set()` in the view itself are also available inside the element.

The `View::element()` method also supports options for the element. The options supported are 'cache' and 'callbacks'. An example:

```
echo $this->element('helpbox', [
        'helptext' => "This is passed to the element as $helptext",
        'foobar' => "This is passed to the element as $foobar",
    ],
    [
        // uses the `long_view` cache configuration
        'cache' => 'long_view',
        // set to true to have before/afterRender called for the element
        'callbacks' => true,
    ]
);
```

Element caching is facilitated through the `Cache` class. You can configure elements to be stored in any Cache configuration you've set up. This gives you a great amount of flexibility to decide where and for how long elements are stored. To cache different versions of the same element in an application, provide a unique cache key value using the following format:

```
$this->element('helpbox', [], [
        'cache' => ['config' => 'short', 'key' => 'unique value'],
    ]
);
```

If you need more logic in your element, such as dynamic data from a datasource, consider using a View Cell instead of an element. Find out more *about View Cells*.

Caching Elements

You can take advantage of CakePHP view caching if you supply a cache parameter. If set to `true`, it will cache the element in the 'default' Cache configuration. Otherwise, you can set which cache configuration should be used. See *Caching* for more information on configuring `Cache`. A simple example of caching an element would be:

```
echo $this->element('helpbox', [], ['cache' => true]);
```

If you render the same element more than once in a view and have caching enabled, be sure to set the 'key' parameter to a different name each time. This will prevent each successive call from overwriting the previous `element()` call's cached result. For example:

```
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']]
);
```

The above will ensure that both element results are cached separately. If you want all element caching to use the same cache configuration, you can avoid some repetition by setting `View::$elementCache` to the cache configuration you want to use. CakePHP will use this configuration when none is given.

Requesting Elements from a Plugin

If you are using a plugin and wish to use elements from within the plugin, just use the familiar *plugin syntax*. If the view is being rendered for a plugin controller/action, the plugin name will automatically be prefixed onto all elements used, unless another plugin name is present. If the element doesn't exist in the plugin, it will look in the main APP folder:

```
echo $this->element('Contacts.helpbox');
```

If your view is a part of a plugin, you can omit the plugin name. For example, if you are in the `ContactsController` of the Contacts plugin, the following:

```
echo $this->element('helpbox');
// and
echo $this->element('Contacts.helpbox');
```

are equivalent and will result in the same element being rendered.

For elements inside subfolder of a plugin (for example, **plugins/Contacts/Template/element/sidebar/helpbox.php**), use the following:

```
echo $this->element('Contacts.sidebar/helpbox');
```

Routing prefix and Elements

If you have a Routing prefix configured, the Element path resolution can switch to a prefix location, as Layouts and action View do. Assuming you have a prefix "Admin" configured and you call:

```
echo $this->element('my_element');
```

The element first be looked for in **templates/Admin/element/**. If such a file does not exist, it will be looked for in the default location.

Caching Sections of Your View

`Cake\View\View::cache`(*callable $block*, *array $options = []*)

Sometimes generating a section of your view output can be expensive because of rendered *View Cells* or expensive helper operations. To help make your application run faster CakePHP provides a way to cache view sections:

```
// Assuming some local variables
echo $this->cache(function () use ($user, $article) {
    echo $this->cell('UserProfile', [$user]);
    echo $this->cell('ArticleFull', [$article]);
}, ['key' => 'my_view_key']);
```

By default cached view content will go into the `View::$elementCache` cache config, but you can use the `config` option to change this.

View Events

Like Controller, view trigger several events/callbacks that you can use to insert logic around the rendering life-cycle:

Event List

- `View.beforeRender`
- `View.beforeRenderFile`
- `View.afterRenderFile`
- `View.afterRender`
- `View.beforeLayout`
- `View.afterLayout`

You can attach application *event listeners* to these events or use *Helper Callbacks*.

Creating Your Own View Classes

You may need to create custom view classes to enable new types of data views, or add additional custom view-rendering logic to your application. Like most components of CakePHP, view classes have a few conventions:

- View class files should be put in **src/View**. For example: **src/View/PdfView.php**
- View classes should be suffixed with `View`. For example: `PdfView`.
- When referencing view class names you should omit the `View` suffix. For example: `$this->viewBuilder()->setClassName('Pdf');`.

You'll also want to extend `View` to ensure things work correctly:

```
// In src/View/PdfView.php
namespace App\View;

use Cake\View\View;

class PdfView extends View
{
    public function render($view = null, $layout = null)
    {
        // Custom logic here.
    }
}
```

Replacing the render method lets you take full control over how your content is rendered.

More About Views

View Cells

View cells are small mini-controllers that can invoke view logic and render out templates. The idea of cells is borrowed from cells in Ruby[112], where they fulfill a similar role and purpose.

When to use Cells

Cells are ideal for building reusable page components that require interaction with models, view logic, and rendering logic. A simple example would be the cart in an online store, or a data-driven navigation menu in a CMS.

[112] https://github.com/trailblazer/cells

Creating a Cell

To create a cell, define a class in **src/View/Cell** and a template in **templates/cell/**. In this example, we'll be making a cell to display the number of messages in a user's notification inbox. First, create the class file. Its contents should look like:

```
namespace App\View\Cell;

use Cake\View\Cell;

class InboxCell extends Cell
{
    public function display()
    {
    }
}
```

Save this file into **src/View/Cell/InboxCell.php**. As you can see, like other classes in CakePHP, Cells have a few conventions:

- Cells live in the `App\View\Cell` namespace. If you are making a cell in a plugin, the namespace would be `PluginName\View\Cell`.
- Class names should end in Cell.
- Classes should inherit from `Cake\View\Cell`.

We added an empty `display()` method to our cell; this is the conventional default method when rendering a cell. We'll cover how to use other methods later in the docs. Now, create the file **templates/cell/Inbox/display.php**. This will be our template for our new cell.

You can generate this stub code quickly using `bake`:

```
bin/cake bake cell Inbox
```

Would generate the code we created above.

Implementing the Cell

Assume that we are working on an application that allows users to send messages to each other. We have a `Messages` model, and we want to show the count of unread messages without having to pollute AppController. This is a perfect use case for a cell. In the class we just made, add the following:

```
namespace App\View\Cell;

use Cake\View\Cell;

class InboxCell extends Cell
{
    public function display()
    {
        $unread = $this->fetchTable('Messages')->find('unread');
        $this->set('unread_count', $unread->count());
    }
}
```

Because Cells use the `LocatorAwareTrait` and `ViewVarsTrait`, they behave very much like a controller would. We can use the `fetchTable()` and `set()` methods just like we would in a controller. In our template file, add the following:

```
<!-- templates/cell/Inbox/display.php -->
<div class="notification-icon">
    You have <?= $unread_count ?> unread messages.
</div>
```

Note: Cell templates have an isolated scope that does not share the same View instance as the one used to render template and layout for the current controller action or other cells. Hence they are unaware of any helper calls made or blocks set in the action's template / layout and vice versa.

Loading Cells

Cells can be loaded from views using the `cell()` method and works the same in both contexts:

```
// Load an application cell
$cell = $this->cell('Inbox');

// Load a plugin cell
$cell = $this->cell('Messaging.Inbox');
```

The above will load the named cell class and execute the `display()` method. You can execute other methods using the following:

```
// Run the expanded() method on the Inbox cell
$cell = $this->cell('Inbox::expanded');
```

If you need controller logic to decide which cells to load in a request, you can use the `CellTrait` in your controller to enable the `cell()` method there:

```
namespace App\Controller;

use App\Controller\AppController;
use Cake\View\CellTrait;

class DashboardsController extends AppController
{
    use CellTrait;

    // More code.
}
```

Passing Arguments to a Cell

You will often want to parameterize cell methods to make cells more flexible. By using the second and third arguments of `cell()`, you can pass action parameters and additional options to your cell classes, as an indexed array:

```
$cell = $this->cell('Inbox::recent', ['-3 days']);
```

The above would match the following function signature:

```
public function recent($since)
{
}
```

Rendering a Cell

Once a cell has been loaded and executed, you'll probably want to render it. The easiest way to render a cell is to echo it:

```
<?= $cell ?>
```

This will render the template matching the lowercased and underscored version of our action name like **display.php**.

Because cells use `View` to render templates, you can load additional cells within a cell template if required.

Note: Echoing a cell uses the PHP `__toString()` magic method which prevents PHP from showing the filename and line number for any fatal errors raised. To obtain a meaningful error message, it is recommended to use the `Cell::render()` method, for example `<?= $cell->render() ?>`.

Rendering Alternate Templates

By convention cells render templates that match the action they are executing. If you need to render a different view template, you can specify the template to use when rendering the cell:

```
// Calling render() explicitly
echo $this->cell('Inbox::recent', ['-3 days'])->render('messages');

// Set template before echoing the cell.
$cell = $this->cell('Inbox');
$cell->viewBuilder()->setTemplate('messages');

echo $cell;
```

Caching Cell Output

When rendering a cell you may want to cache the rendered output if the contents don't change often or to help improve performance of your application. You can define the `cache` option when creating a cell to enable & configure caching:

```
// Cache using the default config and a generated key
$cell = $this->cell('Inbox', [], ['cache' => true]);

// Cache to a specific cache config and a generated key
$cell = $this->cell('Inbox', [], ['cache' => ['config' => 'cell_cache']]);

// Specify the key and config to use.
$cell = $this->cell('Inbox', [], [
    'cache' => ['config' => 'cell_cache', 'key' => 'inbox_' . $user->id]
]);
```

If a key is generated the underscored version of the cell class and template name will be used.

Note: A new `View` instance is used to render each cell and these new objects do not share context with the main template / layout. Each cell is self-contained and only has access to variables passed as arguments to the `View::cell()` call.

Paginating Data inside a Cell

Creating a cell that renders a paginated result set can be done by leveraging a paginator class of the ORM. An example of paginating a user's favorite messages could look like:

```
namespace App\View\Cell;

use Cake\View\Cell;
use Cake\Datasource\Paging\NumericPaginator;

class FavoritesCell extends Cell
{
    public function display($user)
    {
        // Create a paginator
        $paginator = new NumericPaginator();

        // Paginate the model
        $results = $paginator->paginate(
            $this->fetchTable('Messages'),
            $this->request->getQueryParams(),
            [
                // Use a parameterized custom finder.
                'finder' => ['favorites' => [$user]],

                // Use scoped query string parameters.
                'scope' => 'favorites',
            ]
        );
```

(continues on next page)

(continued from previous page)

```
        // Set the paging params as a request attribute for use the PaginatorHelper
        $paging = $paginator->getPagingParams() + (array)$this->request->getAttribute(
↪'paging');
        $this->request = $this->request->withAttribute('paging', $paging);

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

The above cell would paginate the `Messages` model using *scoped pagination parameters*.

Cell Options

Cells can declare constructor options that are converted into properties when creating a cell object:

```
namespace App\View\Cell;

use Cake\View\Cell;

class FavoritesCell extends Cell
{
    protected $_validCellOptions = ['limit'];

    protected $limit = 3;

    public function display($userId)
    {
        $result = $this->fetchTable('Users')->find('friends', ['for' => $userId])
            ->limit($this->limit)
            ->all();
        $this->set('favorites', $result);
    }
}
```

Here we have defined a `$limit` property and add `limit` as a cell option. This will allow us to define the option when creating the cell:

```
$cell = $this->cell('Favorites', [$user->id], ['limit' => 10])
```

Cell options are handy when you want data available as properties allowing you to override default values.

Using Helpers inside a Cell

Cells have their own context and their own View instance but Helpers loaded inside your `AppView::initialize()` function are still loaded as usual.

Loading a specific Helper just for a specific cell can be done via the following example:

```
namespace App\View\Cell;

use Cake\View\Cell;
```

(continues on next page)

(continued from previous page)

```
class FavoritesCell extends Cell
{
    public function initialize(): void
    {
        $this->viewBuilder()->addHelper('MyCustomHelper');
    }
}
```

Themes

Themes in CakePHP are simply plugins that focus on providing template files. See the section on *Creating Your Own Plugins*. You can take advantage of themes, allowing you to switch the look and feel of your page quickly. In addition to template files, they can also provide helpers and cells if your theming requires that. When using cells and helpers from your theme, you will need to continue using the *plugin syntax*.

First ensure your theme plugin is loaded in your application's `bootstrap` method. For example:

```
// Load our plugin theme residing in the folder /plugins/Modern
$this->addPlugin('Modern');
```

To use themes, set the theme name in your controller's action or `beforeRender()` callback:

```
class ExamplesController extends AppController
{
    public function beforeRender(\Cake\Event\EventInterface $event)
    {
        $this->viewBuilder()->setTheme('Modern');
    }
}
```

Theme template files need to be within a plugin with the same name. For example, the above theme would be found in **plugins/Modern/templates**. It's important to remember that CakePHP expects PascalCase plugin/theme names. Beyond that, the folder structure within the **plugins/Modern/templates** folder is exactly the same as **templates/**.

For example, the view file for an edit action of a Posts controller would reside at **plugins/Modern/templates/Posts/edit.php**. Layout files would reside in **plugins/Modern/templates/layout/**. You can provide customized templates for plugins with a theme as well. If you had a plugin named 'Cms', that contained a TagsController, the Modern theme could provide **plugins/Modern/templates/plugin/Cms/Tags/edit.php** to replace the edit template in the plugin.

If a view file can't be found in the theme, CakePHP will try to locate the view file in the **templates/** folder. This way, you can create master template files and simply override them on a case-by-case basis within your theme folder.

Theme Assets

Because themes are standard CakePHP plugins, they can include any necessary assets in their webroot directory. This allows for packaging and distribution of themes. Whilst in development, requests for theme assets will be handled by `CakeRoutingMiddlewareAssetMiddleware` (which is loaded by default in cakephp/app `Application::middleware()`). To improve performance for production environments, it's recommended that you *Improve Your Application's Performance*.

All of CakePHP's built-in helpers are aware of themes and will create the correct paths automatically. Like template files, if a file isn't in the theme folder, it will default to the main webroot folder:

```
// When in a theme with the name of 'purple_cupcake'
$this->Html->css('main.css');

// creates a path like
/purple_cupcake/css/main.css

// and links to
plugins/PurpleCupcake/webroot/css/main.css
```

JSON and XML views

The `JsonView` and `XmlView` integration with CakePHP's *Content Type Negotiation* features and let you create JSON and XML responses.

These view classes are most commonly used alongside `CakeControllerController::viewClasses()`.

There are two ways you can generate data views. The first is by using the `serialize` option, and the second is by creating normal template files.

Defining View Classes to Negotiate With

In your `AppController` or in an individual controller you can implement the `viewClasses()` method and provide all of the views you want to support:

```
use Cake\View\JsonView;
use Cake\View\XmlView;

public function viewClasses(): array
{
    return [JsonView::class, XmlView::class];
}
```

You can optionally enable the json and/or xml extensions with *Routing File Extensions*. This will allow you to access the `JSON`, `XML` or any other special format views by using a custom URL ending with the name of the response type as a file extension such as `http://example.com/articles.json`.

By default, when not enabling *Routing File Extensions*, the request, the `Accept` header is used for, selecting which type of format should be rendered to the user. An example `Accept` format that is used to render `JSON` responses is `application/json`.

Using Data Views with the Serialize Key

The `serialize` option indicates which view variable(s) should be serialized when using a data view. This lets you skip defining template files for your controller actions if you don't need to do any custom formatting before your data is converted into json/xml.

If you need to do any formatting or manipulation of your view variables before generating the response, you should use template files. The value of `serialize` can be either a string or an array of view variables to serialize:

```
namespace App\Controller;

use Cake\View\JsonView;

class ArticlesController extends AppController
{
    public function viewClasses(): array
    {
        return [JsonView::class];
    }

    public function index()
    {
        // Set the view vars
        $this->set('articles', $this->paginate());
        // Specify which view vars JsonView should serialize.
        $this->viewBuilder()->setOption('serialize', 'articles');
    }
}
```

You can also define `serialize` as an array of view variables to combine:

```
namespace App\Controller;

use Cake\View\JsonView;

class ArticlesController extends AppController
{
    public function viewClasses(): array
    {
        return [JsonView::class];
    }

    public function index()
    {
        // Some code that created $articles and $comments

        // Set the view vars
        $this->set(compact('articles', 'comments'));

        // Specify which view vars JsonView should serialize.
        $this->viewBuilder()->setOption('serialize', ['articles', 'comments']);
    }
}
```

Defining `serialize` as an array has added the benefit of automatically appending a top-level `<response>` element

when using *XmlView*. If you use a string value for `serialize` and XmlView, make sure that your view variable has a single top-level element. Without a single top-level element the Xml will fail to generate.

Using a Data View with Template Files

You should use template files if you need to manipulate your view content before creating the final output. For example, if we had articles with a field containing generated HTML, we would probably want to omit that from a JSON response. This is a situation where a view file would be useful:

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

// View code - templates/Articles/json/index.php
foreach ($articles as $article) {
    unset($article->generated_html);
}
echo json_encode(compact('articles'));
```

You can do more complex manipulations, or use helpers to do formatting as well. The data view classes don't support layouts. They assume that the view file will output the serialized content.

Creating XML Views

class XmlView

By default when using `serialize` the XmlView will wrap your serialized view variables with a <response> node. You can set a custom name for this node using the `rootNode` option.

The XmlView class supports the `xmlOptions` option that allows you to customize the options, such as `tags` or `attributes`, used to generate XML.

An example of using `XmlView` would be to generate a sitemap.xml[113]. This document type requires that you change `rootNode` and set attributes. Attributes are defined using the `@` prefix:

```
public function sitemap()
{
    $pages = $this->Pages->find()->all();
    $urls = [];
    foreach ($pages as $page) {
        $urls[] = [
            'loc' => Router::url(['controller' => 'Pages', 'action' => 'view', $page->
↪slug, '_full' => true]),
            'lastmod' => $page->modified->format('Y-m-d'),
            'changefreq' => 'daily',
            'priority' => '0.5',
```

(continues on next page)

[113] https://www.sitemaps.org/protocol.html

(continued from previous page)

```
        ];
    }

    // Define a custom root node in the generated document.
    $this->viewBuilder()
        ->setOption('rootNode', 'urlset')
        ->setOption('serialize', ['@xmlns', 'url']);
    $this->set([
        // Define an attribute on the root node.
        '@xmlns' => 'http://www.sitemaps.org/schemas/sitemap/0.9',
        'url' => $urls,
    ]);
}
```

Creating JSON Views

class JsonView

The JsonView class supports the `jsonOptions` option that allows you to customize the bit-mask used to generate JSON. See the json_encode[114] documentation for the valid values of this option.

For example, to serialize validation error output of CakePHP entities in a consistent form of JSON do:

```
// In your controller's action when saving failed
$this->set('errors', $articles->errors());
$this->viewBuilder()
    ->setOption('serialize', ['errors'])
    ->setOption('jsonOptions', JSON_FORCE_OBJECT);
```

JSONP Responses

When using `JsonView` you can use the special view variable `jsonp` to enable returning a JSONP response. Setting it to `true` makes the view class check if query string parameter named "callback" is set and if so wrap the json response in the function name provided. If you want to use a custom query string parameter name instead of "callback" set `jsonp` to required name instead of `true`.

Choosing a View Class

While you can use the `viewClasses` hook method most of the time, if you want total control over view class selection you can directly choose the view class:

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

use App\Controller\AppController;
use Cake\Http\Exception\NotFoundException;

class VideosController extends AppController
```

(continues on next page)

[114] https://php.net/json_encode

(continued from previous page)

```
{
    public function export($format = '')
    {
        $format = strtolower($format);

        // Format to view mapping
        $formats = [
          'xml' => 'Xml',
          'json' => 'Json',
        ];

        // Error on unknown type
        if (!isset($formats[$format])) {
            throw new NotFoundException(__('Unknown format.'));
        }

        // Set Out Format View
        $this->viewBuilder()->setClassName($formats[$format]);

        // Get data
        $videos = $this->Videos->find('latest')->all();

        // Set Data View
        $this->set(compact('videos'));
        $this->viewBuilder()->setOption('serialize', ['videos']);

        // Set Force Download
        return $this->response->withDownload('report-' . date('YmdHis') . '.' . $format);
    }
}
```

Helpers

Helpers are the component-like classes for the presentation layer of your application. They contain presentational logic that is shared between many views, elements, or layouts. This chapter will show you how to configure helpers. How to load helpers and use those helpers, and outline the simple steps for creating your own custom helpers.

CakePHP includes a number of helpers that aid in view creation. They assist in creating well-formed markup (including forms), aid in formatting text, times and numbers, and can even speed up AJAX functionality. For more information on the helpers included in CakePHP, check out the chapter for each helper:

Breadcrumbs

class Cake\View\Helper**BreadcrumbsHelper**(*View $view*, *array $config = []*)

BreadcrumbsHelper provides a way to easily deal with the creation and rendering of a breadcrumbs trail for your app.

Creating a Breadcrumbs Trail

You can add a crumb to the list using the `add()` method. It takes three arguments:

- **title** The string to be displayed as a the title of the crumb
- **url** A string or an array of parameters that will be given to the *Url*
- **options** An array of attributes for the `item` and `itemWithoutLink` templates. See the section about *defining attributes for the item* for more information.

In addition to adding to the end of the trail, you can do a variety of operations:

```
// Add at the end of the trail
$this->Breadcrumbs->add(
    'Products',
    ['controller' => 'products', 'action' => 'index']
);

// Add multiple crumbs at the end of the trail
$this->Breadcrumbs->add([
    ['title' => 'Products', 'url' => ['controller' => 'products', 'action' => 'index']],
    ['title' => 'Product name', 'url' => ['controller' => 'products', 'action' => 'view',
↪ 1234]],
]);

// Prepended crumbs will be put at the top of the list
$this->Breadcrumbs->prepend(
    'Products',
    ['controller' => 'products', 'action' => 'index']
);

// Prepend multiple crumbs at the top of the trail, in the order given
$this->Breadcrumbs->prepend([
    ['title' => 'Products', 'url' => ['controller' => 'products', 'action' => 'index']],
    ['title' => 'Product name', 'url' => ['controller' => 'products', 'action' => 'view',
↪ 1234]],
]);

// Insert in a specific slot. If the slot is out of
// bounds, an exception will be raised.
$this->Breadcrumbs->insertAt(
    2,
    'Products',
    ['controller' => 'products', 'action' => 'index']
);

// Insert before another crumb, based on the title.
// If the named crumb title cannot be found,
// an exception will be raised.
$this->Breadcrumbs->insertBefore(
    'A product name', // the title of the crumb to insert before
    'Products',
    ['controller' => 'products', 'action' => 'index']
);
```

(continues on next page)

(continued from previous page)

```
// Insert after another crumb, based on the title.
// If the named crumb title cannot be found,
// an exception will be raised.
$this->Breadcrumbs->insertAfter(
    'A product name', // the title of the crumb to insert after
    'Products',
    ['controller' => 'products', 'action' => 'index']
);
```

Using these methods gives you the ability to work with CakePHP's 2-step rendering process. Since templates and layouts are rendered from the inside out (meaning, included elements are rendered first), this allows you to define precisely where you want to add a breadcrumb.

Rendering the Breadcrumbs Trail

After adding crumbs to the trail, you can easily render it using the `render()` method. This method accepts two array arguments:

- `$attributes` : An array of attributes that will applied to the `wrapper` template. This gives you the ability to add attributes to the HTML tag. It accepts the special `templateVars` key to allow the insertion of custom template variables in the template.
- `$separator` : An array of attributes for the `separator` template. Possible properties are:
  - `separator` The string to be displayed as a separator
  - `innerAttrs` To provide attributes in case your separator is divided in two elements
  - `templateVars` Allows the insertion of custom template variable in the template

  All other properties will be converted as HTML attributes and will replace the `attrs` key in the template. If you use the default for this option (empty), it will not render a separator.

Here is an example of how to render a trail:

```
echo $this->Breadcrumbs->render(
    ['class' => 'breadcrumbs-trail'],
    ['separator' => '<i class="fa fa-angle-right"></i>']
);
```

Customizing the Output

The BreadcrumbsHelper internally uses the `StringTemplateTrait`, which gives the ability to easily customize output of various HTML strings. It includes four templates, with the following default declaration:

```
[
    '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>'
]
```

You can easily customize them using the `setTemplates()` method from the `StringTemplateTrait`:

```
$this->Breadcrumbs->setTemplates([
    'wrapper' => '<nav class="breadcrumbs"><ul{{attrs}}>{{content}}</ul></nav>',
]);
```

Since your templates will be rendered, the `templateVars` option allows you to add your own template variables in the various templates:

```
$this->Breadcrumbs->setTemplates([
    'item' => '<li{{attrs}}>{{icon}}<a href="{{url}}"{{innerAttrs}}>{{title}}</a></li>{
↪{separator}}'
]);
```

And to define the `{{icon}}` parameter, just specify it when adding the crumb to the trail:

```
$this->Breadcrumbs->add(
    'Products',
    ['controller' => 'products', 'action' => 'index'],
    [
        'templateVars' => [
            'icon' => '<i class="fa fa-money"></i>',
        ],
    ]
);
```

Defining Attributes for the Item

If you want to apply specific HTML attributes to both the item and its sub-item , you can leverage the `innerAttrs` key, which the `$options` argument provides. Everything except `innerAttrs` and `templateVars` will be rendered as HTML attributes:

```
$this->Breadcrumbs->add(
    'Products',
    ['controller' => 'products', 'action' => 'index'],
    [
        'class' => 'products-crumb',
        'data-foo' => 'bar',
        'innerAttrs' => [
            'class' => 'inner-products-crumb',
            'id' => 'the-products-crumb',
        ],
    ]
);

// Based on the default template, this will render the following HTML:
<li class="products-crumb" data-foo="bar">
    <a href="/products/index" class="inner-products-crumb" id="the-products-crumb">
↪Products</a>
</li>
```

Clearing the Breadcrumbs

You can clear the bread crumbs using the `reset()` method. This can be useful when you want to transform the crumbs and overwrite the list:

```
$crumbs = $this->Breadcrumbs->getCrumbs();
$crumbs = collection($crumbs)->map(function ($crumb) {
    $crumb['options']['class'] = 'breadcrumb-item';
    return $crumb;
})->toArray();

$this->Breadcrumbs->reset()->add($crumbs);
```

Flash

class Cake\View\Helper**FlashHelper**(*View $view*, *array $config = []*)

FlashHelper provides a way to render flash messages that were set in `$_SESSION` by *FlashComponent*. *FlashComponent* and FlashHelper primarily use elements to render flash messages. Flash elements are found under the **templates/element/flash** directory. You'll notice that CakePHP's App template comes with three flash elements: **success.php**, **default.php**, and **error.php**.

Rendering Flash Messages

To render a flash message, you can simply use FlashHelper's `render()` method in your template file:

```
<?= $this->Flash->render() ?>
```

By default, CakePHP uses a "flash" key for flash messages in a session. But, if you've specified a key when setting the flash message in *FlashComponent*, you can specify which flash key to render:

```
<?= $this->Flash->render('other') ?>
```

You can also override any of the options that were set in FlashComponent:

```
// In your Controller
$this->Flash->set('The user has been saved.', [
    'element' => 'success'
]);

// In your template file: Will use great_success.php instead of success.php
<?= $this->Flash->render('flash', [
    'element' => 'great_success'
]);

// In your template file: the flashy element file from the Company Plugin
<?= $this->Flash->render('flash', [
    'element' => 'Company.flashy'
]);
```

Note: When building custom flash message templates, be sure to properly HTML encode any user data. CakePHP won't escape flash message parameters for you.

For more information about the available array options, please refer to the *FlashComponent* section.

Routing Prefix and Flash Messages

If you have a Routing prefix configured, you can now have your Flash elements stored in **templates/{Prefix}/element/flash**. This way, you can have specific messages layouts for each part of your application. For instance, using different layouts for your front-end and admin section.

Flash Messages and Themes

The FlashHelper uses normal elements to render the messages and will therefore obey any theme you might have specified. So when your theme has a **templates/element/flash/error.php** file it will be used, just as with any Elements and Views.

Form

`class` Cake\View\Helper**FormHelper**(*View $view*, *array $config = []*)

The FormHelper does most of the heavy lifting in form creation. The FormHelper focuses on creating forms quickly, in a way that will streamline validation, re-population and layout. The FormHelper is also flexible - it will do almost everything for you using conventions, or you can use specific methods to get only what you need.

Starting a Form

Cake\View\Helper\FormHelper::**create**(*mixed $context = null*, *array $options = []*)

- `$context` - The context for which the form is being defined. Can be an ORM entity, ORM resultset, `Form` instance, array of metadata or `null` (to make a model-less form).
- `$options` - An array of options and/or HTML attributes.

The first method you'll need to use in order to take advantage of the FormHelper is `create()`. This method outputs an opening form tag.

All parameters are optional. If `create()` is called with no parameters supplied, it assumes you are building a form that submits to the current controller, via the current URL. The default method for form submission is POST. If you were to call `create()` inside the view for `UsersController::add()`, you would see something like the following output in the rendered view:

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

The `$context` argument is used as the form's 'context'. There are several built-in form contexts and you can add your own, which we'll cover below, in a following section. The built-in providers map to the following values of `$context`:

- An `Entity` instance or an iterator will map to EntityContext[115]; this context class allows FormHelper to work with results from the built-in ORM.

[115] https://api.cakephp.org/5.x/class-Cake.View.Form.EntityContext.html

- An array containing the `'schema'` key, will map to ArrayContext[116] which allows you to create simple data structures to build forms against.
- `null` will map to NullContext[117]; this context class simply satisfies the interface FormHelper requires. This context is useful if you want to build a short form that doesn't require ORM persistence.

Once a form has been created with a context, all controls you create will use the active context. In the case of an ORM backed form, FormHelper can access associated data, validation errors and schema metadata. You can close the active context using the `end()` method, or by calling `create()` again.

To create a form for an entity, do the following:

```
// If you are on /articles/add
// $article should be an empty Article entity.
echo $this->Form->create($article);
```

Output:

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

This will POST the form data to the `add()` action of ArticlesController. However, you can also use the same logic to create an edit form. The FormHelper uses the `Entity` object to automatically detect whether to create an *add* or *edit* form. If the provided entity is not 'new', the form will be created as an *edit* form.

For example, if we browse to **http://example.org/articles/edit/5**, we could do the following:

```
// src/Controller/ArticlesController.php:
public function edit($id = null)
{
    if (empty($id)) {
        throw new NotFoundException;
    }
    $article = $this->Articles->get($id);
    // Save logic goes here
    $this->set('article', $article);
}

// View/Articles/edit.php:
// Since $article->isNew() is false, we will get an edit form
<?= $this->Form->create($article) ?>
```

Output:

```
<form method="post" action="/articles/edit/5">
<input type="hidden" name="_method" value="PUT" />
```

Note: Since this is an *edit* form, a hidden `input` field is generated to override the default HTTP method.

In some cases, the entity's ID is automatically appended to the end of the form's `action` URL. If you would like to *avoid* an ID being added to the URL, you can pass a string to `$options['url']`, such as `'/my-account'` or `\Cake\Routing\Router::url(['controller' => 'Users', 'action' => 'myAccount'])`.

[116] https://api.cakephp.org/5.x/class-Cake.View.Form.ArrayContext.html
[117] https://api.cakephp.org/5.x/class-Cake.View.Form.NullContext.html

Options for Form Creation

The `$options` array is where most of the form configuration happens. This special array can contain a number of different key-value pairs that affect the way the form tag is generated. Valid values:

- `'type'` - Allows you to choose the type of form to create. If no type is provided then it will be autodetected based on the form context. Valid values:
  - `'get'` - Will set the form method to HTTP GET.
  - `'file'` - Will set the form method to POST and the `'enctype'` to "multipart/form-data".
  - `'post'` - Will set the method to POST.
  - `'put', 'delete', 'patch'` - Will override the HTTP method with PUT, DELETE or PATCH respectively, when the form is submitted.
- `'method'` - Valid values are the same as above. Allows you to explicitly override the form's method.
- `'url'` - Specify the URL the form will submit to. Can be a string or a URL array.
- `'encoding'` - Sets the `accept-charset` encoding for the form. Defaults to `Configure::read('App.encoding')`.
- `'enctype'` - Allows you to set the form encoding explicitly.
- `'templates'` - The templates you want to use for this form. Any templates provided will be merged on top of the already loaded templates. Can be either a filename (without extension) from `/config` or an array of templates to use.
- `'context'` - Additional options for the form context class. (For example the `EntityContext` accepts a `'table'` option that allows you to set the specific Table class the form should be based on.)
- `'idPrefix'` - Prefix for generated ID attributes.
- `'templateVars'` - Allows you to provide template variables for the `formStart` template.
- `autoSetCustomValidity` - Set to `true` to use custom required and notBlank validation messages in the control's HTML5 validity message. Default is `true`.

Tip: Besides the above options you can provide, in the `$options` argument, any valid HTML attributes that you want to pass to the created `form` element.

Getting form values from other values sources

A FormHelper's values sources define where its rendered elements, such as input-tags, receive their values from.

The supported sources are `context`, `data` and `query`. You can use one or more sources by setting `valueSources` option or by using `setValuesSource()`. Any widgets generated by `FormHelper` will gather their values from the sources, in the order you setup.

By default `FormHelper` draws its values from `data` or `context`, i.e. it will fetch data from `$request->getData()` or, if not present, from the active context's data, that are the entity's data in the case of `EntityContext`.

If however, you are building a form that needs to read from the query string, you can change where `FormHelper` reads input data from:

```
// Use query string instead of request data:
echo $this->Form->create($article, [
    'type' => 'get',
    'valueSources' => ['query', 'context'],
]);

// Same effect:
echo $this->Form
    ->setValueSources(['query', 'context'])
    ->create($articles, ['type' => 'get']);
```

When input data has to be processed by the entity, i.e. marshal transformations, table query result or entity computations, and displayed after one or multiple form submissions where request data is retained, you need to put `context` first:

```
// Prioritize context over request data:
echo $this->Form->create($article,
    'valueSources' => ['context', 'data'],
]);
```

The value sources will be reset to the default `['data', 'context']` when `end()` is called.

Changing the HTTP Method for a Form

By using the `type` option you can change the HTTP method a form will use:

```
echo $this->Form->create($article, ['type' => 'get']);
```

Output:

```
<form method="get" action="/articles/edit/5">
```

Specifying a `'file'` value for `type`, changes the form submission method to 'post', and includes an `enctype` of "multipart/form-data" on the form tag. This is to be used if there are any file elements inside the form. The absence of the proper `enctype` attribute will cause the file uploads not to function.

For example:

```
echo $this->Form->create($article, ['type' => 'file']);
```

Output:

```
<form enctype="multipart/form-data" method="post" action="/articles/add">
```

When using `'put'`, `'patch'` or `'delete'` as `'type'` values, your form will be functionally equivalent to a 'post' form, but when submitted, the HTTP request method will be overridden with 'PUT', 'PATCH' or 'DELETE', respectively. This allows CakePHP to emulate proper REST support in web browsers.

Setting a URL for the Form

Using the `'url'` option allows you to point the form to a specific action in your current controller or another controller in your application.

For example, if you'd like to point the form to the `publish()` action of the current controller, you would supply an `$options` array, like the following:

```
echo $this->Form->create($article, ['url' => ['action' => 'publish']]);
```

Output:

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

If the desired form action isn't in the current controller, you can specify a complete URL for the form action. The supplied URL can be relative to your CakePHP application:

```
echo $this->Form->create(null, [
    'url' => [
        'controller' => 'Articles',
        'action' => 'publish',
    ],
]);
```

Output:

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

Or you can point to an external domain:

```
echo $this->Form->create(null, [
    'url' => 'https://www.google.com/search',
    'type' => 'get',
]);
```

Output:

```
<form method="get" action="https://www.google.com/search">
```

Use `'url' => false` if you don't want to output a URL as the form action.

Using Custom Validators

Often models will have multiple validator sets, you can have FormHelper mark fields required based on the specific validator your controller action is going to apply. For example, your Users table has specific validation rules that only apply when an account is being registered:

```
echo $this->Form->create($user, [
    'context' => ['validator' => 'register'],
]);
```

The above will use validation rules defined in the `register` validator, which are defined by `UsersTable::validationRegister()`, for `$user` and all related associations. If you are creating a form for associated entities, you can define validation rules for each association by using an array:

```
echo $this->Form->create($user, [
    'context' => [
        'validator' => [
            'Users' => 'register',
            'Comments' => 'default',
        ],
    ],
]);
```

The above would use `register` for the user, and `default` for the user's comments. FormHelper uses validators to generate HTML5 required attributes, relevant ARIA attributes, and set error messages with the browser validator API[118] . If you would like to disable HTML5 validation messages use:

```
$this->Form->setConfig('autoSetCustomValidity', false);
```

This will not disable `required`/`aria-required` attributes.

Creating context classes

While the built-in context classes are intended to cover the basic cases you'll encounter you may need to build a new context class if you are using a different ORM. In these situations you need to implement the Cake\View\Form\ContextInterface[119] . Once you have implemented this interface you can wire your new context into the FormHelper. It is often best to do this in a `View.beforeRender` event listener, or in an application view class:

```
$this->Form->addContextProvider('myprovider', function ($request, $data) {
    if ($data['entity'] instanceof MyOrmClass) {
        return new MyProvider($data);
    }
});
```

Context factory functions are where you can add logic for checking the form options for the correct type of entity. If matching input data is found you can return an object. If there is no match return null.

Creating Form Controls

`Cake\View\Helper\FormHelper::control`(*string $fieldName*, *array $options = []*)

- `$fieldName` - A field name in the form `'Modelname.fieldname'`.
- `$options` - An optional array that can include both *Options for Control*, and options of the other methods (which `control()` employs internally to generate various HTML elements) as well as any valid HTML attributes.

The `control()` method lets you generate complete form controls. These controls will include a wrapping `div`, `label`, control widget, and validation error if necessary. By using the metadata in the form context, this method will choose an appropriate control type for each field. Internally `control()` uses the other methods of FormHelper.

Tip: Please note that while the fields generated by the `control()` method are called generically "inputs" on this page, technically speaking, the `control()` method can generate not only all of the HTML `input` type elements, but also other HTML form elements such as `select`, `button`, `textarea`.

---

[118] https://developer.mozilla.org/en-US/docs/Learn/HTML/Forms/Form_validation#Customized_error_messages
[119] https://api.cakephp.org/5.x/interface-Cake.View.Form.ContextInterface.html

By default the `control()` method will employ the following widget templates:

```
'inputContainer' => '<div class="input {{type}}{{required}}">{{content}}</div>'
'input' => '<input type="{{type}}" name="{{name}}"{{attrs}}/>'
'requiredClass' => 'required'
```

In case of validation errors it will also use:

```
'inputContainerError' => '<div class="input {{type}}{{required}} error">{{content}}{
↪{error}}</div>'
```

The type of control created (when we provide no additional options to specify the generated element type) is inferred via model introspection and depends on the column datatype:

Column Type
: Resulting Form Field

string, uuid (char, varchar, etc.)
: text

boolean, tinyint(1)
: checkbox

decimal
: number

float
: number

integer
: number

text
: textarea

text, with name of password, passwd
: password

text, with name of email
: email

text, with name of tel, telephone, or phone
: tel

date
: date

datetime, timestamp
: datetime-local

datetimefractional, timestampfractional
: datetime-local

time
: time

month
: month

year
: select with years

binary
: file

The `$options` parameter allows you to choose a specific control type if you need to:

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

Tip: As a small subtlety, generating specific elements via the `control()` form method will always also generate the wrapping `div`, by default. Generating the same type of element via one of the specific form methods (e.g. `$this->Form->checkbox('published');`) in most cases won't generate the wrapping `div`. Depending on your needs you can use one or another.

The wrapping `div` will have a `required` class name appended if the validation rules for the model's field indicate that it is required and not allowed to be empty. You can disable automatic `required` flagging using the `'required'` option:

```
echo $this->Form->control('title', ['required' => false]);
```

To skip browser validation triggering for the whole form you can set option `'formnovalidate' => true` for the input button you generate using *submit()* or set `'novalidate' => true` in options for *create()*.

For example, let's assume that your Users model includes fields for a *username* (varchar), *password* (varchar), *approved* (datetime) and *quote* (text). You can use the `control()` method of the FormHelper to create appropriate controls for all of these form fields:

```
echo $this->Form->create($user);
// The following generates a Text input
echo $this->Form->control('username');
// The following generates a Password input
echo $this->Form->control('password');
// Assuming 'approved' is a datetime or timestamp field the following
//generates an input of type "datetime-local"
echo $this->Form->control('approved');
// The following generates a Textarea element
echo $this->Form->control('quote');

echo $this->Form->button('Add');
echo $this->Form->end();
```

A more extensive example showing some options for a date field:

```
echo $this->Form->control('birth_date', [
    'label' => 'Date of birth',
    'min' => date('Y') - 70,
    'max' => date('Y') - 18,
]);
```

Besides the specific *Options for Control*, you also can specify any option accepted by corresponding specific method for the chosen (or inferred by CakePHP) control type and any HTML attribute (for instance `onfocus`).

If you want to create a `select` form field while using a *belongsTo* (or *hasOne*) relation, you can add the following to your UsersController (assuming your User *belongsTo* Group):

```
$this->set('groups', $this->Users->Groups->find('list')->all());
```

Afterwards, add the following to your view template:

```
echo $this->Form->control('group_id', ['options' => $groups]);
```

To make a `select` box for a *belongsToMany* Groups association you can add the following to your UsersController:

```
$this->set('groups', $this->Users->Groups->find('list')->all());
```

Afterwards, add the following to your view template:

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

If your model name consists of two or more words (e.g. "UserGroups"), when passing the data using `set()` you should name your data in a pluralised and lower camelCased[120] format as follows:

```
$this->set('userGroups', $this->UserGroups->find('list')->all());
```

Note: You should not use `FormHelper::control()` to generate submit buttons. Use *`submit()`* instead.

Field Naming Conventions

When creating control widgets you should name your fields after the matching attributes in the form's entity. For example, if you created a form for an `$article` entity, you would create fields named after the properties. E.g. `title`, `body` and `published`.

You can create controls for associated models, or arbitrary models by passing in `association.fieldname` as the first parameter:

```
echo $this->Form->control('association.fieldname');
```

Any dots in your field names will be converted into nested request data. For example, if you created a field with a name `0.comments.body` you would get a name attribute that looks like `0[comments][body]`. This convention matches the conventions you use with the ORM. Details for the various association types can be found in the *Creating Inputs for Associated Data* section.

When creating datetime related controls, FormHelper will append a field-suffix. You may notice additional fields named `year`, `month`, `day`, `hour`, `minute`, or `meridian` being added. These fields will be automatically converted into `DateTime` objects when entities are marshalled.

Options for Control

`FormHelper::control()` supports a large number of options via its `$options` argument. In addition to its own options, `control()` accepts options for the inferred/chosen generated control types (e.g. for `checkbox` or `textarea`), as well as HTML attributes. This subsection will cover the options specific to `FormHelper::control()`.

- `$options['type']` - A string that specifies the widget type to be generated. In addition to the field types found in the *Creating Form Controls*, you can also create `'file'`, `'password'`, and any other type supported by HTML5. By specifying a `'type'` you will force the type of the generated control, overriding model introspection. Defaults to `null`.

  For example:

[120] https://en.wikipedia.org/wiki/Camel_case#Variations_and_synonyms

```
echo $this->Form->control('field', ['type' => 'file']);
echo $this->Form->control('email', ['type' => 'email']);
```

Output:

```
<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']` - Either a string caption or an array of *options for the label*. You can set this key to the string you would like to be displayed within the label that usually accompanies the `input` HTML element. Defaults to `null`.

  For example:

  ```
  echo $this->Form->control('name', [
      'label' => 'The User Alias'
  ]);
  ```

  Output:

  ```
  <div class="input">
      <label for="name">The User Alias</label>
      <input name="name" type="text" value="" id="name" />
  </div>
  ```

  Alternatively, set this key to `false` to disable the generation of the `label` element.

  For example:

  ```
  echo $this->Form->control('name', ['label' => false]);
  ```

  Output:

  ```
  <div class="input">
      <input name="name" type="text" value="" id="name" />
  </div>
  ```

  If the label is disabled, and a `placeholder` attribute is provided, the generated input will have `aria-label` set.

  Set the `label` option to an array to provide additional options for the `label` element. If you do this, you can use a `'text'` key in the array to customize the label text.

  For example:

  ```
  echo $this->Form->control('name', [
      'label' => [
          'class' => 'thingy',
          'text' => 'The User Alias'
      ]
  ]);
  ```

Output:

```
<div class="input">
    <label for="name" class="thingy">The User Alias</label>
    <input name="name" type="text" value="" id="name" />
</div>
```

- `$options['options']` - You can provide in here an array containing the elements to be generated for widgets such as `radio` or `select`, which require an array of items as an argument (see *Creating Radio Buttons* and *Creating Select Pickers* for more details). Defaults to `null`.
- `$options['error']` - Using this key allows you to override the default model error messages and can be used, for example, to set i18n messages. To disable the error message output & field classes set the `'error'` key to `false`. Defaults to `null`.

  For example:

  ```
  echo $this->Form->control('name', ['error' => false]);
  ```

  To override the model error messages use an array with the keys matching the original validation error messages.

  For example:

  ```
  $this->Form->control('name', [
      'error' => ['Not long enough' => __('This is not long enough')]
  ]);
  ```

  As seen above you can set the error message for each validation rule you have in your models. In addition you can provide i18n messages for your forms.

  To disable the HTML entity encoding for error messages only, the `'escape'` sub key can be used:

  ```
  $this->Form->control('name', [
      'error' => ['escape' => false],
  ]);
  ```

- `$options['nestedInput']` - Used with checkboxes and radio buttons. Controls whether the input element is generated inside or outside the `label` element. When `control()` generates a checkbox or a radio button, you can set this to `false` to force the generation of the HTML `input` element outside of the `label` element.

  On the other hand you can set this to `true` for any control type to force the generated input element inside the label. If you change this for radio buttons then you need to also modify the default *radioWrapper* template. Depending on the generated control type it defaults to `true` or `false`.
- `$options['templates']` - The templates you want to use for this input. Any specified templates will be merged on top of the already loaded templates. This option can be either a filename (without extension) in `/config` that contains the templates you want to load, or an array of templates to use.
- `$options['labelOptions']` - Set this to `false` to disable labels around nestedWidgets or set it to an array of attributes to be provided to the `label` tag.
- `$options['readonly']` - Set the field to `readonly` in form.

  For example:

  ```
  echo $this->Form->control('name', ['readonly' => true]);
  ```

Generating Specific Types of Controls

In addition to the generic `control()` method, `FormHelper` has specific methods for generating a number of different types of controls. These can be used to generate just the control widget itself, and combined with other methods like *`label()`* and *`error()`* to generate fully custom form layouts.

Common Options For Specific Controls

Many of the various control element methods support a common set of options which, depending on the form method used, must be provided inside the `$options` or in the `$attributes` array argument. All of these options are also supported by the `control()` method. To reduce repetition, the common options shared by all control methods are as follows:

- `'id'` - Set this key to force the value of the DOM id for the control. This will override the `'idPrefix'` that may be set.
- `'default'` - Used to set a default value for the control field. The value is used if the data passed to the form does not contain a value for the field (or if no data is passed at all). If no default value is provided, the column's default value will be used.

  Example usage:

  ```
  echo $this->Form->text('ingredient', ['default' => 'Sugar']);
  ```

  Example with `select` field (size "Medium" will be selected as default):

  ```
  $sizes = ['s' => 'Small', 'm' => 'Medium', 'l' => 'Large'];
  echo $this->Form->select('size', $sizes, ['default' => 'm']);
  ```

  Note: You cannot use `default` to check a checkbox - instead you might set the value in `$this->request->getData()` in your controller, or set the control option `'checked'` to `true`.

  Beware of using `false` to assign a default value. A `false` value is used to disable/exclude options of a control field, so `'default' => false` would not set any value at all. Instead use `'default' => 0`.

- `'value'` - Used to set a specific value for the control field. This will override any value that may else be injected from the context, such as Form, Entity or `request->getData()` etc.

  Note: If you want to set a field to not render its value fetched from context or valuesSource you will need to set `'value'` to `''` (instead of setting it to `null`).

In addition to the above options, you can mixin any HTML attribute you wish to use. Any non-special option name will be treated as an HTML attribute, and applied to the generated HTML control element.

Creating Input Elements

The rest of the methods available in the FormHelper are for creating specific form elements. Many of these methods also make use of a special `$options` or `$attributes` parameter. In this case, however, this parameter is used primarily to specify HTML tag attributes (such as the value or DOM id of an element in the form).

Creating Text Inputs

`Cake\View\Helper\FormHelper::text`(*string $name*, *array $options*)

- `$name` - A field name in the form `'Modelname.fieldname'`.
- `$options` - An optional array including any of the *Common Options For Specific Controls* as well as any valid HTML attributes.

Creates a simple `input` HTML element of `text` type.

For example:

```
echo $this->Form->text('username', ['class' => 'users']);
```

Will output:

```
<input name="username" type="text" class="users">
```

Creating Password Inputs

`Cake\View\Helper\FormHelper::password`(*string $fieldName*, *array $options*)

- `$fieldName` - A field name in the form `'Modelname.fieldname'`.
- `$options` - An optional array including any of the *Common Options For Specific Controls* as well as any valid HTML attributes.

Creates a simple `input` element of `password` type.

For example:

```
echo $this->Form->password('password');
```

Will output:

```
<input name="password" value="" type="password">
```

Creating Hidden Inputs

`Cake\View\Helper\FormHelper::hidden`(*string $fieldName*, *array $options*)

- `$fieldName` - A field name in the form `'Modelname.fieldname'`.
- `$options` - An optional array including any of the *Common Options For Specific Controls* as well as any valid HTML attributes.

Creates a hidden form input.

For example:

```
echo $this->Form->hidden('id');
```

Will output:

```
<input name="id" type="hidden" />
```

Creating Textareas

Cake\View\Helper\FormHelper::**textarea**(*string $fieldName*, *array $options*)

- `$fieldName` - A field name in the form `'Modelname.fieldname'`.
- `$options` - An optional array including any of the *Common Options For Specific Controls*, of the specific textarea options (see below) as well as any valid HTML attributes.

Creates a textarea control field. The default widget template used is:

```
'textarea' => '<textarea name="{{name}}"{{attrs}}>{{value}}</textarea>'
```

For example:

```
echo $this->Form->textarea('notes');
```

Will output:

```
<textarea name="notes"></textarea>
```

If the form is being edited (i.e. the array `$this->request->getData()` contains the information previously saved for the `User` entity), the value corresponding to `notes` field will automatically be added to the HTML generated.

Example:

```
<textarea name="notes" id="notes">
    This text is to be edited.
</textarea>
```

Options for Textarea

In addition to the *Common Options For Specific Controls*, `textarea()` supports a couple of specific options:

- `'escape'` - Determines whether or not the contents of the textarea should be escaped. Defaults to `true`.

  For example:

  ```
  echo $this->Form->textarea('notes', ['escape' => false]);
  // OR....
  echo $this->Form->control('notes', ['type' => 'textarea', 'escape' => false]);
  ```

- `'rows'`, `'cols'` - You can use these two keys to set the HTML attributes which specify the number of rows and columns for the `textarea` field.

  For example:

  ```
  echo $this->Form->textarea('comment', ['rows' => '5', 'cols' => '5']);
  ```

  Output:

```
<textarea name="comment" cols="5" rows="5">
</textarea>
```

Creating Select, Checkbox and Radio Controls

These controls share some commonalities and a few options and thus, they are all grouped in this subsection for easier reference.

Options for Select, Checkbox and Radio Controls

You can find below the options which are shared by `select()`, `checkbox()` and `radio()` (the options particular only to one of the methods are described in each method's own section.)

- `'value'` - Sets or selects the value of the affected element(s):
  - For checkboxes, it sets the HTML `'value'` attribute assigned to the `input` element to whatever you provide as value.
  - For radio buttons or select pickers it defines which element will be selected when the form is rendered (in this case `'value'` must be assigned a valid, existent element value). May also be used in combination with any select-type control, such as `date()`, `time()`, `dateTime()`:

    ```
    echo $this->Form->time('close_time', [
        'value' => '13:30:00',
    ]);
    ```

Note: The `'value'` key for `date()` and `dateTime()` controls may also have as value a UNIX timestamp, or a DateTime object.

For a `select` control where you set the `'multiple'` attribute to `true`, you can provide an array with the values you want to select by default:

```
// HTML <option> elements with values 1 and 3 will be rendered preselected
echo $this->Form->select(
    'rooms',
    [1, 2, 3, 4, 5],
    [
        'multiple' => true,
        'value' => [1, 3]
    ]
);
```

- `'empty'` - Applies to `radio()` and `select()`. Defaults to `false`.
  - When passed to `radio()` and set to `true` it will create an extra input element as the first radio button, with a value of `''` and a label caption equal to the string `'empty'`. If you want to control the label caption set this option to a string instead.
  - When passed to a `select` method, this creates a blank HTML `option` element with an empty value in your drop down list. If you want to have an empty value with text displayed instead of just a blank `option`, pass a string to `'empty'`:

```
echo $this->Form->select(
    'field',
    [1, 2, 3, 4, 5],
    ['empty' => '(choose one)']
);
```

Output:

```
<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>
    <option value="4">5</option>
</select>
```

- `'hiddenField'` - For checkboxes and radio buttons, by default, a hidden `input` element is also created, along with the main element, so that the key in `$this->request->getData()` will exist even without a value specified. For checkboxes its value defaults to `0` and for radio buttons to `''`.

  Example of default output:

  ```
  <input type="hidden" name="published" value="0" />
  <input type="checkbox" name="published" value="1" />
  ```

  This can be disabled by setting `'hiddenField'` to `false`:

  ```
  echo $this->Form->checkbox('published', ['hiddenField' => false]);
  ```

  Which outputs:

  ```
  <input type="checkbox" name="published" value="1">
  ```

  If you want to create multiple blocks of controls on a form, that are all grouped together, you should set this parameter to `false` on all controls except the first. If the hidden input is on the page in multiple places, only the last group of `inputs`' values will be saved.

  In this example, only the tertiary colors would be passed, and the primary colors would be overridden:

  ```
  <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" />
      Yellow
  ```

(continues on next page)

(continued from previous page)

```
</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" />
    Green
</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>
```

Disabling `'hiddenField'` on the second control group would prevent this behavior.

You can set a hidden field to a value other than 0, such as 'N':

```
echo $this->Form->checkbox('published', [
    'value' => 'Y',
    'hiddenField' => 'N',
]);
```

Using Collections to build options

It's possible to use the Collection class to build your options array. This approach is ideal if you already have a collection of entities and would like to build a select element from them.

You can use the `combine` method to build a basic options array.:

```
$options = $examples->combine('id', 'name');
```

It's also possible to add extra attributes by expanding the array. The following will create a data attribute on the option element, using the `map` collection method.:

```
$options = $examples->map(function ($value, $key) {
    return [
        'value' => $value->id,
        'text' => $value->name,
        'data-created' => $value->created
    ];
});
```

Creating Checkboxes

`Cake\View\Helper\FormHelper::checkbox`(*string $fieldName*, *array $options*)

- `$fieldName` - A field name in the form `'Modelname.fieldname'`.
- `$options` - An optional array including any of the *Common Options For Specific Controls*, or of the *Options for Select, Checkbox and Radio Controls* above, of the checkbox-specific options (see below), as well as any valid HTML attributes.

Creates a `checkbox` form element. The widget template used is:

```
'checkbox' => '<input type="checkbox" name="{{name}}" value="{{value}}"{{attrs}}>'
```

Options for Checkboxes

- `'checked'` - Boolean to indicate whether this checkbox will be checked. Defaults to `false`.
- `'disabled'` - Create a disabled checkbox input.

This method also generates an associated hidden form `input` element to force the submission of data for the specified field.

For example:

```
echo $this->Form->checkbox('done');
```

Will output:

```
<input type="hidden" name="done" value="0">
<input type="checkbox" name="done" value="1">
```

It is possible to specify the value of the checkbox by using the `$options` array.

For example:

```
echo $this->Form->checkbox('done', ['value' => 555]);
```

Will output:

```
<input type="hidden" name="done" value="0">
<input type="checkbox" name="done" value="555">
```

If you don't want the FormHelper to create a hidden input use `'hiddenField'`.

For example:

```
echo $this->Form->checkbox('done', ['hiddenField' => false]);
```

Will output:

```
<input type="checkbox" name="done" value="1">
```

Creating Radio Buttons

Cake\View\Helper\FormHelper::**radio**(*string $fieldName*, *array $options*, *array $attributes*)

- `$fieldName` - A field name in the form `'Modelname.fieldname'`.
- `$options` - An optional array containing at minimum the labels for the radio buttons. Can also contain values and HTML attributes. When this array is missing, the method will either generate only the hidden input (if `'hiddenField'` is `true`) or no element at all (if `'hiddenField'` is `false`).
- `$attributes` - An optional array including any of the *Common Options For Specific Controls*, or of the *Options for Select, Checkbox and Radio Controls*, of the radio button specific attributes (see below), as well as any valid HTML attributes.

Creates a set of radio button inputs. The default widget templates used are:

```
'radio' => '<input type="radio" name="{{name}}" value="{{value}}"{{attrs}}>'
'radioWrapper' => '{{label}}'
```

Attributes for Radio Buttons

- `'label'` - Boolean to indicate whether or not labels for widgets should be displayed, or an array of attributes to apply to all labels. In case a `class` attribute is defined, `selected` will be added to the `class` attribute of checked buttons. Defaults to `true`.
- `'hiddenField'` - If set to `true` a hidden input with a value of `''` will be included. This is useful for creating radio sets that are non-continuous. Defaults to `true`.
- `'disabled'` - Set to `true` or `'disabled'` to disable all the radio buttons. Defaults to `false`.

You must provide the label captions for the radio buttons via the `$options` argument.

For example:

```
$this->Form->radio('gender', ['Masculine', 'Feminine', 'Neuter']);
```

Will output:

```
<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>
```

Generally `$options` contains simple `key => value` pairs. However, if you need to put custom attributes on your radio buttons you can use an expanded format.

For example:

```
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;'],
    ]
);
```

Will output:

```
<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>
```

You can define additional attributes for an individual option's label as well:

```
echo $this->Form->radio(
    'favorite_color',
    [
        ['value' => 'r', 'text' => 'Red', 'label' => ['class' => 'red']],
        ['value' => 'u', 'text' => 'Blue', 'label' => ['class' => 'blue']],
    ]
);
```

Will output:

```
<input type="hidden" name="favorite_color" value="">
<label for="favorite-color-r" class="red">
    <input type="radio" name="favorite_color" value="r" style="color:red;" id="favorite-
↪color-r">
    Red
</label>
<label for="favorite-color-u" class="blue">
    <input type="radio" name="favorite_color" value="u" style="color:blue;" id="favorite-
↪color-u">
    Blue
</label>
```

If the `label` key is used on an option, the attributes in `$attributes['label']` will be ignored.

Creating Select Pickers

Cake\View\Helper\FormHelper::**select**(*string $fieldName*, *array $options*, *array $attributes*)

- `$fieldName` - A field name in the form `'fieldname'` or `'related_entity.fieldname'`. This will provide the `name` attribute of the `select` element.
- `$options` - An optional array containing the list of items for the select picker. When this array is missing, the method will generate only the empty `select` HTML element without any `option` elements inside it.
- `$attributes` - An optional array including any of the *Common Options For Specific Controls*, or of the *Options for Select, Checkbox and Radio Controls*, or of the select-specific attributes (see below), as well as any valid HTML attributes.

Creates a `select` element, populated with the items from the `$options` array. If `$attributes['value']` is provided, then the HTML `option` element(s) which have the specified value(s) will be shown as selected when rendering the select picker.

By default `select` uses the following widget templates:

```
'select' => '<select name="{{name}}"{{attrs}}>{{content}}</select>'
'option' => '<option value="{{value}}"{{attrs}}>{{text}}</option>'
```

May also use:

```
'optgroup' => '<optgroup label="{{label}}"{{attrs}}>{{content}}</optgroup>'
'selectMultiple' => '<select name="{{name}}[]" multiple="multiple"{{attrs}}>{{content}}</
↪select>'
```

Attributes for Select Pickers

- `'multiple'` - If set to `true` allows multiple selections in the select picker. If set to `'checkbox'`, multiple checkboxes will be created instead. Defaults to `null`.
- `'escape'` - Boolean. If `true` the contents of the `option` elements inside the select picker will be HTML entity encoded. Defaults to `true`.
- `'val'` - Allows preselecting a value in the select picker.
- `'disabled'` - Controls the `disabled` attribute. If set to `true` disables the whole select picker. If set to an array it will disable only those specific `option` elements whose values are provided in the array.

The `$options` argument allows you to manually specify the contents of the `option` elements of a `select` control.

For example:

```
echo $this->Form->select('field', [1, 2, 3, 4, 5]);
```

Output:

```
<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>
```

The array for `$options` can also be supplied as key-value pairs.

For example:

```
echo $this->Form->select('field', [
    'Value 1' => 'Label 1',
    'Value 2' => 'Label 2',
    'Value 3' => 'Label 3'
]);
```

Output:

```
<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>
```

If you would like to generate a `select` with optgroups, just pass data in hierarchical format (nested array). This works on multiple checkboxes and radio buttons too, but instead of `optgroup` it wraps the elements in `fieldset` elements.

For example:

```
$options = [
    'Group 1' => [
        'Value 1' => 'Label 1',
        'Value 2' => 'Label 2',
    ],
    'Group 2' => [
        'Value 3' => 'Label 3',
    ],
];
echo $this->Form->select('field', $options);
```

Output:

```
<select name="field">
    <optgroup label="Group 1">
        <option value="Value 1">Label 1</option>
        <option value="Value 2">Label 2</option>
    </optgroup>
    <optgroup label="Group 2">
        <option value="Value 3">Label 3</option>
    </optgroup>
</select>
```

To generate HTML attributes within an `option` tag:

```
$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);
```

Output:

```
<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>
```

Controlling Select Pickers via Attributes

By using specific options in the `$attributes` parameter you can control certain behaviors of the `select()` method.

- `'empty'` - Set the `'empty'` key in the `$attributes` argument to `true` (the default value is `false`) to add a blank option with an empty value at the top of your dropdown list.

  For example:

  ```
  $options = ['M' => 'Male', 'F' => 'Female'];
  echo $this->Form->select('gender', $options, ['empty' => true]);
  ```

  Will output:

  ```
  <select name="gender">
      <option value=""></option>
      <option value="M">Male</option>
      <option value="F">Female</option>
  </select>
  ```

- `'escape'` - The `select()` method allows for an attribute called `'escape'` which accepts a boolean value and determines whether to HTML entity encode the contents of the `select`'s `option` elements.

  For example:

  ```
  // 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'` - If set to `true`, the select picker will allow multiple selections.

  For example:

  ```
  echo $this->Form->select('field', $options, ['multiple' => true]);
  ```

  Alternatively, set `'multiple'` to `'checkbox'` in order to output a list of related checkboxes:

  ```
  $options = [
      'Value 1' => 'Label 1',
      'Value 2' => 'Label 2'
  ];
  echo $this->Form->select('field', $options, [
      'multiple' => 'checkbox'
  ]);
  ```

  Output:

  ```
  <input name="field" value="" type="hidden">
  <div class="checkbox">
      <label for="field-1">
  ```

(continues on next page)

(continued from previous page)

```
        <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'` - This option can be set in order to disable all or some of the `select`'s `option` items. To disable all items set `'disabled'` to `true`. To disable only certain items, assign to `'disabled'` an array containing the keys of the items to be disabled.

  For example:

```
$options = [
    'M' => 'Masculine',
    'F' => 'Feminine',
    'N' => 'Neuter'
];
echo $this->Form->select('gender', $options, [
    'disabled' => ['M', 'N']
]);
```

  Will output:

```
<select name="gender">
    <option value="M" disabled="disabled">Masculine</option>
    <option value="F">Feminine</option>
    <option value="N" disabled="disabled">Neuter</option>
</select>
```

  This option also works when `'multiple'` is set to `'checkbox'`:

```
$options = [
    'Value 1' => 'Label 1',
    'Value 2' => 'Label 2'
];
echo $this->Form->select('field', $options, [
    'multiple' => 'checkbox',
    'disabled' => ['Value 1']
]);
```

  Output:

```
<input name="field" value="" type="hidden">
<div class="checkbox">
    <label for="field-1">
        <input name="field[]" disabled="disabled" value="Value 1" type="checkbox">
        Label 1
    </label>
```

(continues on next page)

(continued from previous page)

```
</div>
<div class="checkbox">
    <label for="field-2">
        <input name="field[]" value="Value 2" id="field-2" type="checkbox">
        Label 2
    </label>
</div>
```

Creating File Inputs

Cake\View\Helper\FormHelper::**file**(*string $fieldName*, *array $options*)

- `$fieldName` - A field name in the form `'Modelname.fieldname'`.
- `$options` - An optional array including any of the *Common Options For Specific Controls* as well as any valid HTML attributes.

Creates a file upload field in the form. The widget template used by default is:

```
'file' => '<input type="file" name="{{name}}"{{attrs}}>'
```

To add a file upload field to a form, you must first make sure that the form enctype is set to `'multipart/form-data'`.

So start off with a `create()` method such as the following:

```
echo $this->Form->create($document, ['enctype' => 'multipart/form-data']);
// OR
echo $this->Form->create($document, ['type' => 'file']);
```

Next add a line that looks like either of the following two lines to your form's view template file:

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

// OR
echo $this->Form->file('submittedfile');
```

Note: Due to the limitations of HTML itself, it is not possible to put default values into input fields of type 'file'. Each time the form is displayed, the value inside will be empty.

To prevent the `submittedfile` from being over-written as blank, remove it from `$_accessible`. Alternatively, you can unset the index by using `beforeMarshal`:

```
public function beforeMarshal(\Cake\Event\EventInterface $event, \ArrayObject $data, \
↪ArrayObject $options)
{
   if ($data['submittedfile'] === '') {
      unset($data['submittedfile']);
   }
}
```

Upon submission, file fields can be accessed though `UploadedFileInterface` objects on the request. To move uploaded files to a permanent location, you can use:

```
$fileobject = $this->request->getData('submittedfile');
$destination = UPLOAD_DIRECTORY . $fileobject->getClientFilename();

// Existing files with the same name will be replaced.
$fileobject->moveTo($destination);
```

Note: When using `$this->Form->file()`, remember to set the form encoding-type, by setting the `'type'` option to `'file'` in `$this->Form->create()`.

Creating Date & Time Related Controls

`Cake\View\Helper\FormHelper::dateTime`(*$fieldName*, *$options = []*)

- `$fieldName` - A string that will be used as a prefix for the HTML `name` attribute of the `select` elements.
- `$options` - An optional array including any of the *Common Options For Specific Controls* as well as any valid HTML attributes.

This method will generate an input tag with type "datetime-local".

For example

```
<?= $this->form->dateTime('registered') ?>
```

Output:

```
<input type="datetime-local" name="registered" />
```

The value for the input can be any valid datetime string or `DateTime` instance.

For example

```
<?= $this->form->dateTime('registered', ['value' => new DateTime()]) ?>
```

Output:

```
<input type="datetime-local" name="registered" value="2019-02-08T18:20:10" />
```

Creating Date Controls

`Cake\View\Helper\FormHelper::date`(*$fieldName*, *$options = []*)

- `$fieldName` - A field name that will be used as a prefix for the HTML `name` attribute of the `select` elements.
- `$options` - An optional array including any of the *Common Options For Specific Controls* as well as any valid HTML attributes.

This method will generate an input tag with type "date".

For example

```
<?= $this->form->date('registered') ?>
```

Output:

```
<input type="date" name="registered" />
```

Creating Time Controls

Cake\View\Helper\FormHelper::**time**(*$fieldName*, *$options = []*)

- `$fieldName` - A field name that will be used as a prefix for the HTML `name` attribute of the `select` elements.
- `$options` - An optional array including any of the *Common Options For Specific Controls* as well as any valid HTML attributes.

This method will generate an input tag with type "time".

For example

```
echo $this->Form->time('released');
```

Output:

```
<input type="time" name="released" />
```

Creating Month Controls

Cake\View\Helper\FormHelper::**month**(*string $fieldName*, *array $attributes*)

- `$fieldName` - A field name that will be used as a prefix for the HTML `name` attribute of the `select` element.
- `$options` - An optional array including any of the *Common Options For Specific Controls* as well as any valid HTML attributes.

This method will generate an input tag with type "month".

For example:

```
echo $this->Form->month('mob');
```

Will output:

```
<input type="month" name="mob" />
```

Creating Year Controls

Cake\View\Helper\FormHelper::**year**(*string $fieldName*, *array $options = []*)

- `$fieldName` - A field name that will be used as a prefix for the HTML `name` attribute of the `select` element.
- `$options` - An optional array including any of the *Common Options For Specific Controls* as well as any valid HTML attributes. Other valid options are:
  - `min`: The lowest value to use in the year select picker.

- `max`: The maximum value to use in the year select picker.
- `order`: The order of year values in the year select picker. Possible values are `'asc'` and `'desc'`. Defaults to `'desc'`.

Creates a `select` element populated with the years from `min` to `max` (when these options are provided) or else with values starting from -5 years to +5 years counted from today. Additionally, HTML attributes may be supplied in `$options`. If `$options['empty']` is `false`, the select picker will not include an empty item in the list.

For example, to create a year range from 2000 to the current year you would do the following:

```
echo $this->Form->year('purchased', [
    'min' => 2000,
    'max' => date('Y')
]);
```

If it was 2009, you would get the following:

```
<select name="purchased">
    <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>
```

Creating Labels

`Cake\View\Helper\FormHelper::label`(*string $fieldName*, *string $text*, *array $options*)

- `$fieldName` - A field name in the form `'Modelname.fieldname'`.
- `$text` - An optional string providing the label caption text.
- `$options` - Optional. Array containing any of the *Common Options For Specific Controls* as well as any valid HTML attributes.

Creates a `label` element. The argument `$fieldName` is used for generating the HTML `for` attribute of the element; if `$text` is undefined, `$fieldName` will also be used to inflect the label's `text` attribute.

For example:

```
echo $this->Form->label('name');
echo $this->Form->label('name', 'Your username');
```

Output:

```
<label for="name">Name</label>
<label for="name">Your username</label>
```

With the third parameter `$options` you can set the id or class:

```
echo $this->Form->label('name', null, ['id' => 'user-label']);
echo $this->Form->label('name', 'Your username', ['class' => 'highlight']);
```

Output:

```
<label for="name" id="user-label">Name</label>
<label for="name" class="highlight">Your username</label>
```

Displaying and Checking Errors

FormHelper exposes a couple of methods that allow us to easily check for field errors and when necessary display customized error messages.

Displaying Errors

`Cake\View\Helper\FormHelper::error`(*string $fieldName*, *mixed $text*, *array $options*)

- `$fieldName` - A field name in the form `'Modelname.fieldname'`.
- `$text` - Optional. A string or array providing the error message(s). If an array, then it should be a hash of key names => messages. Defaults to `null`.
- `$options` - An optional array that can only contain a boolean with the key `'escape'`, which will define whether to HTML escape the contents of the error message. Defaults to `true`.

Shows a validation error message, specified by `$text`, for the given field, in the event that a validation error has occurred. If `$text` is not provided then the default validation error message for that field will be used.

Uses the following template widgets:

```
'error' => '<div class="error-message">{{content}}</div>'
'errorList' => '<ul>{{content}}</ul>'
'errorItem' => '<li>{{text}}</li>'
```

The `'errorList'` and `'errorItem'` templates are used to format mutiple error messages per field.

Example:

```
// If in TicketsTable you have a 'notEmpty' validation rule:
public function validationDefault(Validator $validator): Validator
{
    $validator
        ->requirePresence('ticket', 'create')
        ->notEmpty('ticket');
}

// And inside templates/Tickets/add.php you have:
echo $this->Form->text('ticket');

if ($this->Form->isFieldError('ticket')) {
    echo $this->Form->error('ticket', 'Completely custom error message!');
}
```

If you would click the *Submit* button of your form without providing a value for the *Ticket* field, your form would output:

```
<input name="ticket" class="form-error" required="required" value="" type="text">
<div class="error-message">Completely custom error message!</div>
```

Note: When using *control()*, errors are rendered by default, so you don't need to use `isFieldError()` or call `error()` manually.

Tip: If you use a certain model field to generate multiple form fields via `control()`, and you want the same validation error message displayed for each one, you will probably be better off defining a custom error message inside the respective *validator rules*.

Checking for Errors

`Cake\View\Helper\FormHelper::isFieldError`(*string $fieldName*)

- `$fieldName` - A field name in the form `'Modelname.fieldname'`.

Returns `true` if the supplied `$fieldName` has an active validation error, otherwise returns `false`.

Example:

```
if ($this->Form->isFieldError('gender')) {
    echo $this->Form->error('gender');
}
```

Displaying validation messages in HTML5 validity messages

If the `autoSetCustomValidity` FormHelper option is set to `true`, error messages for the field's required and notBlank validation rules will be used in lieu of the default browser HTML5 required messages. Enabling the option will add the `onvalid` and `oninvalid` event attributes to your fields, for example:

```
<input type="text" name="field" required onvalid="this.setCustomValidity('')" oninvalid=
↪"this.setCustomValidity('Custom notBlank message')" />
```

If you want to manually set those events with custom JavaScript, you can set the `autoSetCustomValidity` option to `false` and use the special `customValidityMessage` template variable instead. This template variable is added when a field is required:

```
// example template
[
    'input' => '<input type="{{type}}" name="{{name}}" data-error-message="{
↪{customValidityMessage}}" {{attrs}}/>',
]

// would create an input like this
<input type="text" name="field" required data-error-message="Custom notBlank message" />
```

You could then use JavaScript to set the `onvalid` and `oninvalid` events as you like.

Creating Buttons and Submit Elements

Creating Submit Elements

Cake\View\Helper\FormHelper::**submit**(*string $caption*, *array $options*)

- `$caption` - An optional string providing the button's text caption or a path to an image. Defaults to `'Submit'`.
- `$options` - An optional array including any of the *Common Options For Specific Controls*, or of the specific submit options (see below) as well as any valid HTML attributes.

Creates an `input` element of `submit` type, with `$caption` as value. If the supplied `$caption` is a URL pointing to an image (i.e. if the string contains '://' or contains any of the extensions '.jpg, .jpe, .jpeg, .gif'), an image submit button will be generated, using the specified image if it exists. If the first character is '/' then the image path is relative to *webroot*, else if the first character is not '/' then the image path is relative to *webroot/img*.

By default it will use the following widget templates:

```
'inputSubmit' => '<input type="{{type}}"{{attrs}}/>'
'submitContainer' => '<div class="submit">{{content}}</div>'
```

Options for Submit

- `'type'` - Set this option to `'reset'` in order to generate reset buttons. It defaults to `'submit'`.
- `'templateVars'` - Set this array to provide additional template variables for the input element and its container.
- Any other provided attributes will be assigned to the `input` element.

The following:

```
echo $this->Form->submit('Click me');
```

Will output:

```
<div class="submit"><input value="Click me" type="submit"></div>
```

You can pass a relative or absolute URL of an image to the caption parameter instead of the caption text:

```
echo $this->Form->submit('ok.png');
```

Will output:

```
<div class="submit"><input type="image" src="/img/ok.png"></div>
```

Submit inputs are useful when you only need basic text or images. If you need more complex button content you should use `button()`.

Creating Button Elements

Cake\View\Helper\FormHelper::**button**(*string $title*, *array $options = []*)

- `$title` - Mandatory string providing the button's text caption.
- `$options` - An optional array including any of the *Common Options For Specific Controls*, or of the specific button options (see below) as well as any valid HTML attributes.

Creates an HTML button with the specified title and a default type of `'button'`.

Options for Button

- `'type'` - You can set this to one of the following three possible values:
  1. `'submit'` - Similarly to the `$this->Form->submit()` method it will create a submit button. However this won't generate a wrapping `div` as `submit()` does. This is the default type.
  2. `'reset'` - Creates a form reset button.
  3. `'button'` - Creates a standard push button.
- `'escapeTitle'` - Boolean. If set to `true` it will HTML encode the value provided inside `$title`. Defaults to `true`.
- `'escape'` - Boolean. If set to `true` it will HTML encode all the HTML attributes generated for the button. Defaults to `true`.
- `'confirm'` - The confirmation message to display on click. Defaults to `null`.

For example:

```
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']);
```

Will output:

```
<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>
```

Example use of the `'escapeTitle'` option:

```
// Will render unescaped HTML.
echo $this->Form->button('<em>Submit Form</em>', [
    'type' => 'submit',
    'escapeTitle' => false,
]);
```

Closing the Form

Cake\View\Helper\FormHelper::**end**(*$secureAttributes = []*)

- `$secureAttributes` - Optional. Allows you to provide secure attributes which will be passed as HTML attributes into the hidden input elements generated for the SecurityComponent.

The `end()` method closes and completes a form. Often, `end()` will only output a closing form tag, but using `end()` is a good practice as it enables FormHelper to insert the hidden form elements that `Cake\Controller\Component\SecurityComponent` requires:

```
<?= $this->Form->create(); ?>

<!-- Form elements go here -->

<?= $this->Form->end(); ?>
```

If you need to add additional attributes to the generated hidden inputs you can use the `$secureAttributes` argument.

For example:

```
echo $this->Form->end(['data-type' => 'hidden']);
```

Will output:

```
<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: If you are using `Cake\Controller\Component\SecurityComponent` in your application you should always end your forms with `end()`.

Creating Standalone Buttons and POST Links

Creating POST Buttons

Cake\View\Helper\FormHelper::**postButton**(*string $title*, *mixed $url*, *array $options = []*)

- `$title` - Mandatory string providing the button's text caption. By default not HTML encoded.
- `$url` - The URL of the form provided as a string or as array.
- `$options` - An optional array including any of the *Common Options For Specific Controls*, or of the specific options (see below) as well as any valid HTML attributes.

Creates a <button> tag with a surrounding <form> element that submits via POST, by default. Also, by default, it generates hidden input fields for the SecurityComponent.

Options for POST Button

- `'data'` - Array with key/value to pass in hidden input.

- `'method'` - Request method to use. E.g. set to `'delete'` to simulate a HTTP/1.1 DELETE request. Defaults to `'post'`.
- `'form'` - Array with any option that `FormHelper::create()` can take.
- Also, the `postButton()` method will accept the options which are valid for the `button()` method.

For example:

```
// In templates/Tickets/index.php
<?= $this->Form->postButton('Delete Record', ['controller' => 'Tickets', 'action' =>
↪'delete', 5]) ?>
```

Will output HTML similar to:

```
<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">Delete Record</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>
```

Since this method generates a `form` element, do not use this method in an already opened form. Instead use *`Cake\View\Helper\FormHelper::submit()`* or *`Cake\View\Helper\FormHelper::button()`* to create buttons inside opened forms.

Creating POST Links

`Cake\View\Helper\FormHelper::`**`postLink`**(*string $title*, *mixed $url = null*, *array $options = []*)

- `$title` - Mandatory string providing the text to be wrapped in <a> tags.
- `$url` - Optional. String or array which contains the URL of the form (Cake-relative or external URL starting with `http://`).
- `$options` - An optional array including any of the *Common Options For Specific Controls*, or of the specific options (see below) as well as any valid HTML attributes.

Creates an HTML link, but accesses the URL using the method you specify (defaults to POST). Requires JavaScript to be enabled in browser:

```
// In your template, to delete an article, for example
<?= $this->Form->postLink(
    'Delete',
    ['action' => 'delete', $article->id],
    ['confirm' => 'Are you sure?'])
?>
```

Options for POST Link

- `'data'` - Array with key/value to pass in hidden input.
- `'method'` - Request method to use. For example, setting it to `'delete'` will simulate a HTTP/1.1 DELETE request. Defaults to `'post'`.
- `'confirm'` - The confirmation message to display on click. Defaults to `null`.
- `'block'` - Set this option to `true` to append the form to view block `'postLink'` or provide a custom block name. Defaults to `null`.
- Also, the `postLink` method will accept the options which are valid for the `link()` method.

This method creates a `<form>` element. If you want to use this method inside of an existing form, you must use the `block` option so that the new form is being set to a *view block* that can be rendered outside of the main form.

If all you are looking for is a button to submit your form, then you should use *Cake\View\Helper\FormHelper::button()* or *Cake\View\Helper\FormHelper::submit()* instead.

Note: Be careful to not put a postLink inside an open form. Instead use the `block` option to buffer the form into a *view block*

Customizing the Templates FormHelper Uses

Like many helpers in CakePHP, FormHelper uses string templates to format the HTML it creates. While the default templates are intended to be a reasonable set of defaults, you may need to customize the templates to suit your application.

To change the templates when the helper is loaded you can set the `'templates'` option when including the helper in your controller:

```
// In a View class
$this->loadHelper('Form', [
    'templates' => 'app_form',
]);
```

This would load the tags found in **config/app_form.php**. This file should contain an array of templates *indexed by name*:

```
// in config/app_form.php
return [
    'inputContainer' => '<div class="form-control">{{content}}</div>',
];
```

Any templates you define will replace the default ones included in the helper. Templates that are not replaced, will continue to use the default values.

You can also change the templates at runtime using the `setTemplates()` method:

```
$myTemplates = [
    'inputContainer' => '<div class="form-control">{{content}}</div>',
];
$this->Form->setTemplates($myTemplates);
```

Warning: Template strings containing a percentage sign (%) need special attention; you should prefix this character with another percentage so it looks like %%. The reason is that internally templates are compiled to be used with `sprintf()`. Example: `'<div style="width:{{size}}%%">{{content}}</div>'`

List of Templates

The list of default templates, their default format and the variables they expect can be found in the FormHelper API documentation[121].

Using Distinct Custom Control Containers

In addition to these templates, the `control()` method will attempt to use distinct templates for each control container. For example, when creating a datetime control the `datetimeContainer` will be used if it is present. If that container is missing the `inputContainer` template will be used.

For example:

```
// Add custom radio wrapping HTML
$this->Form->setTemplates([
    'radioContainer' => '<div class="form-radio">{{content}}</div>'
]);

// Create a radio set with our custom wrapping div.
echo $this->Form->control('email_notifications', [
    'options' => ['y', 'n'],
    'type' => 'radio'
]);
```

Using Distinct Custom Form Groups

Similar to controlling containers, the `control()` method will also attempt to use distinct templates for each form group. A form group is a combo of label and control. For example, when creating a radio control the `radioFormGroup` will be used if it is present. If that template is missing by default each set of `label` & `input` is rendered using the default `formGroup` template.

For example:

```
// Add custom radio form group
$this->Form->setTemplates([
    'radioFormGroup' => '<div class="radio">{{label}}{{input}}</div>'
]);
```

[121] https://api.cakephp.org/5.x/class-Cake.View.Helper.FormHelper.html#%24_defaultConfig

Adding Additional Template Variables to Templates

You can add additional template placeholders in custom templates, and populate those placeholders when generating controls.

For example:

```
// Add a template with the help placeholder.
$this->Form->setTemplates([
    'inputContainer' => '<div class="input {{type}}{{required}}">
        {{content}} <span class="help">{{help}}</span></div>'
]);

// Generate an input and populate the help variable
echo $this->Form->control('password', [
    'templateVars' => ['help' => 'At least 8 characters long.'],
]);
```

Output:

```
<div class="input password">
    <label for="password">
        Password
    </label>
    <input name="password" id="password" type="password">
    <span class="help">At least 8 characters long.</span>
</div>
```

Moving Checkboxes & Radios Outside of a Label

By default CakePHP nests checkboxes created via `control()` and radio buttons created by both `control()` and `radio()` within label elements. This helps make it easier to integrate popular CSS frameworks. If you need to place checkbox/radio inputs outside of the label you can do so by modifying the templates:

```
$this->Form->setTemplates([
    'nestingLabel' => '{{hidden}}{{input}}<label{{attrs}}>{{text}}</label>',
    'formGroup' => '{{input}}{{label}}',
]);
```

This will make radio buttons and checkboxes render outside of their labels.

Generating Entire Forms

Creating Multiple Controls

`Cake\View\Helper\FormHelper::controls`(*array $fields = [], $options = []*)

- `$fields` - An array of fields to generate. Allows setting custom types, labels and other options for each specified field.
- `$options` - Optional. An array of options. Valid keys are:

1. `'fieldset'` - Set this to `false` to disable the fieldset. If empty, the fieldset will be enabled. Can also be an array of parameters to be applied as HTML attributes to the `fieldset` tag.
2. `legend` - String used to customize the `legend` text. Set this to `false` to disable the legend for the generated input set.

Generates a set of controls for the given context wrapped in a `fieldset`. You can specify the generated fields by including them:

```
echo $this->Form->controls([
    'name',
    'email'
]);
```

You can customize the legend text using an option:

```
echo $this->Form->controls($fields, ['legend' => 'Update news post']);
```

You can customize the generated controls by defining additional options in the `$fields` parameter:

```
echo $this->Form->controls([
    'name' => ['label' => 'custom label'],
]);
```

When customizing, `$fields`, you can use the `$options` parameter to control the generated legend/fieldset.

For example:

```
echo $this->Form->controls(
    [
        'name' => ['label' => 'custom label'],
    ],
    [
        'legend' => 'Update your post',
    ]
);
```

If you disable the `fieldset`, the `legend` will not print.

Creating Controls for a Whole Entity

`Cake\View\Helper\FormHelper::allControls`(*array $fields*, *array $options = []*)

- `$fields` - Optional. An array of customizations for the fields that will be generated. Allows setting custom types, labels and other options.
- `$options` - Optional. An array of options. Valid keys are:
  1. `'fieldset'` - Set this to `false` to disable the fieldset. If empty, the fieldset will be enabled. Can also be an array of parameters to be applied as HTMl attributes to the `fieldset` tag.
  2. `legend` - String used to customize the `legend` text. Set this to `false` to disable the legend for the generated control set.

This method is closely related to `controls()`, however the `$fields` argument is defaulted to *all* fields in the current top-level entity. To exclude specific fields from the generated controls, set them to `false` in the `$fields` parameter:

```
echo $this->Form->allControls(['password' => false]);
```

Creating Inputs for Associated Data

Creating forms for associated data is straightforward and is closely related to the paths in your entity's data. Assuming the following table relations:

- Authors HasOne Profiles
- Authors HasMany Articles
- Articles HasMany Comments
- Articles BelongsTo Authors
- Articles BelongsToMany Tags

If we were editing an article with its associations loaded we could create the following controls:

```
$this->Form->create($article);

// Article controls.
echo $this->Form->control('title');

// Author controls (belongsTo)
echo $this->Form->control('author.id');
echo $this->Form->control('author.first_name');
echo $this->Form->control('author.last_name');

// Author profile (belongsTo + hasOne)
echo $this->Form->control('author.profile.id');
echo $this->Form->control('author.profile.username');

// Tags controls (belongsToMany)
// as separate inputs
echo $this->Form->control('tags.0.id');
echo $this->Form->control('tags.0.name');
echo $this->Form->control('tags.1.id');
echo $this->Form->control('tags.1.name');

// Inputs for the joint table (articles_tags)
echo $this->Form->control('tags.0._joinData.starred');
echo $this->Form->control('tags.1._joinData.starred');

// Comments controls (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');
```

The above controls could then be marshalled into a completed entity graph using the following code in your controller:

```
$article = $this->Articles->patchEntity($article, $this->request->getData(), [
    'associated' => [
```

(continues on next page)

(continued from previous page)

```
        'Authors',
        'Authors.Profiles',
        'Tags',
        'Comments',
    ],
]);
```

The above example shows an expanded example for belongs to many associations, with separate inputs for each entity and join data record. You can also create a multiple select input for belongs to many associations:

```
// Multiple select element for belongsToMany
// Does not support _joinData
echo $this->Form->control('tags._ids', [
    'type' => 'select',
    'multiple' => true,
    'options' => $tagList,
]);
```

Adding Custom Widgets

You can add custom control widgets in CakePHP, and use them like any other control type. All of the core control types are implemented as widgets, which means you can override any core widget with your own implementation as well.

Building a Widget Class

Widget classes have a very simple required interface. They must implement the `Cake\View\Widget\WidgetInterface`. This interface requires the `render(array $data)` and `secureFields(array $data)` methods to be implemented. The `render()` method expects an array of data to build the widget and is expected to return a string of HTML for the widget. The `secureFields()` method expects an array of data as well and is expected to return an array containing the list of fields to secure for this widget. If CakePHP is constructing your widget you can expect to get a `Cake\View\StringTemplate` instance as the first argument, followed by any dependencies you define. If we wanted to build an Autocomplete widget you could do the following:

```
namespace App\View\Widget;

use Cake\View\Form\ContextInterface;
use Cake\View\StringTemplate;
use Cake\View\Widget\WidgetInterface;

class AutocompleteWidget implements WidgetInterface
{
    /**
     * StringTemplate instance.
     *
     * @var \Cake\View\StringTemplate
     */
    protected $_templates;

    /**
     * Constructor.
```

(continues on next page)

(continued from previous page)

```
     *
     * @param \Cake\View\StringTemplate $templates Templates list.
     */
    public function __construct(StringTemplate $templates)
    {
        $this->_templates = $templates;
    }

    /**
     * Methods that render the widget.
     *
     * @param array $data The data to build an input with.
     * @param \Cake\View\Form\ContextInterface $context The current form context.
     *
     * @return string
     */
    public function render(array $data, ContextInterface $context): string
    {
        $data += [
            'name' => '',
        ];
        return $this->_templates->format('autocomplete', [
            'name' => $data['name'],
            'attrs' => $this->_templates->formatAttributes($data, ['name'])
        ]);
    }

    public function secureFields(array $data): array
    {
        return [$data['name']];
    }
}
```

Obviously, this is a very simple example, but it demonstrates how a custom widget could be built. This widget would render the "autocomplete" string template, such as:

```
$this->Form->setTemplates([
    'autocomplete' => '<input type="autocomplete" name="{{name}}" {{attrs}} />'
]);
```

For more information on string templates, see *Customizing the Templates FormHelper Uses*.

Using Widgets

You can load custom widgets when loading FormHelper or by using the `addWidget()` method. When loading FormHelper, widgets are defined as a setting:

```
// In View class
$this->loadHelper('Form', [
    'widgets' => [
        'autocomplete' => ['Autocomplete'],
```

(continues on next page)

(continued from previous page)

```
    ],
]);
```

If your widget requires other widgets, you can have FormHelper populate those dependencies by declaring them:

```
$this->loadHelper('Form', [
    'widgets' => [
        'autocomplete' => [
            'App\View\Widget\AutocompleteWidget',
            'text',
            'label',
        ],
    ],
]);
```

In the above example, the `autocomplete` widget would depend on the `text` and `label` widgets. If your widget needs access to the View, you should use the `_view` 'widget'. When the `autocomplete` widget is created, it will be passed the widget objects that are related to the `text` and `label` names. To add widgets using the `addWidget()` method would look like:

```
// Using a classname.
$this->Form->addWidget(
    'autocomplete',
    ['Autocomplete', 'text', 'label']
);

// Using an instance - requires you to resolve dependencies.
$autocomplete = new AutocompleteWidget(
    $this->Form->getTemplater(),
    $this->Form->getWidgetLocator()->get('text'),
    $this->Form->getWidgetLocator()->get('label'),
);
$this->Form->addWidget('autocomplete', $autocomplete);
```

Once added/replaced, widgets can be used as the control 'type':

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

This will create the custom widget with a `label` and wrapping `div` just like `controls()` always does. Alternatively, you can create just the control widget using the magic method:

```
echo $this->Form->autocomplete('search', $options);
```

Working with SecurityComponent

`Cake\Controller\Component\SecurityComponent` offers several features that make your forms safer and more secure. By simply including the `SecurityComponent` in your controller, you'll automatically benefit from form tampering-prevention features.

As mentioned previously when using SecurityComponent, you should always close your forms using *end()*. This will ensure that the special `_Token` inputs are generated.

`Cake\View\Helper\FormHelper::unlockField`(*$name*)

- `$name` - Optional. The dot-separated name for the field.

Unlocks a field making it exempt from the `SecurityComponent` field hashing. This also allows the fields to be manipulated by JavaScript. The `$name` parameter should be the entity property name for the field:

```
$this->Form->unlockField('id');
```

`Cake\View\Helper\FormHelper::secure`(*array $fields = []*, *array $secureAttributes = []*)

- `$fields` - Optional. An array containing the list of fields to use when generating the hash. If not provided, then `$this->fields` will be used.
- `$secureAttributes` - Optional. An array of HTML attributes to be passed into the generated hidden input elements.

Generates a hidden `input` field with a security hash based on the fields used in the form or an empty string when secured forms are not in use. If `$secureAttributes` is set, these HTML attributes will be merged into the hidden input tags generated for the SecurityComponent. This is especially useful to set HTML5 attributes like `'form'`.

Html

class `Cake\View\Helper\HtmlHelper`(*View $view*, *array $config = []*)

The role of the HtmlHelper in CakePHP is to make HTML-related options easier, faster, and more resilient to change. Using this helper will enable your application to be more light on its feet, and more flexible on where it is placed in relation to the root of a domain.

Many HtmlHelper methods include a `$attributes` parameter, that allow you to tack on any extra attributes on your tags. Here are a few examples of how to use the `$attributes` parameter:

```
Desired attributes: <tag class="someClass" />
Array parameter: ['class' => 'someClass']

Desired attributes: <tag name="foo" value="bar" />
Array parameter:  ['name' => 'foo', 'value' => 'bar']
```

Inserting Well-Formatted Elements

The most important task the HtmlHelper accomplishes is creating well formed markup. This section will cover some of the methods of the HtmlHelper and how to use them.

Creating Charset Tags

`Cake\View\Helper\HtmlHelper::charset(`*$charset=null*`)`

Used to create a meta tag specifying the document's character. The default value is UTF-8. An example use:

```
echo $this->Html->charset();
```

Will output:

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
```

Alternatively,

```
echo $this->Html->charset('ISO-8859-1');
```

Will output:

```
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
```

Linking to CSS Files

`Cake\View\Helper\HtmlHelper::css(`*mixed $path*, *array $options = []*`)`

Creates a link(s) to a CSS style-sheet. If the `block` option is set to `true`, the link tags are added to the `css` block which you can print inside the head tag of the document.

You can use the `block` option to control which block the link element will be appended to. By default it will append to the `css` block.

If key 'rel' in `$options` array is set to 'import' the stylesheet will be imported.

This method of CSS inclusion assumes that the CSS file specified resides inside the **webroot/css** directory if path doesn't start with a '/'.

```
echo $this->Html->css('forms');
```

Will output:

```
<link rel="stylesheet" href="/css/forms.css" />
```

The first parameter can be an array to include multiple files.

```
echo $this->Html->css(['forms', 'tables', 'menu']);
```

Will output:

```
<link rel="stylesheet" href="/css/forms.css" />
<link rel="stylesheet" href="/css/tables.css" />
<link rel="stylesheet" href="/css/menu.css" />
```

You can include CSS files from any loaded plugin using *plugin syntax*. To include **plugins/DebugKit/webroot/css/toolbar.css** you could use the following:

```
echo $this->Html->css('DebugKit.toolbar.css');
```

If you want to include a CSS file which shares a name with a loaded plugin you can do the following. For example if you had a `Blog` plugin, and also wanted to include **webroot/css/Blog.common.css**, you would:

```
echo $this->Html->css('Blog.common.css', ['plugin' => false]);
```

Creating CSS Programatically

`Cake\View\Helper\HtmlHelper::style`(*array $data*, *boolean $oneline = true*)

Builds CSS style definitions based on the keys and values of the array passed to the method. Especially handy if your CSS file is dynamic.

```
echo $this->Html->style([
    'background' => '#633',
    'border-bottom' => '1px solid #000',
    'padding' => '10px'
]);
```

Will output:

```
background:#633; border-bottom:1px solid #000; padding:10px;
```

Creating meta Tags

`Cake\View\Helper\HtmlHelper::meta`(*string|array $type*, *string $url = null*, *array $options = []*)

This method is handy for linking to external resources like RSS/Atom feeds and favicons. Like css(), you can specify whether or not you'd like this tag to appear inline or appended to the `meta` block by setting the 'block' key in the $attributes parameter to `true`, ie - `['block' => true]`.

If you set the "type" attribute using the $attributes parameter, CakePHP contains a few shortcuts:

type	translated value
html	text/html
rss	application/rss+xml
atom	application/atom+xml
icon	image/x-icon

```
<?= $this->Html->meta(
    'favicon.ico',
    '/favicon.ico',
    ['type' => 'icon']
);
?>
// Output (line breaks added)
```

(continues on next page)

(continued from previous page)

```
// Note: The helper code makes two meta tags to  ensure the
// icon is downloaded by both newer and older browsers
// which require different rel attribute values.
<link
    href="/subdir/favicon.ico"
    type="image/x-icon"
    rel="icon"
/>
<link
    href="/subdir/favicon.ico"
    type="image/x-icon"
    rel="shortcut icon"
/>

<?= $this->Html->meta(
    'Comments',
    '/comments/index.rss',
    ['type' => 'rss']
);
?>
// Output (line breaks added)
<link
    href="http://example.com/comments/index.rss"
    title="Comments"
    type="application/rss+xml"
    rel="alternate"
/>
```

This method can also be used to add the meta keywords and descriptions. Example:

```
<?= $this->Html->meta(
    'keywords',
    'enter any meta keyword here'
);
?>
// Output
<meta name="keywords" content="enter any meta keyword here" />

<?= $this->Html->meta(
    'description',
    'enter any meta description here'
);
?>
// Output
<meta name="description" content="enter any meta description here" />
```

In addition to making predefined meta tags, you can create link elements:

```
<?= $this->Html->meta([
    'link' => 'http://example.com/manifest',
    'rel' => 'manifest'
]);
```

(continues on next page)

(continued from previous page)

```
?>
// Output
<link href="http://example.com/manifest" rel="manifest"/>
```

Any attributes provided to meta() when called this way will be added to the generated link tag.

Linking to Images

`Cake\View\Helper\HtmlHelper::image`(*string $path*, *array $options = []*)

Creates a formatted image tag. The path supplied should be relative to **webroot/img/**.

```
echo $this->Html->image('cake_logo.png', ['alt' => 'CakePHP']);
```

Will output:

```
<img src="/img/cake_logo.png" alt="CakePHP" />
```

To create an image link specify the link destination using the `url` option in `$attributes`.

```
echo $this->Html->image("recipes/6.jpg", [
    "alt" => "Brownies",
    'url' => ['controller' => 'Recipes', 'action' => 'view', 6]
]);
```

Will output:

```
<a href="/recipes/view/6">
    <img src="/img/recipes/6.jpg" alt="Brownies" />
</a>
```

If you are creating images in emails, or want absolute paths to images you can use the `fullBase` option:

```
echo $this->Html->image("logo.png", ['fullBase' => true]);
```

Will output:

```
<img src="http://example.com/img/logo.jpg" alt="" />
```

You can include image files from any loaded plugin using *plugin syntax*. To include **plugins/DebugKit/webroot/img/icon.png** You could use the following:

```
echo $this->Html->image('DebugKit.icon.png');
```

If you want to include an image file which shares a name with a loaded plugin you can do the following. For example if you had a `Blog` plugin, and also wanted to include **webroot/img/Blog.icon.png**, you would:

```
echo $this->Html->image('Blog.icon.png', ['plugin' => false]);
```

If you would like the prefix of the URL to not be `/img`, you can override this setting by specifying the prefix in the `$options` array

```
echo $this->Html->image("logo.png", ['pathPrefix' => '']);
```

Will output:

```
<img src="logo.jpg" alt="" />
```

Creating Links

Cake\View\Helper\HtmlHelper::**link**(*$title*, *$url = null*, *array $options = []*)

General purpose method for creating HTML links. Use `$options` to specify attributes for the element and whether or not the `$title` should be escaped.

```
echo $this->Html->link(
    'Enter',
    '/pages/home',
    ['class' => 'button', 'target' => '_blank']
);
```

Will output:

```
<a href="/pages/home" class="button" target="_blank">Enter</a>
```

Use `'_full'=>true` option for absolute URLs:

```
echo $this->Html->link(
    'Dashboard',
    ['controller' => 'Dashboards', 'action' => 'index', '_full' => true]
);
```

Will output:

```
<a href="http://www.yourdomain.com/dashboards/index">Dashboard</a>
```

Specify `confirm` key in options to display a JavaScript `confirm()` dialog:

```
echo $this->Html->link(
    'Delete',
    ['controller' => 'Recipes', 'action' => 'delete', 6],
    ['confirm' => 'Are you sure you wish to delete this recipe?']
);
```

Will output:

```
<a href="/recipes/delete/6"
    onclick="return confirm(
        'Are you sure you wish to delete this recipe?'
    );">
    Delete
</a>
```

Query strings can also be created with `link()`.

```
echo $this->Html->link('View image', [
    'controller' => 'Images',
    'action' => 'view',
```

(continues on next page)

(continued from previous page)

```
    1,
    '?' => ['height' => 400, 'width' => 500]
]);
```

Will output:

```
<a href="/images/view/1?height=400&width=500">View image</a>
```

HTML special characters in `$title` will be converted to HTML entities. To disable this conversion, set the escape option to `false` in the `$options` array.

```
echo $this->Html->link(
    $this->Html->image("recipes/6.jpg", ["alt" => "Brownies"]),
    "recipes/view/6",
    ['escape' => false]
);
```

Will output:

```
<a href="/recipes/view/6">
    <img src="/img/recipes/6.jpg" alt="Brownies" />
</a>
```

Setting `escape` to `false` will also disable escaping of attributes of the link. You can use the option `escapeTitle` to disable just escaping of title and not the attributes.

```
echo $this->Html->link(
    $this->Html->image('recipes/6.jpg', ['alt' => 'Brownies']),
    'recipes/view/6',
    ['escapeTitle' => false, 'title' => 'hi "howdy"']
);
```

Will output:

```
<a href="/recipes/view/6" title="hi &quot;howdy&quot;">
    <img src="/img/recipes/6.jpg" alt="Brownies" />
</a>
```

Also check *Cake\View\Helper\UrlHelper::build()* method for more examples of different types of URLs.

Cake\View\Helper\HtmlHelper::**linkFromPath**(*string $title*, *string $path*, *array $params = []*, *array $options = []*)

If you want to use route path strings, you can do that using this method:

```
echo $this->Html->linkFromPath('Index', 'Articles::index');
// outputs: <a href="/articles">Index</a>

echo $this->Html->linkFromPath('View', 'MyBackend.Admin/Articles::view', [3]);
// outputs: <a href="/admin/my-backend/articles/view/3">View</a>
```

Linking to Videos and Audio Files

Cake\View\Helper\HtmlHelper::**media**(*string|array $path*, *array $options*)

Options:

- `type` Type of media element to generate, valid values are "audio" or "video". If type is not provided media type is guessed based on file's mime type.
- `text` Text to include inside the video tag
- `pathPrefix` Path prefix to use for relative URLs, defaults to 'files/'
- `fullBase` If provided the src attribute will get a full address including domain name

Returns a formatted audio/video tag:

```
<?= $this->Html->media('audio.mp3') ?>

// Output
<audio src="/files/audio.mp3"></audio>

<?= $this->Html->media('video.mp4', [
    'fullBase' => true,
    'text' => 'Fallback text'
]) ?>

// Output
<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']
) ?>

// Output
<video autoplay="autoplay">
    <source src="/files/video.mp4" type="video/mp4"/>
    <source src="/files/video.ogg" type="video/ogg;
        codecs='theora, vorbis'"/>
</video>
```

Linking to Javascript Files

Cake\View\Helper\HtmlHelper::**script**(*mixed $url*, *mixed $options*)

Include a script file(s), contained either locally or as a remote URL.

By default, script tags are added to the document inline. If you override this by setting `$options['block']` to `true`, the script tags will instead be added to the `script` block which you can print elsewhere in the document. If you wish to override which block name is used, you can do so by setting `$options['block']`.

`$options['once']` controls whether or not you want to include this script once per request or more than once. This defaults to `true`.

You can use $options to set additional properties to the generated script tag. If an array of script tags is used, the attributes will be applied to all of the generated script tags.

This method of JavaScript file inclusion assumes that the JavaScript file specified resides inside the **webroot/js** directory:

```
echo $this->Html->script('scripts');
```

Will output:

```
<script src="/js/scripts.js"></script>
```

You can link to files with absolute paths as well to link files that are not in **webroot/js**:

```
echo $this->Html->script('/otherdir/script_file');
```

You can also link to a remote URL:

```
echo $this->Html->script('https://code.jquery.com/jquery.min.js');
```

Will output:

```
<script src="https://code.jquery.com/jquery.min.js"></script>
```

The first parameter can be an array to include multiple files.

```
echo $this->Html->script(['jquery', 'wysiwyg', 'scripts']);
```

Will output:

```
<script src="/js/jquery.js"></script>
<script src="/js/wysiwyg.js"></script>
<script src="/js/scripts.js"></script>
```

You can append the script tag to a specific block using the `block` option:

```
$this->Html->script('wysiwyg', ['block' => 'scriptBottom']);
```

In your layout you can output all the script tags added to 'scriptBottom':

```
echo $this->fetch('scriptBottom');
```

You can include script files from any loaded plugin using *plugin syntax*. To include **plugins/DebugKit/webroot/js/toolbar.js** You could use the following:

```
echo $this->Html->script('DebugKit.toolbar.js');
```

If you want to include a script file which shares a name with a loaded plugin you can do the following. For example if you had a `Blog` plugin, and also wanted to include **webroot/js/Blog.plugins.js**, you would:

```
echo $this->Html->script('Blog.plugins.js', ['plugin' => false]);
```

Creating Inline Javascript Blocks

Cake\View\Helper\HtmlHelper::**scriptBlock**(*$code*, *$options = []*)

To generate Javascript blocks from PHP view code, you can use one of the script block methods. Scripts can either be output in place, or buffered into a block:

```
// Define a script block all at once, with the defer attribute.
$this->Html->scriptBlock('alert("hi")', ['defer' => true]);

// Buffer a script block to be output later.
$this->Html->scriptBlock('alert("hi")', ['block' => true]);
```

Cake\View\Helper\HtmlHelper::**scriptStart**(*$options = []*)

Cake\View\Helper\HtmlHelper::**scriptEnd**()

You can use the `scriptStart()` method to create a capturing block that will output into a `<script>` tag. Captured script snippets can be output inline, or buffered into a block:

```
// Append into the 'script' block.
$this->Html->scriptStart(['block' => true]);
echo "alert('I am in the JavaScript');";
$this->Html->scriptEnd();
```

Once you have buffered javascript, you can output it as you would any other *View Block*:

```
// In your layout
echo $this->fetch('script');
```

Creating Nested Lists

Cake\View\Helper\HtmlHelper::**nestedList**(*array $list*, *array $options = []*, *array $itemOptions = []*)

Build a nested list (UL/OL) out of an associative array:

```
$list = [
    'Languages' => [
        'English' => [
            'American',
            'Canadian',
            'British',
        ],
        'Spanish',
        'German',
    ]
];
echo $this->Html->nestedList($list);
```

Output:

```
// Output (minus the whitespace)
<ul>
```

(continues on next page)

(continued from previous page)

```
    <li>Languages
        <ul>
            <li>English
                <ul>
                    <li>American</li>
                    <li>Canadian</li>
                    <li>British</li>
                </ul>
            </li>
            <li>Spanish</li>
            <li>German</li>
        </ul>
    </li>
</ul>
```

Creating Table Headings

Cake\View\Helper\HtmlHelper::**tableHeaders**(*array $names*, *array $trOptions = null*, *array $thOptions = null*)

Creates a row of table header cells to be placed inside of <table> tags.

```
echo $this->Html->tableHeaders(['Date', 'Title', 'Active']);
```

Output:

```
<tr>
    <th>Date</th>
    <th>Title</th>
    <th>Active</th>
</tr>
```

```
echo $this->Html->tableHeaders(
    ['Date', 'Title','Active'],
    ['class' => 'status'],
    ['class' => 'product_table']
);
```

Output:

```
<tr class="status">
     <th class="product_table">Date</th>
     <th class="product_table">Title</th>
     <th class="product_table">Active</th>
</tr>
```

You can set attributes per column, these are used instead of the defaults provided in the `$thOptions`:

```
echo $this->Html->tableHeaders([
    'id',
    ['Name' => ['class' => 'highlight']],
```

(continues on next page)

(continued from previous page)

```
    ['Date' => ['class' => 'sortable']]
]);
```

Output:

```
<tr>
    <th>id</th>
    <th class="highlight">Name</th>
    <th class="sortable">Date</th>
</tr>
```

Creating Table Cells

Cake\View\Helper\HtmlHelper::**tableCells**(*array $data*, *array $oddTrOptions = null*, *array $evenTrOptions = null*, *$useCount = false*, *$continueOddEven = true*)

Creates table cells, in rows, assigning <tr> attributes differently for odd- and even-numbered rows. Wrap a single table cell within an [] for specific <td>-attributes.

```
echo $this->Html->tableCells([
    ['Jul 7th, 2007', 'Best Brownies', 'Yes'],
    ['Jun 21st, 2007', 'Smart Cookies', 'Yes'],
    ['Aug 1st, 2006', 'Anti-Java Cake', 'No'],
]);
```

Output:

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

Output:

```
<tr>
    <td>
        Jul 7th, 2007
    </td>
    <td class="highlight">
        Best Brownies
    </td>
    <td>
        Yes
    </td>
</tr>
<tr>
```

(continues on next page)

(continued from previous page)

```
    <td>
        Jun 21st, 2007
    </td>
    <td>
        Smart Cookies
    </td>
    <td>
        Yes
    </td>
</tr>
<tr>
    <td>
        Aug 1st, 2006
    </td>
    <td>
        Anti-Java Cake
    </td>
    <td id="special">
        No
    </td>
</tr>
```

```
echo $this->Html->tableCells(
    [
        ['Red', 'Apple'],
        ['Orange', 'Orange'],
        ['Yellow', 'Banana'],
    ],
    ['class' => 'darker']
);
```

Output:

```
<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>
```

Changing the Tags Output by HtmlHelper

Cake\View\Helper\HtmlHelper::**setTemplates**(*array $templates*)

Load an array of templates to add/replace templates:

```
// Load specific templates.
$this->Html->setTemplates([
    'javascriptlink' => '<script src="{{url}}" type="text/javascript"{{attrs}}></script>'
]);
```

You can load a configuration file containing templates using the templater directly:

```
// Load a configuration file with templates.
$this->Html->templater()->load('my_tags');
```

When loading files of templates, your file should look like:

```
<?php
return [
    'javascriptlink' => '<script src="{{url}}" type="text/javascript"{{attrs}}></script>'
];
```

Warning: Template strings containing a percentage sign (%) need special attention, you should prefix this character with another percentage so it looks like %%. The reason is that internally templates are compiled to be used with `sprintf()`. Example: `<div style="width:{{size}}%%">{{content}}</div>`

Number

`class` Cake\View\Helper**NumberHelper**(*View $view*, *array $config = []*)

The NumberHelper contains convenient methods that enable display numbers in common formats in your views. These methods include ways to format currency, percentages, data sizes, format numbers to specific precisions and also to give you more flexibility with formatting numbers.

All of these functions return the formatted number; they do not automatically echo the output into the view.

Formatting Currency Values

Cake\View\Helper\NumberHelper::**currency**(*mixed $value*, *string $currency = null*, *array $options = []*)

This method is used to display a number in common currency formats (EUR, GBP, USD), based on the 3-letter ISO 4217 currency code. Usage in a view looks like:

```
// Called as NumberHelper
echo $this->Number->currency($value, $currency);

// Called as Number
echo Number::currency($value, $currency);
```

The first parameter, `$value`, should be a floating point number that represents the amount of money you are expressing. The second parameter is a string used to choose a predefined currency formatting scheme:

$currency	1234.56, formatted by currency type
EUR	€1.234,56
GBP	£1,234.56
USD	$1,234.56

The third parameter is an array of options for further defining the output. The following options are available:

Option	Description
before	Text to display before the rendered number.
after	Text to display after the rendered number.
zero	The text to use for zero values; can be a string or a number. ie. 0, 'Free!'.
places	Number of decimal places to use, ie. 2
precision	Maximal number of decimal places to use, ie. 2
locale	The locale name to use for formatting number, ie. "fr_FR".
fractionSymbol	String to use for fraction numbers, ie. ' cents'.
fractionPosition	Either 'before' or 'after' to place the fraction symbol.
pattern	An ICU number pattern to use for formatting the number ie. #,###.00
useIntlCode	Set to `true` to replace the currency symbol with the international currency code.

If `$currency` value is `null`, the default currency will be retrieved from `Cake\I18n\Number::defaultCurrency()`. To format currencies in an accounting format you should set the currency format:

```
Number::setDefaultCurrencyFormat(Number::FORMAT_CURRENCY_ACCOUNTING);
```

Setting the Default Currency

`Cake\View\Helper\NumberHelper::`**`setDefaultCurrency`**(*$currency*)

Setter for the default currency. This removes the need to always pass the currency to *`Cake\I18n\Number::currency()`* and change all currency outputs by setting other default. If `$currency` is set to `null`, it will clear the currently stored value.

Getting the Default Currency

`Cake\View\Helper\NumberHelper::`**`getDefaultCurrency`**()

Getter for the default currency. If default currency was set earlier using `setDefaultCurrency()`, then that value will be returned. By default, it will retrieve the `intl.default_locale` ini value if set and `'en_US'` if not.

Formatting Floating Point Numbers

`Cake\View\Helper\NumberHelper::`**`precision`**(*float $value*, *int $precision = 3*, *array $options = []*)

This method displays a number with the specified amount of precision (decimal places). It will round in order to maintain the level of precision defined.

```
// Called as NumberHelper
echo $this->Number->precision(456.91873645, 2);

// Outputs
456.92

// Called as Number
echo Number::precision(456.91873645, 2);
```

Formatting Percentages

`Cake\View\Helper\NumberHelper::toPercentage(mixed $value, int $precision = 2, array $options = [])`

Option	Description
multiply	Boolean to indicate whether the value has to be multiplied by 100. Useful for decimal percentages.

Like *Cake\I18n\Number::precision()*, this method formats a number according to the supplied precision (where numbers are rounded to meet the given precision). This method also expresses the number as a percentage and appends the output with a percent sign.

```
// Called as NumberHelper. Output: 45.69%
echo $this->Number->toPercentage(45.691873645);

// Called as Number. Output: 45.69%
echo Number::toPercentage(45.691873645);

// Called with multiply. Output: 45.7%
echo Number::toPercentage(0.45691, 1, [
    'multiply' => true
]);
```

Interacting with Human Readable Values

`Cake\View\Helper\NumberHelper::toReadableSize(string $size)`

This method formats data sizes in human readable forms. It provides a shortcut way to convert bytes to KB, MB, GB, and TB. The size is displayed with a two-digit precision level, according to the size of data supplied (i.e. higher sizes are expressed in larger terms):

```
// Called as 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

// Called as 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
```

Formatting Numbers

Cake\View\Helper\NumberHelper::**format**(*mixed $value*, *array $options = []*)

This method gives you much more control over the formatting of numbers for use in your views (and is used as the main method by most of the other NumberHelper methods). Using this method might looks like:

```
// Called as NumberHelper
$this->Number->format($value, $options);

// Called as Number
Number::format($value, $options);
```

The `$value` parameter is the number that you are planning on formatting for output. With no `$options` supplied, the number 1236.334 would output as 1,236. Note that the default precision is zero decimal places.

The `$options` parameter is where the real magic for this method resides.

- If you pass an integer then this becomes the amount of precision or places for the function.
- If you pass an associated array, you can use the following keys:

Option	Description
places	Number of decimal places to use, ie. 2
precision	Maximum number of decimal places to use, ie. 2
pattern	An ICU number pattern to use for formatting the number ie. #,###.00
locale	The locale name to use for formatting number, ie. “fr_FR”.
before	Text to display before the rendered number.
after	Text to display after the rendered number.

Example:

```
// Called as NumberHelper
echo $this->Number->format('123456.7890', [
    'places' => 2,
    'before' => '¥ ',
    'after' => ' !'
]);
// Output '¥ 123,456.79 !'

echo $this->Number->format('123456.7890', [
    'locale' => 'fr_FR'
]);
// Output '123 456,79 !'

// Called as Number
echo Number::format('123456.7890', [
    'places' => 2,
    'before' => '¥ ',
    'after' => ' !'
]);
// Output '¥ 123,456.79 !'

echo Number::format('123456.7890', [
```

(continues on next page)

(continued from previous page)

```
    'locale' => 'fr_FR'
]);
// Output '123 456,79 !'
```

Cake\View\Helper\NumberHelper::**ordinal**(*mixed $value*, *array $options = []*)

This method will output an ordinal number.

Examples:

```
echo Number::ordinal(1);
// Output '1st'

echo Number::ordinal(2);
// Output '2nd'

echo Number::ordinal(2, [
    'locale' => 'fr_FR'
]);
// Output '2e'

echo Number::ordinal(410);
// Output '410th'
```

Format Differences

Cake\View\Helper\NumberHelper::**formatDelta**(*mixed $value*, *array $options = []*)

This method displays differences in value as a signed number:

```
// Called as NumberHelper
$this->Number->formatDelta($value, $options);

// Called as Number
Number::formatDelta($value, $options);
```

The `$value` parameter is the number that you are planning on formatting for output. With no `$options` supplied, the number 1236.334 would output as 1,236. Note that the default precision is zero decimal places.

The `$options` parameter takes the same keys as `Number::format()` itself:

Option	Description
places	Number of decimal places to use, ie. 2
precision	Maximum number of decimal places to use, ie. 2
locale	The locale name to use for formatting number, ie. “fr_FR”.
before	Text to display before the rendered number.
after	Text to display after the rendered number.

Example:

```php
// Called as NumberHelper
echo $this->Number->formatDelta('123456.7890', [
    'places' => 2,
    'before' => '[',
    'after' => ']'
]);
// Output '[+123,456.79]'

// Called as Number
echo Number::formatDelta('123456.7890', [
    'places' => 2,
    'before' => '[',
    'after' => ']'
]);
// Output '[+123,456.79]'
```

Warning: All symbols are UTF-8.

Paginator

class Cake\View\Helper**PaginatorHelper**(*View $view*, *array $config = []*)

The PaginatorHelper is used to output pagination controls such as page numbers and next/previous links.

See also *Pagination* for information on how to create paginated datasets and do paginated queries.

Setting the paginated resultset

Cake\View\Helper\PaginatorHelper::**setPaginated**(*$paginated*, *$options*)

By default the helper uses the first instance of `Cake\Datasource\Paging\PaginatedInterface` it finds in the view variables. (Generally the result of `Controller::paginate()`).

You can use `PaginatorHelper::setPaginated()` to explicitly set the paginated resultset that the helper should use.

PaginatorHelper Templates

Internally PaginatorHelper uses a series of simple HTML templates to generate markup. You can modify these templates to customize the HTML generated by the PaginatorHelper.

Templates use `{{var}}` style placeholders. It is important to not add any spaces around the `{{}}` or the replacements will not work.

Loading Templates from a File

When adding the PaginatorHelper in your controller, you can define the 'templates' setting to define a template file to load. This allows you to customize multiple templates and keep your code DRY:

```
// In your AppView.php
public function initialize(): void
{
    ...
    $this->loadHelper('Paginator', ['templates' => 'paginator-templates']);
}
```

This will load the file located at **config/paginator-templates.php**. See the example below for how the file should look like. You can also load templates from a plugin using *plugin syntax*:

```
// In your AppView.php
public function initialize(): void
{
    ...
    $this->loadHelper('Paginator', ['templates' => 'MyPlugin.paginator-templates']);
}
```

Whether your templates are in the primary application or a plugin, your templates file should look something like:

```
return [
    'number' => '<a href="{{url}}">{{text}}</a>',
];
```

Changing Templates at Run-time

Cake\View\Helper\PaginatorHelper::**setTemplates**(*$templates*)

This method allows you to change the templates used by PaginatorHelper at runtime. This can be useful when you want to customize templates for a particular method call:

```
// Read the current template value.
$result = $this->Paginator->getTemplates('number');

// Change a template
$this->Paginator->setTemplates([
    'number' => '<em><a href="{{url}}">{{text}}</a></em>'
]);
```

Warning: Template strings containing a percentage sign (%) need special attention, you should prefix this character with another percentage so it looks like %%. The reason is that internally templates are compiled to be used with `sprintf()`. Example: ' {{content}} '

Template Names

PaginatorHelper uses the following templates:

- `nextActive` The active state for a link generated by next().
- `nextDisabled` The disabled state for next().
- `prevActive` The active state for a link generated by prev().
- `prevDisabled` The disabled state for prev()
- `counterRange` The template counter() uses when format == range.
- `counterPages` The template counter() uses when format == pages.
- `first` The template used for a link generated by first().
- `last` The template used for a link generated by last()
- `number` The template used for a link generated by numbers().
- `current` The template used for the current page.
- `ellipsis` The template used for ellipses generated by numbers().
- `sort` The template for a sort link with no direction.
- `sortAsc` The template for a sort link with an ascending direction.
- `sortDesc` The template for a sort link with a descending direction.

Creating Sort Links

`Cake\View\Helper\PaginatorHelper::sort`(*$key*, *$title = null*, *$options = []*)

Parameters

- **$key** (`string`) – The name of the column that the recordset should be sorted.
- **$title** (`string`) – Title for the link. If $title is null, $key will be used converted to "Title Case" format and used as the title.
- **$options** (`array`) – Options for sorting link.

Generates a sorting link. Sets querystring parameters for the sort and direction. Links will default to sorting by asc. After the first click, links generated with `sort()` will handle direction switching automatically. If the resultset is sorted 'asc' by the specified key the returned link will sort by 'desc'. Uses the `sort`, `sortAsc`, `sortDesc`, `sortAscLocked` and `sortDescLocked` templates.

Accepted keys for `$options`:

- `escape` Whether you want the contents HTML entity encoded, defaults to `true`.
- `direction` The default direction to use when this link isn't active.
- `lock` Lock direction. Will only use the default direction then, defaults to `false`.

Assuming you are paginating some posts, and are on page one:

```
echo $this->Paginator->sort('user_id');
```

Output:

```
<a href="/posts/index?page=1&amp;sort=user_id&amp;direction=asc">User Id</a>
```

You can use the title parameter to create custom text for your link:

```
echo $this->Paginator->sort('user_id', 'User account');
```

Output:

```
<a href="/posts/index?page=1&amp;sort=user_id&amp;direction=asc">User account</a>
```

If you are using HTML like images in your links remember to set escaping off:

```
echo $this->Paginator->sort(
  'user_id',
  '<em>User account</em>',
  ['escape' => false]
);
```

Output:

```
<a href="/posts/index?page=1&amp;sort=user_id&amp;direction=asc"><em>User account</em></
↪a>
```

The direction option can be used to set the default direction for a link. Once a link is active, it will automatically switch directions like normal:

```
echo $this->Paginator->sort('user_id', null, ['direction' => 'desc']);
```

Output:

```
<a href="/posts/index?page=1&amp;sort=user_id&amp;direction=desc">User Id</a>
```

The lock option can be used to lock sorting into the specified direction:

```
echo $this->Paginator->sort('user_id', null, ['direction' => 'asc', 'lock' => true]);
```

`Cake\View\Helper\PaginatorHelper::sortDir`(*string $model = null*, *mixed $options = []*)

Gets the current direction the recordset is sorted.

`Cake\View\Helper\PaginatorHelper::sortKey`(*string $model = null*, *mixed $options = []*)

Gets the current key by which the recordset is sorted.

Creating Page Number Links

`Cake\View\Helper\PaginatorHelper::numbers`(*$options = []*)

Returns a set of numbers for the paged result set. Uses a modulus to decide how many numbers to show on each side of the current page By default 8 links on either side of the current page will be created if those pages exist. Links will not be generated for pages that do not exist. The current page is also not a link. The `number`, `current` and `ellipsis` templates will be used.

Supported options are:

- `before` Content to be inserted before the numbers.
- `after` Content to be inserted after the numbers.

- `modulus` how many numbers to include on either side of the current page, defaults to 8.
- `first` Whether you want first links generated, set to an integer to define the number of 'first' links to generate. Defaults to `false`. If a string is set a link to the first page will be generated with the value as the title:

```
echo $this->Paginator->numbers(['first' => 'First page']);
```

- `last` Whether you want last links generated, set to an integer to define the number of 'last' links to generate. Defaults to `false`. Follows the same logic as the `first` option. There is a *`last()`* method to be used separately as well if you wish.

While this method allows a lot of customization for its output. It is also ok to just call the method without any parameters.

```
echo $this->Paginator->numbers();
```

Using the first and last options you can create links to the beginning and end of the page set. The following would create a set of page links that include links to the first 2 and last 2 pages in the paged results:

```
echo $this->Paginator->numbers(['first' => 2, 'last' => 2]);
```

Creating Jump Links

In addition to generating links that go directly to specific page numbers, you'll often want links that go to the previous and next links, first and last pages in the paged data set.

`Cake\View\Helper\PaginatorHelper::`**`prev`**(*$title = '<< Previous'*, *$options = []*)

Parameters

- **`$title`** (`string`) – Title for the link.
- **`$options`** (`mixed`) – Options for pagination link.

Generates a link to the previous page in a set of paged records. Uses the `prevActive` and `prevDisabled` templates.

`$options` supports the following keys:

- `escape` Whether you want the contents HTML entity encoded, defaults to `true`.
- `disabledTitle` The text to use when the link is disabled. Defaults to the `$title` parameter.

A simple example would be:

```
echo $this->Paginator->prev(' << ' . __('previous'));
```

If you were currently on the second page of posts, you would get the following:

```
<li class="prev">
    <a rel="prev" href="/posts/index?page=1&amp;sort=title&amp;order=desc">
        &lt;&lt; previous
    </a>
</li>
```

If there were no previous pages you would get:

```
<li class="prev disabled"><a href="" onclick="return false;">&lt;&lt; previous</a></
↪li>
```

To change the templates used by this method see *PaginatorHelper Templates*.

Cake\View\Helper\PaginatorHelper::**next**(*$title = 'Next >>'*, *$options = []*)

This method is identical to *prev()* with a few exceptions. It creates links pointing to the next page instead of the previous one. It also uses `next` as the rel attribute value instead of `prev`. Uses the `nextActive` and `nextDisabled` templates.

Cake\View\Helper\PaginatorHelper::**first**(*$first = '<< first'*, *$options = []*)

Returns a first or set of numbers for the first pages. If a string is given, then only a link to the first page with the provided text will be created:

```
echo $this->Paginator->first('< first');
```

The above creates a single link for the first page. Will output nothing if you are on the first page. You can also use an integer to indicate how many first paging links you want generated:

```
echo $this->Paginator->first(3);
```

The above will create links for the first 3 pages, once you get to the third or greater page. Prior to that nothing will be output. Uses the `first` template.

The options parameter accepts the following:

- `escape` Whether or not the text should be escaped. Set to `false` if your content contains HTML.

Cake\View\Helper\PaginatorHelper::**last**(*$last = 'last >>'*, *$options = []*)

This method works very much like the *first()* method. It has a few differences though. It will not generate any links if you are on the last page for a string values of `$last`. For an integer value of `$last` no links will be generated once the user is inside the range of last pages. Uses the `last` template.

Creating Header Link Tags

PaginatorHelper can be used to create pagination link tags in your page <head> elements:

```
// Create next/prev links for the current model.
echo $this->Paginator->meta();

// Create next/prev & first/last links for the current model.
echo $this->Paginator->meta(['first' => true, 'last' => true]);
```

Checking the Pagination State

Cake\View\Helper\PaginatorHelper::**current**()

Gets the current page of the recordset:

```
// Our URL is: /comments?page=3
echo $this->Paginator->current();
// Output is 3
```

Uses the `current` template.

Cake\View\Helper\PaginatorHelper::**hasNext**(*string $model = null*)

Returns `true` if the given result set is not at the last page.

Cake\View\Helper\PaginatorHelper::**hasPrev**()

Returns `true` if the given result set is not at the first page.

Cake\View\Helper\PaginatorHelper::**hasPage**(*int $page = 1*)

Returns `true` if the given result set has the page number given by `$page`.

Cake\View\Helper\PaginatorHelper::**total**()

Returns the total number of pages for the provided model.

Creating a Page Counter

Cake\View\Helper\PaginatorHelper::**counter**(*string $format = 'pages'*, *array $options = []*)

Returns a counter string for the paged result set. Using a provided format string and a number of options you can create localized and application specific indicators of where a user is in the paged data set. Uses the `counterRange`, and `counterPages` templates.

Supported formats are 'range', 'pages' and custom. Defaults to pages which would output like '1 of 10'. In the custom mode the supplied string is parsed and tokens are replaced with actual values. The available tokens are:

- `{{page}}` - the current page displayed.
- `{{pages}}` - total number of pages.
- `{{current}}` - current number of records being shown.
- `{{count}}` - the total number of records in the result set.
- `{{start}}` - number of the first record being displayed.
- `{{end}}` - number of the last record being displayed.
- `{{model}}` - The pluralized human form of the model name. If your model was 'RecipePage', `{{model}}` would be 'recipe pages'.

You could also supply only a string to the counter method using the tokens available. For example:

```
echo $this->Paginator->counter(
    'Page {{page}} of {{pages}}, showing {{current}} records out of
     {{count}} total, starting on record {{start}}, ending on {{end}}'
);
```

Setting 'format' to range would output like '1 - 3 of 13':

```
echo $this->Paginator->counter('range');
```

Generating Pagination URLs

Cake\View\Helper\PaginatorHelper::**generateUrl**(*array $options = []*, *?string $model = null*, *array $url = []*, *array $urlOptions = []*)

By default returns a full pagination URL string for use in non-standard contexts (i.e. JavaScript).

```
// Generates a URL similar to: /articles?sort=title&page=2
echo $this->Paginator->generateUrl(['sort' => 'title']);
```

(continues on next page)

(continued from previous page)

```
// Generates a URL for a different model
echo $this->Paginator->generateUrl(['sort' => 'title'], 'Comments');

// Generates a URL to a different controller.
echo $this->Paginator->generateUrl(
    ['sort' => 'title'],
    null,
    ['controller' => 'Comments']
);
```

Creating a Limit Selectbox Control

Cake\View\Helper\PaginatorHelper::**limitControl**(*array $limits = []*, *$default = null*, *array $options = []*)

Create a dropdown control that changes the `limit` query parameter:

```
// Use the defaults.
echo $this->Paginator->limitControl();

// Define which limit options you want.
echo $this->Paginator->limitControl([25 => 25, 50 => 50]);

// Custom limits and set the selected option
echo $this->Paginator->limitControl([25 => 25, 50 => 50], $user->perPage);
```

The generated form and control will automatically submit on change.

Configuring Pagination Options

Cake\View\Helper\PaginatorHelper::**options**(*$options = []*)

Sets all the options for the PaginatorHelper. Supported options are:

- `url` The URL of the paginating action.

  The option allows your to set/override any element for URLs generated by the helper:

  ```
  $this->Paginator->options([
      'url' => [
          'lang' => 'en',
          '?' => [
              'sort' => 'email',
              'direction' => 'desc',
              'page' => 6,
          ],
      ]
  ]);
  ```

  The example above adds the `en` route parameter to all links the helper will generate. It will also create links with specific sort, direction and page values. By default `PaginatorHelper` will merge in all of the current passed arguments and query string parameters.

- `escape` Defines if the title field for links should be HTML escaped. Defaults to `true`.

Example Usage

It's up to you to decide how to show records to the user, but most often this will be done inside HTML tables. The examples below assume a tabular layout, but the PaginatorHelper available in views doesn't always need to be restricted as such.

See the details on PaginatorHelper[122] in the API. As mentioned, the PaginatorHelper also offers sorting features which can be integrated into your table column headers:

```
<!-- templates/Posts/index.php -->
<table>
    <tr>
        <th><?= $this->Paginator->sort('id', 'ID') ?></th>
        <th><?= $this->Paginator->sort('title', 'Title') ?></th>
    </tr>
       <?php foreach ($recipes as $recipe): ?>
    <tr>
        <td><?= $recipe->id ?> </td>
        <td><?= h($recipe->title) ?> </td>
    </tr>
    <?php endforeach; ?>
</table>
```

The links output from the `sort()` method of the `PaginatorHelper` allow users to click on table headers to toggle the sorting of the data by a given field.

It is also possible to sort a column based on associations:

```
<table>
    <tr>
        <th><?= $this->Paginator->sort('title', 'Title') ?></th>
        <th><?= $this->Paginator->sort('Authors.name', 'Author') ?></th>
    </tr>
       <?php foreach ($recipes as $recipe): ?>
    <tr>
        <td><?= h($recipe->title) ?> </td>
        <td><?= h($recipe->name) ?> </td>
    </tr>
    <?php endforeach; ?>
</table>
```

Note: Sorting by columns in associated models requires setting these in the `PaginationComponent::paginate` property. Using the example above, the controller handling the pagination would need to set its `sortableFields` key as follows:

```
$this->paginate = [
    'sortableFields' => [
        'Posts.title',
        'Authors.name',
    ],
];
```

[122] https://api.cakephp.org/5.x/class-Cake.View.Helper.PaginatorHelper.html

For more information on using the `sortableFields` option, please see *Control which Fields Used for Ordering*.

The final ingredient to pagination display in views is the addition of page navigation, also supplied by the PaginationHelper:

```
// Shows the page numbers
<?= $this->Paginator->numbers() ?>

// Shows the next and previous links
<?= $this->Paginator->prev('« Previous') ?>
<?= $this->Paginator->next('Next »') ?>

// Prints X of Y, where X is current page and Y is number of pages
<?= $this->Paginator->counter() ?>
```

The wording output by the counter() method can also be customized using special markers:

```
<?= $this->Paginator->counter([
    'format' => 'Page {{page}} of {{pages}}, showing {{current}} records out of
             {{count}} total, starting on record {{start}}, ending on {{end}}'
]) ?>
```

Paginating Multiple Results

If you are *paginating multiple queries* you'll need to use `PaginatorHelper::setPaginated()` first before calling other methods of the helper, so that they generate expected output.

`PaginatorHelper` will automatically use the `scope` defined in when the query was paginated. To set additional URL parameters for multiple pagination you can include the scope names in `options()`:

```
$this->Paginator->options([
    'url' => [
        // Additional URL parameters for the 'articles' scope
        'articles' => [
            '?' => ['articles' => 'yes']
        ],
        // Additional URL parameters for the 'comments' scope
        'comments' => [
            'articleId' => 1234,
        ],
    ],
]);
```

Text

`class` Cake\View\Helper**TextHelper**(*View $view*, *array $config = []*)

The TextHelper contains methods to make text more usable and friendly in your views. It aids in enabling links, formatting URLs, creating excerpts of text around chosen words or phrases, highlighting key words in blocks of text, and gracefully truncating long stretches of text.

Linking Email addresses

Cake\View\Helper\TextHelper::**autoLinkEmails**(*string $text*, *array $options = []*)

Adds links to the well-formed email addresses in $text, according to any options defined in `$options` (see *HtmlHelper::link()*).

```
$myText = 'For more information regarding our world-famous ' .
    'pastries and desserts, contact info@example.com';
$linkedText = $this->Text->autoLinkEmails($myText);
```

Output:

```
For more information regarding our world-famous pastries and desserts,
contact <a href="mailto:info@example.com">info@example.com</a>
```

This method automatically escapes its input. Use the `escape` option to disable this if necessary.

Linking URLs

Cake\View\Helper\TextHelper::**autoLinkUrls**(*string $text*, *array $options = []*)

Same as `autoLinkEmails()`, only this method searches for strings that start with https, http, ftp, or nntp and links them appropriately.

This method automatically escapes its input. Use the `escape` option to disable this if necessary.

Linking Both URLs and Email Addresses

Cake\View\Helper\TextHelper::**autoLink**(*string $text*, *array $options = []*)

Performs the functionality in both `autoLinkUrls()` and `autoLinkEmails()` on the supplied `$text`. All URLs and emails are linked appropriately given the supplied `$options`.

This method automatically escapes its input. Use the `escape` option to disable this if necessary.

Converting Text into Paragraphs

Cake\View\Helper\TextHelper::**autoParagraph**(*string $text*)

Adds proper <p> around text where double-line returns are found, and
 where single-line returns are found.

```
$myText = 'For more information
regarding our world-famous pastries and desserts.

contact info@example.com';
$formattedText = $this->Text->autoParagraph($myText);
```

Output:

```
<p>For more information<br />
regarding our world-famous pastries and desserts.</p>
<p>contact info@example.com</p>
```

Highlighting Substrings

Cake\View\Helper\TextHelper::**highlight**(*string $haystack*, *string $needle*, *array $options = []*)

Highlights `$needle` in `$haystack` using the `$options['format']` string specified or a default string.

Options:

- `format` string - The piece of HTML with the phrase that will be highlighted
- `html` bool - If `true`, will ignore any HTML tags, ensuring that only the correct text is highlighted

Example:

```
// Called as TextHelper
echo $this->Text->highlight(
    $lastSentence,
    'using',
    ['format' => '<span class="highlight">\1</span>']
);

// Called as Text
use Cake\Utility\Text;

echo Text::highlight(
    $lastSentence,
    'using',
    ['format' => '<span class="highlight">\1</span>']
);
```

Output:

Removing Links

Cake\View\Helper\TextHelper::**stripLinks**(*$text*)

Strips the supplied `$text` of any HTML links.

Truncating Text

Cake\View\Helper\TextHelper::**truncate**(*string $text*, *int $length = 100*, *array $options*)

If `$text` is longer than `$length`, this method truncates it at `$length` and adds a suffix consisting of `'ellipsis'`, if defined. If `'exact'` is passed as `false`, the truncation will occur at the first whitespace after the point at which `$length` is exceeded. If `'html'` is passed as `true`, HTML tags will be respected and will not be cut off.

`$options` is used to pass all extra parameters, and has the following possible keys by default, all of which are optional:

```
[
    'ellipsis' => '...',
    'exact' => true,
    'html' => false
]
```

Example:

```
// Called as TextHelper
echo $this->Text->truncate(
    'The killer crept forward and tripped on the rug.',
    22,
    [
        'ellipsis' => '...',
        'exact' => false
    ]
);

// Called as Text
use Cake\Utility\Text;

echo Text::truncate(
    'The killer crept forward and tripped on the rug.',
    22,
    [
        'ellipsis' => '...',
        'exact' => false
    ]
);
```

Output:

```
The killer crept...
```

Truncating the Tail of a String

`Cake\View\Helper\TextHelper::tail`(*string $text*, *int $length = 100*, *array $options*)

If `$text` is longer than `$length`, this method removes an initial substring with length consisting of the difference and prepends a prefix consisting of `'ellipsis'`, if defined. If `'exact'` is passed as `false`, the truncation will occur at the first whitespace prior to the point at which truncation would otherwise take place.

`$options` is used to pass all extra parameters, and has the following possible keys by default, all of which are optional:

```
[
    'ellipsis' => '...',
    'exact' => true
]
```

Example:

```
$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'

// Called as TextHelper
echo $this->Text->tail(
    $sampleText,
    70,
    [
        'ellipsis' => '...',
        'exact' => false
    ]
);

// Called as Text
use Cake\Utility\Text;

echo Text::tail(
    $sampleText,
    70,
    [
        'ellipsis' => '...',
        'exact' => false
    ]
);
```

Output:

```
...a TV, a C# program that can divide by zero, death metal t-shirts
```

Extracting an Excerpt

Cake\View\Helper\TextHelper::**excerpt**(*string $haystack*, *string $needle*, *integer $radius=100*, *string $ellipsis="..."*)

Extracts an excerpt from `$haystack` surrounding the `$needle` with a number of characters on each side determined by `$radius`, and prefix/suffix with `$ellipsis`. This method is especially handy for search results. The query string or keywords can be shown within the resulting document.

```
// Called as TextHelper
echo $this->Text->excerpt($lastParagraph, 'method', 50, '...');

// Called as Text
use Cake\Utility\Text;

echo Text::excerpt($lastParagraph, 'method', 50, '...');
```

Output:

```
... by $radius, and prefix/suffix with $ellipsis. This method is especially
handy for search results. The query...
```

Converting an Array to Sentence Form

Cake\View\Helper\TextHelper::**toList**(*array $list*, *$and='and'*, *$separator=', '*)

Creates a comma-separated list where the last two items are joined with 'and':

```
$colors = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet'];

// Called as TextHelper
echo $this->Text->toList($colors);

// Called as Text
use Cake\Utility\Text;

echo Text::toList($colors);
```

Output:

```
red, orange, yellow, green, blue, indigo and violet
```

Time

class Cake\View\Helper**TimeHelper**(*View $view*, *array $config = []*)

The TimeHelper allows for the quick processing of time related information. The TimeHelper has two main tasks that it can perform:

1. It can format time strings.
2. It can test time.

Using the Helper

A common use of the TimeHelper is to offset the date and time to match a user's time zone. Lets use a forum as an example. Your forum has many users who may post messages at any time from any part of the world. A way to manage the time is to save all dates and times as GMT+0 or UTC. Uncomment the line `date_default_timezone_set('UTC');` in **config/bootstrap.php** to ensure your application's time zone is set to GMT+0.

Next add a time zone field to your users table and make the necessary modifications to allow your users to set their time zone. Now that we know the time zone of the logged in user we can correct the date and time on our posts using the TimeHelper:

```
echo $this->Time->format(
  $post->created,
  \IntlDateFormatter::FULL,
  null,
  $user->time_zone
);
// Will display 'Saturday, August 22, 2011 at 11:53:00 PM GMT'
// for a user in GMT+0. While displaying,
// 'Saturday, August 22, 2011 at 03:53 PM GMT-8:00'
// for a user in GMT-8
```

Most of TimeHelper's features are intended as backwards compatible interfaces for applications that are upgrading from older versions of CakePHP. Because the ORM returns *Cake\I18n\Time* instances for every `timestamp` and `datetime` column, you can use the methods there to do most tasks. For example, to read about the accepted formatting strings take a look at the Cake\I18n\Time::i18nFormat()[123] method.

Url

`class` Cake\View\Helper**UrlHelper**(*View $view*, *array $config = []*)

The UrlHelper helps you to generate URLs from your other helpers. It also gives you a single place to customize how URLs are generated by overriding the core helper with an application one. See the *Aliasing Helpers* section for how to do this.

Generating URLs

Cake\View\Helper\UrlHelper::**build**(*$url = null*, *array $options = []*)

Returns a URL pointing to a combination of controller and action. If `$url` is empty, it returns the `REQUEST_URI`, otherwise it generates the URL for the controller and action combo. If `fullBase` is `true`, the full base URL will be prepended to the result:

```
echo $this->Url->build([
    'controller' => 'Posts',
    'action' => 'view',
    'bar',
]);

// Output
/posts/view/bar
```

[123] https://api.cakephp.org/5.x/class-Cake.I18n.Time.html#i18nFormat()

Here are a few more usage examples:

URL with extension:

```
echo $this->Url->build([
    'controller' => 'Posts',
    'action' => 'list',
    '_ext' => 'rss',
]);

// Output
/posts/list.rss
```

URL with prefix:

```
echo $this->Url->build([
    'controller' => 'Posts',
    'action' => 'list',
    'prefix' => 'Admin',
]);

// Output
/admin/posts/list
```

URL (starting with '/') with the full base URL prepended:

```
echo $this->Url->build('/posts', ['fullBase' => true]);

// Output
http://somedomain.com/posts
```

URL with GET parameters and fragment anchor:

```
echo $this->Url->build([
    'controller' => 'Posts',
    'action' => 'search',
    '?' => ['foo' => 'bar'],
    '#' => 'first',
]);

// Output
/posts/search?foo=bar#first
```

The above example uses the ? special key for specifying query string parameters and # key for URL fragment.

URL for named route:

```
// Assuming a route is setup as a named route:
// $router->connect(
//     '/products/{slug}',
//     [
//         'controller' => 'Products',
//         'action' => 'view',
//     ],
//     [
```

(continues on next page)

(continued from previous page)

```
//         '_name' => 'product-page',
//     ]
// );

echo $this->Url->build(['_name' => 'product-page', 'slug' => 'i-m-slug']);
// Will result in:
/products/i-m-slug
```

The 2nd parameter allows you to define options controlling HTML escaping, and whether or not the base path should be added:

```
$this->Url->build('/posts', [
    'escape' => false,
    'fullBase' => true,
]);
```

Cake\View\Helper\UrlHelper::**buildFromPath**(*string $path*, *array $params = []*, *array $options = []*)

If you want to use route path strings, you can do that using this method:

```
echo $this->Url->buildFromPath('Articles::index');
// outputs: /articles

echo $this->Url->buildFromPath('MyBackend.Admin/Articles::view', [3]);
// outputs: /admin/my-backend/articles/view/3
```

URL with asset timestamp wrapped by a `<link rel="preload"/>`, here pre-loading a font. Note: The file must exist and `Configure::read('Asset.timestamp')` must return `true` or `'force'` for the timestamp to be appended:

```
echo $this->Html->meta([
    'rel' => 'preload',
    'href' => $this->Url->assetUrl(
        '/assets/fonts/your-font-pack/your-font-name.woff2'
    ),
    'as' => 'font',
]);
```

If you are generating URLs for CSS, Javascript or image files there are helper methods for each of these asset types:

```
// Outputs /img/icon.png
$this->Url->image('icon.png');

// Outputs /js/app.js
$this->Url->script('app.js');

// Outputs /css/app.css
$this->Url->css('app.css');

// Force timestamps for one method call.
$this->Url->css('app.css', ['timestamp' => 'force']);

// Or disable timestamps for one method call.
$this->Url->css('app.css', ['timestamp' => false]);
```

Customizing Asset URL generation

If you need to customize how asset URLs are generated, or want to use custom asset cache busting parameters you can use the `assetUrlClassName` option:

```
// In view initialize
$this->loadHelper('Url', ['assetUrlClassName' => AppAsset::class]);
```

When using the `assetUrlClassName` you must implement the same methods as `Cake\Routing\Asset` does.

For further information check Router::url[124] in the API.

Configuring Helpers

You configure helpers in CakePHP by declaring them in a view class. An `AppView` class comes with every CakePHP application and is the ideal place to add helpers for global use:

```
class AppView extends View
{
    public function initialize(): void
    {
        parent::initialize();
        $this->addHelper('Html');
        $this->addHelper('Form');
        $this->addHelper('Flash');
    }
}
```

To add helpers from plugins use the *plugin syntax* used elsewhere in CakePHP:

```
$this->addHelper('Blog.Comment');
```

You don't have to explicitly add Helpers that come from CakePHP or your application. These helpers can be lazily loaded upon first use. For example:

```
// Loads the FormHelper if it has not already been explicitly added/loaded.
$this->Form->create($article);
```

From within a plugin's views, plugin helpers can also be lazily loaded. For example, view templates in the 'Blog' plugin, can lazily load helpers from the same plugin.

Conditionally Loading Helpers

You can use the current action name to conditionally add helpers:

```
class AppView extends View
{
    public function initialize(): void
    {
        parent::initialize();
        if ($this->request->getParam('action') === 'index') {
```

(continues on next page)

[124] https://api.cakephp.org/5.x/class-Cake.Routing.Router.html#_url

(continued from previous page)

```
            $this->addHelper('ListPage');
        }
    }
}
```

You can also use your controller's `beforeRender` method to add helpers:

```
class ArticlesController extends AppController
{
    public function beforeRender(EventInterface $event)
    {
        parent::beforeRender($event);
        $this->viewBuilder()->addHelper('MyHelper');
    }
}
```

Configuration options

You can pass configuration options to helpers. These options can be used to set attribute values or modify the behavior of a helper:

```
namespace App\View\Helper;

use Cake\View\Helper;
use Cake\View\View;

class AwesomeHelper extends Helper
{
    public function initialize(array $config): void
    {
        debug($config);
    }
}
```

By default all configuration options will be merged with the `$_defaultConfig` property. This property should define the default values of any configuration your helper requires. For example:

```
namespace App\View\Helper;

use Cake\View\Helper;
use Cake\View\StringTemplateTrait;

class AwesomeHelper extends Helper
{
    use StringTemplateTrait;

    /**
     * @var array<string, mixed>
     */
    protected array $_defaultConfig = [
        'errorClass' => 'error',
```

(continues on next page)

(continued from previous page)

```
        'templates' => [
            'label' => '<label for="{{for}}">{{content}}</label>',
        ],
    ];
}
```

Any configuration provided to your helper's constructor will be merged with the default values during construction and the merged data will be set to `_config`. You can use the `getConfig()` method to read runtime configuration:

```
// Read the errorClass config option.
$class = $this->Awesome->getConfig('errorClass');
```

Using helper configuration allows you to declaratively configure your helpers and keep configuration logic out of your controller actions. If you have configuration options that cannot be included as part of a class declaration, you can set those in your controller's beforeRender callback:

```
class PostsController extends AppController
{
    public function beforeRender(EventInterface $event)
    {
        parent::beforeRender($event);
        $builder = $this->viewBuilder();
        $builder->helpers([
            'CustomStuff' => $this->_getCustomStuffConfig(),
        ]);
    }
}
```

Aliasing Helpers

One common setting to use is the `className` option, which allows you to create aliased helpers in your views. This feature is useful when you want to replace `$this->Html` or another common Helper reference with a custom implementation:

```
// src/View/AppView.php
class AppView extends View
{
    public function initialize(): void
    {
        $this->addHelper('Html', [
            'className' => 'MyHtml',
        ]);
    }
}

// src/View/Helper/MyHtmlHelper.php
namespace App\View\Helper;

use Cake\View\Helper\HtmlHelper;

class MyHtmlHelper extends HtmlHelper
```

(continues on next page)

(continued from previous page)

```
{
    // Add your code to override the core HtmlHelper
}
```

The above would *alias* `MyHtmlHelper` to `$this->Html` in your views.

Note: Aliasing a helper replaces that instance anywhere that helper is used, including inside other Helpers.

Using Helpers

Once you've configured which helpers you want to use in your controller, each helper is exposed as a public property in the view. For example, if you were using the `HtmlHelper` you would be able to access it by doing the following:

```
echo $this->Html->css('styles');
```

The above would call the `css()` method on the HtmlHelper. You can access any loaded helper using `$this->{$helperName}`.

Loading Helpers On The Fly

There may be situations where you need to dynamically load a helper from inside a view. You can use the view's `Cake\View\HelperRegistry` to do this:

```
// Either one works.
$mediaHelper = $this->loadHelper('Media', $mediaConfig);
$mediaHelper = $this->helpers()->load('Media', $mediaConfig);
```

The HelperRegistry is a *registry* and supports the registry API used elsewhere in CakePHP.

Callback Methods

Helpers feature several callbacks that allow you to augment the view rendering process. See the *Helper Class* and the *Events System* documentation for more information.

Creating Helpers

You can create custom helper classes for use in your application or plugins. Like most components of CakePHP, helper classes have a few conventions:

- Helper class files should be put in **src/View/Helper**. For example: **src/View/Helper/LinkHelper.php**
- Helper classes should be suffixed with `Helper`. For example: `LinkHelper`.
- When referencing helper names you should omit the `Helper` suffix. For example: `$this->addHelper('Link');` or `$this->loadHelper('Link');`.

You'll also want to extend `Helper` to ensure things work correctly:

```php
/* src/View/Helper/LinkHelper.php */
namespace App\View\Helper;

use Cake\View\Helper;

class LinkHelper extends Helper
{
    public function makeEdit($title, $url)
    {
        // Logic to create specially formatted link goes here...
    }
}
```

Including Other Helpers

You may wish to use some functionality already existing in another helper. To do so, you can specify helpers you wish to use with a `$helpers` array, formatted just as you would in a controller:

```php
/* src/View/Helper/LinkHelper.php (using other helpers) */

namespace App\View\Helper;

use Cake\View\Helper;

class LinkHelper extends Helper
{
    protected array $helpers = ['Html'];

    public function makeEdit($title, $url)
    {
        // Use the HTML helper to output
        // Formatted data:

        $link = $this->Html->link($title, $url, ['class' => 'edit']);

        return '<div class="editOuter">' . $link . '</div>';
    }
}
```

Using Your Helper

Once you've created your helper and placed it in **src/View/Helper/**, you can load it in your views:

```php
class AppView extends View
{
    public function initialize(): void
    {
        parent::initialize();
        $this->addHelper('Link');
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Once your helper has been loaded, you can use it in your views by accessing the matching view property:

```
<!-- make a link using the new helper -->
<?= $this->Link->makeEdit('Change this Recipe', '/recipes/edit/5') ?>
```

Note: The `HelperRegistry` will attempt to lazy load any helpers not specifically identified in your `Controller`.

Accessing View Variables Inside Your Helper

If you would like to access a View variable inside a helper, you can use `$this->getView()->get()` like:

```
class AwesomeHelper extends Helper
{
    public array $helpers = ['Html'];

    public function someMethod()
    {
        // set meta description
        return $this->Html->meta(
            'description', $this->getView()->get('metaDescription'), ['block' => 'meta']
        );
    }
}
```

Rendering A View Element Inside Your Helper

If you would like to render an Element inside your Helper you can use `$this->getView()->element()` like:

```
class AwesomeHelper extends Helper
{
    public function someFunction()
    {
        return $this->getView()->element(
            '/path/to/element',
            ['foo'=>'bar','bar'=>'foo']
        );
    }
}
```

Helper Class

`class Helper`

Callbacks

By implementing a callback method in a helper, CakePHP will automatically subscribe your helper to the relevant event. Unlike previous versions of CakePHP you should *not* call `parent` in your callbacks, as the base Helper class does not implement any of the callback methods.

`Helper::beforeRenderFile`(*EventInterface $event*, *$viewFile*)

Is called before each view file is rendered. This includes elements, views, parent views and layouts.

`Helper::afterRenderFile`(*EventInterface $event*, *$viewFile*, *$content*)

Is called after each view file is rendered. This includes elements, views, parent views and layouts. A callback can modify and return `$content` to change how the rendered content will be displayed in the browser.

`Helper::beforeRender`(*EventInterface $event*, *$viewFile*)

The beforeRender method is called after the controller's beforeRender method but before the controller renders view and layout. Receives the file being rendered as an argument.

`Helper::afterRender`(*EventInterface $event*, *$viewFile*)

Is called after the view has been rendered but before layout rendering has started.

`Helper::beforeLayout`(*EventInterface $event*, *$layoutFile*)

Is called before layout rendering starts. Receives the layout filename as an argument.

`Helper::afterLayout`(*EventInterface $event*, *$layoutFile*)

Is called after layout rendering is complete. Receives the layout filename as an argument.

CHAPTER 12

Database Access & ORM

In CakePHP, working with data through the database is done with two primary object types:

- **Repositories** or **table objects** provide access to collections of data. They allow you to save new records, modify/delete existing ones, define relations, and perform bulk operations.
- **Entities** represent individual records and allow you to define row/record level behavior & functionality.

These two classes are usually responsible for managing almost everything that happens regarding your data, its validity, interactions and evolution of the information workflow in your domain of work.

CakePHP's built-in ORM specializes in relational databases, but can be extended to support alternative datasources.

The CakePHP ORM borrows ideas and concepts from both ActiveRecord and Datamapper patterns. It aims to create a hybrid implementation that combines aspects of both patterns to create a fast, simple to use ORM.

Before we get started exploring the ORM, make sure you *configure your database connections*.

Quick Example

To get started you don't have to write any code. If you've followed the *CakePHP conventions for your database tables* you can just start using the ORM. For example if we wanted to load some data from our `articles` table we would start off creating our `Articles` table class. Create **src/Model/Table/ArticlesTable.php** with the following code:

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

use Cake\ORM\Table;

class ArticlesTable extends Table
{
}
```

Then in a controller or command we can have CakePHP create an instance for us:

```
public function someMethod()
{
    $resultset = $this->fetchTable('Articles')->find()->all();

    foreach ($resultset as $row) {
        echo $row->title;
    }
}
```

In other contexts, you can use the `LocatorAwareTrait` which add accessor methods for ORM tables:

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

public function someMethod()
{
    $articles = $this->fetchTable('Articles');
    // more code.
}
```

Within a static method you can use the `FactoryLocator` to get the table locator:

```
$articles = TableRegistry::getTableLocator()->get('Articles');
```

Table classes represent **collections** of **entities**. Next, lets create an entity class for our Articles. Entity classes let you define accessor and mutator methods, define custom logic for individual records and much more. We'll start off by adding the following to **src/Model/Entity/Article.php** after the <?php opening tag:

```
namespace App\Model\Entity;

use Cake\ORM\Entity;

class Article extends Entity
{
}
```

Entities use the singular CamelCase version of the table name as their class name by default. Now that we have created our entity class, when we load entities from the database we'll get instances of our new Article class:

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

$articles = $this->fetchTable('Articles');
$resultset = $articles->find()->all();

foreach ($resultset as $row) {
    // Each row is now an instance of our Article class.
    echo $row->title;
}
```

CakePHP uses naming conventions to link the Table and Entity class together. If you need to customize which entity a table uses you can use the `entityClass()` method to set a specific classname.

See the chapters on *Table Objects* and *Entities* for more information on how to use table objects and entities in your application.

More Information

Database Basics

The CakePHP database access layer abstracts and provides help with most aspects of dealing with relational databases such as, keeping connections to the server, building queries, preventing SQL injections, inspecting and altering schemas, and with debugging and profiling queries sent to the database.

Quick Tour

The functions described in this chapter illustrate what is possible to do with the lower-level database access API. If instead you want to learn more about the complete ORM, you can read the *Query Builder* and *Table Objects* sections.

The easiest way to create a database connection is using a `DSN` string:

```
use Cake\Datasource\ConnectionManager;

$dsn = 'mysql://root:password@localhost/my_database';
ConnectionManager::setConfig('default', ['url' => $dsn]);
```

Once created, you can access the connection object to start using it:

```
$connection = ConnectionManager::get('default');
```

Note: For supported databases, see *installation notes*.

Running Select Statements

Running raw SQL queries is a breeze:

```
use Cake\Datasource\ConnectionManager;

$connection = ConnectionManager::get('default');
$results = $connection->execute('SELECT * FROM articles')->fetchAll('assoc');
```

You can use prepared statements to insert parameters:

```
$results = $connection
    ->execute('SELECT * FROM articles WHERE id = :id', ['id' => 1])
    ->fetchAll('assoc');
```

It is also possible to use complex data types as arguments:

```
use Cake\Datasource\ConnectionManager;
use DateTime;

$connection = ConnectionManager::get('default');
$results = $connection
    ->execute(
        'SELECT * FROM articles WHERE created >= :created',
```

(continues on next page)

(continued from previous page)

```
        ['created' => new DateTime('1 day ago')],
        ['created' => 'datetime']
    )
    ->fetchAll('assoc');
```

Instead of writing the SQL manually, you can use the query builder:

```
// Prior to 4.5 use $connection->query() instead.
$results = $connection
    ->selectQuery('*', 'articles')
    ->where(['created >' => new DateTime('1 day ago')], ['created' => 'datetime'])
    ->order(['title' => 'DESC'])
    ->execute()
    ->fetchAll('assoc');
```

Running Insert Statements

Inserting rows in the database is usually a matter of a couple lines:

```
use Cake\Datasource\ConnectionManager;
use DateTime;

$connection = ConnectionManager::get('default');
$connection->insert('articles', [
    'title' => 'A New Article',
    'created' => new DateTime('now')
], ['created' => 'datetime']);
```

Running Update Statements

Updating rows in the database is equally intuitive, the following example will update the article with **id** 10:

```
use Cake\Datasource\ConnectionManager;
$connection = ConnectionManager::get('default');
$connection->update('articles', ['title' => 'New title'], ['id' => 10]);
```

Running Delete Statements

Similarly, the `delete()` method is used to delete rows from the database, the following example deletes the article with **id** 10:

```
use Cake\Datasource\ConnectionManager;
$connection = ConnectionManager::get('default');
$connection->delete('articles', ['id' => 10]);
```

Configuration

By convention database connections are configured in **config/app.php**. The connection information defined in this file is fed into *Cake\Datasource\ConnectionManager* creating the connection configuration your application will be using. Sample connection information can be found in **config/app.default.php**. A sample connection configuration would look like:

```
'Datasources' => [
    'default' => [
        'className' => 'Cake\Database\Connection',
        'driver' => 'Cake\Database\Driver\Mysql',
        'persistent' => false,
        'host' => 'localhost',
        'username' => 'my_app',
        'password' => 'secret',
        'database' => 'my_app',
        'encoding' => 'utf8mb4',
        'timezone' => 'UTC',
        'cacheMetadata' => true,
    ],
],
```

The above will create a 'default' connection, with the provided parameters. You can define as many connections as you want in your configuration file. You can also define additional connections at runtime using `Cake\Datasource\ConnectionManager::setConfig()`. An example of that would be:

```
use Cake\Datasource\ConnectionManager;

ConnectionManager::setConfig('default', [
    'className' => 'Cake\Database\Connection',
    'driver' => 'Cake\Database\Driver\Mysql',
    'persistent' => false,
    'host' => 'localhost',
    'username' => 'my_app',
    'password' => 'secret',
    'database' => 'my_app',
    'encoding' => 'utf8mb4',
    'timezone' => 'UTC',
    'cacheMetadata' => true,
]);
```

Configuration options can also be provided as a *DSN* string. This is useful when working with environment variables or *PaaS* providers:

```
ConnectionManager::setConfig('default', [
    'url' => 'mysql://my_app:sekret@localhost/my_app?encoding=utf8&timezone=UTC&
↪cacheMetadata=true',
]);
```

When using a DSN string you can define any additional parameters/options as query string arguments.

By default, all Table objects will use the `default` connection. To use a non-default connection, see *Configuring Connections*.

There are a number of keys supported in database configuration. A full list is as follows:

className
The fully namespaced class name of the class that represents the connection to a database server. This class is responsible for loading the database driver, providing SQL transaction mechanisms and preparing SQL statements among other things.

driver
The class name of the driver used to implement all specificities for a database engine. This can either be a short classname using *plugin syntax*, a fully namespaced name, or a constructed driver instance. Examples of short classnames are Mysql, Sqlite, Postgres, and Sqlserver.

persistent
Whether or not to use a persistent connection to the database. This option is not supported by SqlServer. An exception is thrown if you attempt to set `persistent` to `true` with SqlServer.

host
The database server's hostname (or IP address).

username
The username for the account.

password
The password for the account.

database
The name of the database for this connection to use. Avoid using `.` in your database name. Because of how it complicates identifier quoting CakePHP does not support `.` in database names. The path to your SQLite database should be an absolute path (for example, `ROOT . DS . 'my_app.db'`) to avoid incorrect paths caused by relative paths.

port (*optional*)
The TCP port or Unix socket used to connect to the server.

encoding
Indicates the character set to use when sending SQL statements to the server. This defaults to the database's default encoding for all databases other than DB2.

timezone
Server timezone to set.

schema
Used in PostgreSQL database setups to specify which schema to use.

unix_socket
Used by drivers that support it to connect via Unix socket files. If you are using PostgreSQL and want to use Unix sockets, leave the host key blank.

ssl_key
The file path to the SSL key file. (Only supported by MySQL).

ssl_cert
The file path to the SSL certificate file. (Only supported by MySQL).

ssl_ca
The file path to the SSL certificate authority. (Only supported by MySQL).

init
A list of queries that should be sent to the database server as when the connection is created.

log
Set to `true` to enable query logging. When enabled queries will be logged at a `debug` level with the `queriesLog` scope.

quoteIdentifiers
: Set to `true` if you are using reserved words or special characters in your table or column names. Enabling this setting will result in queries built using the *Query Builder* having identifiers quoted when creating SQL. It should be noted that this decreases performance because each query needs to be traversed and manipulated before being executed.

flags
: An associative array of PDO constants that should be passed to the underlying PDO instance. See the PDO documentation for the flags supported by the driver you are using.

cacheMetadata
: Either boolean `true`, or a string containing the cache configuration to store meta data in. Having metadata caching disabled by setting it to `false` is not advised and can result in very poor performance. See the *Metadata Caching* section for more information.

mask
: Set the permissions on the generated database file. (Only supported by SQLite)

cache
: The `cache` flag to send to SQLite.

mode
: The `mode` flag value to send to SQLite.

At this point, you might want to take a look at the *CakePHP Conventions*. The correct naming for your tables (and the addition of some columns) can score you some free functionality and help you avoid configuration. For example, if you name your database table big_boxes, your table BigBoxesTable, and your controller BigBoxesController, everything will work together automatically. By convention, use underscores, lower case, and plural forms for your database table names - for example: bakers, pastry_stores, and savory_cakes.

Note: If your MySQL server is configured with `skip-character-set-client-handshake` then you MUST use the `flags` config to set your charset encoding. For example:

```
'flags' => [\PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES utf8']
```

Read and Write Connections

Connections can have separate read and write roles. Read roles are expected to represent read-only replicas and write roles are expected to be the default connection and support write operations.

Read roles are configured by providing a `read` key in the connection config. Write roles are configured by providing a `write` key.

Role configurations override the values in the shared connection config. If the read and write role configurations are the same, a single connection to the database is used for both:

```
'default' => [
    'driver' => 'mysql',
    'username' => '...',
    'password' => '...',
    'database' => '...',
    'read' => [
        'host' => 'read-db.example.com',
    ],
    'write' => [
```

(continues on next page)

(continued from previous page)

```
        'host' => 'write-db.example.com',
    ]
];
```

You can specify the same value for both `read` and `write` key without creating multiple connections to the database.

Managing Connections

class Cake\Datasource**ConnectionManager**

The `ConnectionManager` class acts as a registry to access database connections your application has. It provides a place that other objects can get references to existing connections.

Accessing Connections

static Cake\Datasource\ConnectionManager::**get**(*$name*)

Once configured connections can be fetched using *Cake\Datasource\ConnectionManager::get()*. This method will construct and load a connection if it has not been built before, or return the existing known connection:

```
use Cake\Datasource\ConnectionManager;

$connection = ConnectionManager::get('default');
```

Attempting to load connections that do not exist will throw an exception.

Creating Connections at Runtime

Using `setConfig()` and `get()` you can create new connections that are not defined in your configuration files at runtime:

```
ConnectionManager::setConfig('my_connection', $config);
$connection = ConnectionManager::get('my_connection');
```

See the *Configuration* for more information on the configuration data used when creating connections.

Data Types

class Cake\Database**TypeFactory**

Since not every database vendor includes the same set of data types, or the same names for similar data types, CakePHP provides a set of abstracted data types for use with the database layer. The types CakePHP supports are:

string
: Maps to `VARCHAR` type. In SQL Server the `NVARCHAR` types are used.

char
: Maps to `CHAR` type. In SQL Server the `NCHAR` type is used.

text
: Maps to `TEXT` types.

uuid
Maps to the UUID type if a database provides one, otherwise this will generate a `CHAR(36)` field.

binaryuuid
Maps to the UUID type if the database provides one, otherwise this will generate a `BINARY(16)` column

integer
Maps to the `INTEGER` type provided by the database. BIT is not yet supported at this moment.

smallinteger
Maps to the `SMALLINT` type provided by the database.

tinyinteger
Maps to the `TINYINT` or `SMALLINT` type provided by the database. In MySQL `TINYINT(1)` is treated as a boolean.

biginteger
Maps to the `BIGINT` type provided by the database.

float
Maps to either `DOUBLE` or `FLOAT` depending on the database. The `precision` option can be used to define the precision used.

decimal
Maps to the `DECIMAL` type. Supports the `length` and `precision` options. Values for decimal type ares be represented as strings (not as float as some might expect). This is because decimal types are used to represent exact numeric values in databases and using float type for them in PHP can potentially lead to precision loss.

If you want the values to be *float* in your PHP code then consider using *FLOAT* or *DOUBLE* type columns in your database. Also, depending on your use case you can explicitly map your decimal columns to *float* type in your table schema.

boolean
Maps to `BOOLEAN` except in MySQL, where `TINYINT(1)` is used to represent booleans. `BIT(1)` is not yet supported at this moment.

binary
Maps to the `BLOB` or `BYTEA` type provided by the database.

date
Maps to a native `DATE` column type. The return value of this column type is *Cake\I18n\Date* which emulates the date related methods of PHP's `DateTime` class.

datetime
See *DateTime Type*.

datetimefractional
See *DateTime Type*.

timestamp
Maps to the `TIMESTAMP` type.

timestampfractional
Maps to the `TIMESTAMP(N)` type.

time
Maps to a `TIME` type in all databases.

json
Maps to a `JSON` type if it's available, otherwise it maps to `TEXT`.

enum
See *Enum Type*.

These types are used in both the schema reflection features that CakePHP provides, and schema generation features CakePHP uses when using test fixtures.

Each type can also provide translation functions between PHP and SQL representations. These methods are invoked based on the type hints provided when doing queries. For example a column that is marked as 'datetime' will automatically convert input parameters from `DateTime` instances into a timestamp or formatted datestrings. Likewise, 'binary' columns will accept file handles, and generate file handles when reading data.

DateTime Type

class Cake\Database**DateTimeType**

Maps to a native `DATETIME` column type. In PostgreSQL and SQL Server this turns into a `TIMESTAMP` type. The default return value of this column type is *Cake\I18n\DateTime* which extends Chronos[125] and the native `DateTimeImmutable`.

Cake\Database\DateTimeType::**setTimezone**(*string|\DateTimeZone|null $timezone*)

If your database server's timezone does not match your application's PHP timezone then you can use this method to specify your database's timezone. This timezone will then used when converting PHP objects to database's datetime string and vice versa.

class Cake\Database**DateTimeFractionalType**

Can be used to map datetime columns that contain microseconds such as `DATETIME(6)` in MySQL. To use this type you need to add it as a mapped type:

```
// in config/bootstrap.php
use Cake\Database\TypeFactory;
use Cake\Database\Type\DateTimeFractionalType;

// Overwrite the default datetime type with a more precise one.
TypeFactory::map('datetime', DateTimeFractionalType::class);
```

class Cake\Database**DateTimeTimezoneType**

Can be used to map datetime columns that contain time zones such as `TIMESTAMPTZ` in PostgreSQL. To use this type you need to add it as a mapped type:

```
// in config/bootstrap.php
use Cake\Database\TypeFactory;
use Cake\Database\Type\DateTimeTimezoneType;

// Overwrite the default datetime type with a more precise one.
TypeFactory::map('datetime', DateTimeTimezoneType::class);
```

[125] https://github.com/cakephp/chronos

Enum Type

class Cake\Database**EnumType**

Maps a BackedEnum[126] to a string or integer column. To use this type you need to specify which column is associated to which BackedEnum inside the table class:

```
use \Cake\Database\Type\EnumType;
use \App\Model\Enum\ArticleStatus;

// in src/Model/Table/ArticlesTable.php
public function initialize(array $config): void
{
    $this->getSchema()->setColumnType('status', EnumType::from(ArticleStatus::class));
}
```

Where `ArticleStatus` contains something like:

```
namespace App\Model\Enum;

enum ArticleStatus: string
{
    case Published = 'Y';
    case Unpublished = 'N';
}
```

CakePHP recommends a few conventions for enums:

- Enum classnames should follow `{Entity}{ColumnName}` style to enable detection while running bake and to aid with project consistency.
- Enum cases should use CamelCase style.
- Enums should implement the `Cake\Database\Type\EnumLabelInterface` to improve compatibility with bake, and `FormHelper`.

Adding Custom Types

class Cake\Database**TypeFactory**

static Cake\Database\TypeFactory::**map**(*$name*, *$class*)

If you need to use vendor specific types that are not built into CakePHP you can add additional new types to CakePHP's type system. Type classes are expected to implement the following methods:

- `toPHP`: Casts given value from a database type to a PHP equivalent.
- `toDatabase`: Casts given value from a PHP type to one acceptable by a database.
- `toStatement`: Casts given value to its Statement equivalent.
- `marshal`: Marshals flat data into PHP objects.

To fulfill the basic interface, extend `Cake\Database\Type`. For example if we wanted to add a JSON type, we could make the following type class:

[126] https://www.php.net/manual/en/language.enumerations.backed.php

```php
// in src/Database/Type/JsonType.php

namespace App\Database\Type;

use Cake\Database\Driver;
use Cake\Database\Type\BaseType;
use PDO;

class JsonType extends BaseType
{
    public function toPHP(mixed $value, Driver $driver): mixed
    {
        if ($value === null) {
            return null;
        }
        return json_decode($value, true);
    }

    public function marshal(mixed $value): mixed
    {
        if (is_array($value) || $value === null) {
            return $value;
        }
        return json_decode($value, true);
    }

    public function toDatabase(mixed $value, Driver $driver): mixed
    {
        return json_encode($value);
    }

    public function toStatement(mixed $value, Driver $driver): int
    {
        if ($value === null) {
            return PDO::PARAM_NULL;
        }
        return PDO::PARAM_STR;
    }
}
```

By default the `toStatement()` method will treat values as strings which will work for our new type.

Connecting Custom Datatypes to Schema Reflection and Generation

Once we've created our new type, we need to add it into the type mapping. During our application bootstrap we should do the following:

```php
use Cake\Database\TypeFactory;

TypeFactory::map('json', \App\Database\Type\JsonType:class);
```

We then have two ways to use our datatype in our models.

1. The first path is to overwrite the reflected schema data to use our new type.
2. The second is to implement `Cake\Database\Type\ColumnSchemaAwareInterface` and define the SQL column type and reflection logic.

Overwriting the reflected schema with our custom type will enable CakePHP's database layer to automatically convert JSON data when creating queries. In your Table's *getSchema() method* add the following:

```
class WidgetsTable extends Table
{
    public function getSchema(): TableSchemaInterface
    {
        return parent::getSchema()->setColumnType('widget_prefs', 'json');
    }
}
```

Implementing `ColumnSchemaAwareInterface` gives you more control over custom datatypes. This avoids overwriting schema definitions if your datatype has an unambiguous SQL column definition. For example, we could have our JSON type be used anytime a `TEXT` column with a specific comment is used:

```
// in src/Database/Type/JsonType.php

namespace App\Database\Type;

use Cake\Database\Driver;
use Cake\Database\Type\BaseType;
use Cake\Database\Type\ColumnSchemaAwareInterface;
use Cake\Database\Schema\TableSchemaInterface;
use PDO;

class JsonType extends BaseType
    implements ColumnSchemaAwareInterface
{
    // other methods from earlier

    /**
     * Convert abstract schema definition into a driver specific
     * SQL snippet that can be used in a CREATE TABLE statement.
     *
     * Returning null will fall through to CakePHP's built-in types.
     */
    public function getColumnSql(
        TableSchemaInterface $schema,
        string $column,
        Driver $driver
    ): ?string {
        $data = $schema->getColumn($column);
        $sql = $driver->quoteIdentifier($column);
        $sql .= ' JSON';
        if (isset($data['null']) && $data['null'] === false) {
            $sql .= ' NOT NULL';
        }
        return $sql;
    }
```

(continues on next page)

(continued from previous page)

```
    /**
     * Convert the column data returned from schema reflection
     * into the abstract schema data.
     *
     * Returning null will fall through to CakePHP's built-in types.
     */
    public function convertColumnDefinition(
        array $definition,
        Driver $driver
    ): ?array {
        return [
            'type' => $this->_name,
            'length' => null,
        ];
    }
```

The `$definition` data passed to `convertColumnDefinition()` will contain the following keys. All keys will exist but may contain `null` if the key has no value for the current database driver:

- `length` The length of a column if available..
- `precision` The precision of the column if available.
- `scale` Can be included for SQLServer connections.

Mapping Custom Datatypes to SQL Expressions

The previous example maps a custom datatype for a 'json' column type which is easily represented as a string in a SQL statement. Complex SQL data types cannot be represented as strings/integers in SQL queries. When working with these datatypes your Type class needs to implement the `Cake\Database\Type\ExpressionTypeInterface` interface. This interface lets your custom type represent a value as a SQL expression. As an example, we'll build a simple Type class for handling `POINT` type data out of MySQL. First we'll define a 'value' object that we can use to represent `POINT` data in PHP:

```
// in src/Database/Point.php
namespace App\Database;

// Our value object is immutable.
class Point
{
    protected $_lat;
    protected $_long;

    // Factory method.
    public static function parse($value)
    {
        // Parse the WKB data from MySQL.
        $unpacked = unpack('x4/corder/Ltype/dlat/dlong', $value);

        return new static($unpacked['lat'], $unpacked['long']);
    }

    public function __construct($lat, $long)
```

(continues on next page)

(continued from previous page)

```
    {
        $this->_lat = $lat;
        $this->_long = $long;
    }

    public function lat()
    {
        return $this->_lat;
    }

    public function long()
    {
        return $this->_long;
    }
}
```

With our value object created, we'll need a Type class to map data into this value object and into SQL expressions:

```
namespace App\Database\Type;

use App\Database\Point;
use Cake\Database\Driver;
use Cake\Database\Expression\FunctionExpression;
use Cake\Database\ExpressionInterface;
use Cake\Database\Type\BaseType;
use Cake\Database\Type\ExpressionTypeInterface;

class PointType extends BaseType implements ExpressionTypeInterface
{
    public function toPHP($value, Driver $d): mixed
    {
        return $value === null ? null : Point::parse($value);
    }

    public function marshal($value): mixed
    {
        if (is_string($value)) {
            $value = explode(',', $value);
        }
        if (is_array($value)) {
            return new Point($value[0], $value[1]);
        }
        return null;
    }

    public function toExpression($value): ExpressionInterface
    {
        if ($value instanceof Point) {
            return new FunctionExpression(
                'POINT',
                [
                    $value->lat(),
```

(continues on next page)

(continued from previous page)

```
                    $value->long()
                ]
            );
        }
        if (is_array($value)) {
            return new FunctionExpression('POINT', [$value[0], $value[1]]);
        }
        // Handle other cases.
    }

    public function toDatabase($value, Driver $driver): mixed
    {
        return $value;
    }
}
```

The above class does a few interesting things:

- The `toPHP` method handles parsing the SQL query results into a value object.
- The `marshal` method handles converting, data such as given request data, into our value object. We're going to accept string values like `'10.24,12.34` and arrays for now.
- The `toExpression` method handles converting our value object into the equivalent SQL expressions. In our example the resulting SQL would be something like `POINT(10.24, 12.34)`.

Once we've built our custom type, we'll need to *connect our type to our table class*.

Connection Classes

class Cake\Database**Connection**

Connection classes provide a simple interface to interact with database connections in a consistent way. They are intended as a more abstract interface to the driver layer and provide features for executing queries, logging queries, and doing transactional operations.

Executing Queries

Cake\Database\Connection::**execute**(*string $sql*, *array $params = []*, *array $types = []*) → \Cake\Database\StatementInterface

Once you've gotten a connection object, you'll probably want to issue some queries with it. CakePHP's database abstraction layer provides wrapper features on top of PDO and native drivers. These wrappers provide a similar interface to PDO. There are a few different ways you can run queries depending on the type of query you need to run and what kind of results you need back. The most basic method is `execute()` which allows you to run complet SQL queries:

```
$statement = $connection->execute('UPDATE articles SET published = 1 WHERE id = 2');
```

For parameterized queries use the 2nd argument:

```
$statement = $connection->execute(
    'UPDATE articles SET published = ? WHERE id = ?',
    [1, 2]
);
```

Without any type hinting information, `execute` will assume all placeholders are string values. If you need to bind specific types of data, you can use their abstract type names when creating a query:

```
$statement = $connection->execute(
    'UPDATE articles SET published_date = ? WHERE id = ?',
    [new DateTime('now'), 2],
    ['date', 'integer']
);
```

`Cake\Database\Connection::selectQuery()`

These methods allow you to use rich data types in your applications and properly convert them into SQL statements. The last and most flexible way of creating queries is to use the *Query Builder*. This approach allows you to build complex and expressive queries without having to use platform specific SQL. When using the query builder, no SQL will be sent to the database server until the `execute()` method is called, or the query is iterated. Iterating a query will first execute it and then start iterating over the result set:

```
$query = $connection->selectQuery();
$query->select('*')
    ->from('articles')
    ->where(['published' => true]);

foreach ($query as $row) {
    // Do something with the row.
}
```

Note: Instead of iterating the `$query` you can also call it's `all()` method to get the results.

`Cake\Database\Connection::updateQuery()`

This method provides you a builder for `UPDATE` queries:

```
$query = $connection->updateQuery('articles')
    ->set(['published' => true])
    ->where(['id' => 2]);
$statement = $query->execute();
```

`Cake\Database\Connection::insertQuery()`

This method provides you a builder for `INSERT` queries:

```
$query = $connection->insertQuery();
$query->into('articles')
    ->columns(['title'])
    ->values(['1st article']);
$statement = $query->execute();
```

`Cake\Database\Connection::deleteQuery()`

This method provides you a builder for `DELETE` queries:

```
$query = $connection->deleteQuery();
$query->delete('articles')
```

(continues on next page)

(continued from previous page)

```
    ->where(['id' => 2]);
$statement = $query->execute();
```

Using Transactions

The connection objects provide you a few simple ways you do database transactions. The most basic way of doing transactions is through the `begin()`, `commit()` and `rollback()` methods, which map to their SQL equivalents:

```
$connection->begin();
$connection->execute('UPDATE articles SET published = ? WHERE id = ?', [true, 2]);
$connection->execute('UPDATE articles SET published = ? WHERE id = ?', [false, 4]);
$connection->commit();
```

`Cake\Database\Connection::transactional`(*callable $callback*)

In addition to this interface connection instances also provide the `transactional()` method which makes handling the begin/commit/rollback calls much simpler:

```
$connection->transactional(function ($connection) {
    $connection->execute('UPDATE articles SET published = ? WHERE id = ?', [true, 2]);
    $connection->execute('UPDATE articles SET published = ? WHERE id = ?', [false, 4]);
});
```

In addition to basic queries, you can execute more complex queries using either the *Query Builder* or *Table Objects*. The transactional method will do the following:

- Call `begin`.
- Call the provided closure.
- If the closure raises an exception, a rollback will be issued. The original exception will be re-thrown.
- If the closure returns `false`, a rollback will be issued.
- If the closure executes successfully, the transaction will be committed.

Interacting with Statements

When using the lower level database API, you will often encounter statement objects. These objects allow you to manipulate the underlying prepared statement from the driver. After creating and executing a query object, or using `execute()` you will have a `StatementInterface` instance.

Executing & Fetching Rows

Once a query is executed using `execute()`, results can be fetched using `fetch()`, `fetchAll()`:

```
$statement->execute();

// Read one row.
$row = $statement->fetch('assoc');

// Read all rows.
$rows = $statement->fetchAll('assoc');
```

Getting affected Row Counts

After executing a statement, you can fetch the number of affected rows:

```
$rowCount = $statement->rowCount();
```

Checking Error Codes

If your query was not successful, you can get related error information using the `errorCode()` and `errorInfo()` methods. These methods work the same way as the ones provided by PDO:

```
$code = $statement->errorCode();
$info = $statement->errorInfo();
```

Query Logging

Query logging can be enabled when configuring your connection by setting the `log` option to `true`.

When query logging is enabled, queries will be logged to *Cake\Log\Log* using the 'debug' level, and the 'queriesLog' scope. You will need to have a logger configured to capture this level & scope. Logging to `stderr` can be useful when working on unit tests, and logging to files/syslog can be useful when working with web requests:

```
use Cake\Log\Log;

// Console logging
Log::setConfig('queries', [
    'className' => 'Console',
    'stream' => 'php://stderr',
    'scopes' => ['queriesLog']
]);

// File logging
Log::setConfig('queries', [
    'className' => 'File',
    'path' => LOGS,
    'file' => 'queries.log',
    'scopes' => ['queriesLog']
]);
```

Note: Query logging is only intended for debugging/development uses. You should never leave query logging on in production as it will negatively impact the performance of your application.

Identifier Quoting

By default CakePHP does **not** quote identifiers in generated SQL queries. The reason for this is identifier quoting has a few drawbacks:

- Performance overhead - Quoting identifiers is much slower and complex than not doing it.
- Not necessary in most cases - In non-legacy databases that follow CakePHP's conventions there is no reason to quote identifiers.

If you are using a legacy schema that requires identifier quoting you can enable it using the `quoteIdentifiers` setting in your *Configuration*. You can also enable this feature at runtime:

```
$connection->getDriver()->enableAutoQuoting();
```

When enabled, identifier quoting will cause additional query traversal that converts all identifiers into `IdentifierExpression` objects.

Note: SQL snippets contained in QueryExpression objects will not be modified.

Metadata Caching

CakePHP's ORM uses database reflection to determine the schema, indexes and foreign keys your application contains. Because this metadata changes infrequently and can be expensive to access, it is typically cached. By default, metadata is stored in the `_cake_model_` cache configuration. You can define a custom cache configuration using the `cacheMetatdata` option in your datasource configuration:

```
'Datasources' => [
    'default' => [
        // Other keys go here.

        // Use the 'orm_metadata' cache config for metadata.
        'cacheMetadata' => 'orm_metadata',
    ]
],
```

You can also configure the metadata caching at runtime with the `cacheMetadata()` method:

```
// Disable the cache
$connection->cacheMetadata(false);

// Enable the cache
$connection->cacheMetadata(true);

// Use a custom cache config
$connection->cacheMetadata('orm_metadata');
```

CakePHP also includes a CLI tool for managing metadata caches. See the *Schema Cache Tool* chapter for more information.

Creating Databases

If you want to create a connection without selecting a database you can omit the database name:

```
$dsn = 'mysql://root:password@localhost/';
```

You can now use your connection object to execute queries that create/modify databases. For example to create a database:

```
$connection->execute("CREATE DATABASE IF NOT EXISTS my_database");
```

Note: When creating a database it is a good idea to set the character set and collation parameters. If these values are missing, the database will set whatever system default values it uses.

Query Builder

class Cake\ORM\Query\SelectQuery**SelectQuery**

The ORM's query builder provides a simple to use fluent interface for creating and running queries. By composing queries together, you can create advanced queries using unions and subqueries with ease.

Underneath the covers, the query builder uses PDO prepared statements which protect against SQL injection attacks.

The SelectQuery Object

The easiest way to create a `SelectQuery` object is to use `find()` from a `Table` object. This method will return an incomplete query ready to be modified. You can also use a table's connection object to access the lower level query builder that does not include ORM features, if necessary. See the *Executing Queries* section for more information:

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

$articles = $this->fetchTable('Articles');

// Start a new query.
$query = $articles->find();
```

When inside a controller, you can use the automatic table variable that is created using the conventions system:

```
// Inside ArticlesController.php

$query = $this->Articles->find();
```

Selecting Rows From A Table

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

$query = $this->fetchTable('Articles')->find();

foreach ($query->all() as $article) {
    debug($article->title);
}
```

For the remaining examples, assume that `$articles` is a *Table*. When inside controllers, you can use `$this->Articles` instead of `$articles`.

Almost every method in a `SelectQuery` object will return the same query, this means that `SelectQuery` objects are lazy, and will not be executed unless you tell them to:

```
$query->where(['id' => 1]); // Return the same query object
$query->order(['title' => 'DESC']); // Still same object, no SQL executed
```

You can of course chain the methods you call on SelectQuery objects:

```
$query = $articles
    ->find()
    ->select(['id', 'name'])
    ->where(['id !=' => 1])
    ->order(['created' => 'DESC']);

foreach ($query->all() as $article) {
    debug($article->created);
}
```

If you try to call `debug()` on a SelectQuery object, you will see its internal state and the SQL that will be executed in the database:

```
debug($articles->find()->where(['id' => 1]));

// Outputs
// ...
// 'sql' => 'SELECT * FROM articles where id = ?'
// ...
```

You can execute a query directly without having to use `foreach` on it. The easiest way is to either call the `all()` or `toList()` methods:

```
$resultsIteratorObject = $articles
    ->find()
    ->where(['id >' => 1])
    ->all();

foreach ($resultsIteratorObject as $article) {
    debug($article->id);
}

$resultsArray = $articles
```

(continues on next page)

(continued from previous page)

```
    ->find()
    ->where(['id >' => 1])
    ->all()
    ->toList();

foreach ($resultsArray as $article) {
    debug($article->id);
}

debug($resultsArray[0]->title);
```

In the above example, `$resultsIteratorObject` will be an instance of `Cake\ORM\ResultSet`, an object you can iterate and apply several extracting and traversing methods on.

Often, there is no need to call `all()`, you can simply iterate the SelectQuery object to get its results. Query objects can also be used directly as the result object; trying to iterate the query, calling `toList()` or `toArray()`, will result in the query being executed and results returned to you.

Selecting A Single Row From A Table

You can use the `first()` method to get the first result in the query:

```
$article = $articles
    ->find()
    ->where(['id' => 1])
    ->first();

debug($article->title);
```

Getting A List Of Values From A Column

```
// Use the extract() method from the collections library
// This executes the query as well
$allTitles = $articles->find()->all()->extract('title');

foreach ($allTitles as $title) {
    echo $title;
}
```

You can also get a key-value list out of a query result:

```
$list = $articles->find('list')->all();
foreach ($list as $id => $title) {
    echo "$id : $title"
}
```

For more information on how to customize the fields used for populating the list refer to *Finding Key/Value Pairs* section.

Resultset Are Collection Objects

Once you get familiar with the Query object methods, it is strongly encouraged that you visit the *Collection* section to improve your skills in efficiently traversing the results. The resultset (returned by calling the `SelectQuery`'s `all()` method) implements the collection interface:

```
// Use the combine() method from the collections library
// This is equivalent to find('list')
$keyValueList = $articles->find()->all()->combine('id', 'title');

// An advanced example
$results = $articles->find()
    ->where(['id >' => 1])
    ->order(['title' => 'DESC'])
    ->all()
    ->map(function ($row) {
        $row->trimmedTitle = trim($row->title);
        return $row;
    })
    ->combine('id', 'trimmedTitle') // combine() is another collection method
    ->toArray(); // Also a collections library method

foreach ($results as $id => $trimmedTitle) {
    echo "$id : $trimmedTitle";
}
```

Queries Are Lazily Evaluated

Query objects are lazily evaluated. This means a query is not executed until one of the following things occur:

- The query is iterated with `foreach`.
- The query's `execute()` method is called. This will return the underlying statement object, and is to be used with insert/update/delete queries.
- The query's `first()` method is called. This will return the first result in the set built by `SELECT` (it adds `LIMIT 1` to the query).
- The query's `all()` method is called. This will return the result set and can only be used with `SELECT` statements.
- The query's `toList()` or `toArray()` method is called.

Until one of these conditions are met, the query can be modified without additional SQL being sent to the database. It also means that if a Query hasn't been evaluated, no SQL is ever sent to the database. Once executed, modifying and re-evaluating a query will result in additional SQL being run. Calling the same query without modification multiple times will return same reference.

If you want to take a look at what SQL CakePHP is generating, you can turn database *query logging* on.

Selecting Data

CakePHP makes building SELECT queries simple. To limit the fields fetched, you can use the `select()` method:

```
$query = $articles->find();
$query->select(['id', 'title', 'body']);
foreach ($query->all() as $row) {
    debug($row->title);
}
```

You can set aliases for fields by providing fields as an associative array:

```
// Results in SELECT id AS pk, title AS aliased_title, body ...
$query = $articles->find();
$query->select(['pk' => 'id', 'aliased_title' => 'title', 'body']);
```

To select distinct fields, you can use the `distinct()` method:

```
// Results in SELECT DISTINCT country FROM ...
$query = $articles->find();
$query->select(['country'])
    ->distinct(['country']);
```

To set some basic conditions you can use the `where()` method:

```
// Conditions are combined with AND
$query = $articles->find();
$query->where(['title' => 'First Post', 'published' => true]);

// You can call where() multiple times
$query = $articles->find();
$query->where(['title' => 'First Post'])
    ->where(['published' => true]);
```

You can also pass an anonymous function to the `where()` method. The passed anonymous function will receive an instance of `\Cake\Database\Expression\QueryExpression` as its first argument, and `\Cake\ORM\Query\SelectQuery` as its second:

```
$query = $articles->find();
$query->where(function (QueryExpression $exp, SelectQuery $q) {
    return $exp->eq('published', true);
});
```

See the *Advanced Conditions* section to find out how to construct more complex WHERE conditions.

Selecting Specific Fields

By default a query will select all fields from a table, the exception is when you call the `select()` function yourself and pass certain fields:

```
// Only select id and title from the articles table
$articles->find()->select(['id', 'title']);
```

If you wish to still select all fields from a table after having called `select($fields)`, you can pass the table instance to `select()` for this purpose:

```
// Only all fields from the articles table including
// a calculated slug field.
$query = $articlesTable->find();
$query
    ->select(['slug' => $query->func()->concat(['title' => 'identifier', '-', 'id' =>
↪'identifier'])])
    ->select($articlesTable); // Select all fields from articles
```

You can use `selectAlso()` to select all fields on a table and *also* select some additional fields:

```
$query = $articlesTable->find();
$query->selectAlso(['count' => $query->func()->count('*')]);
```

If you want to select all but a few fields on a table, you can use `selectAllExcept()`:

```
$query = $articlesTable->find();

// Get all fields except the published field.
$query->selectAllExcept($articlesTable, ['published']);
```

You can also pass an `Association` object when working with contained associations.

Using SQL Functions

CakePHP's ORM offers abstraction for some commonly used SQL functions. Using the abstraction allows the ORM to select the platform specific implementation of the function you want. For example, `concat` is implemented differently in MySQL, PostgreSQL and SQL Server. Using the abstraction allows your code to be portable:

```
// Results in SELECT COUNT(*) count FROM ...
$query = $articles->find();
$query->select(['count' => $query->func()->count('*')]);
```

Note that most of the functions accept an additional argument to specify the types to bind to the arguments and/or the return type, for example:

```
$query->select(['minDate' => $query->func()->min('date', ['date']);
```

For details, see the documentation for `Cake\Database\FunctionsBuilder`.

You can access existing wrappers for several SQL functions through `SelectQuery::func()`:

`rand()`
: Generate a random value between 0 and 1 via SQL.

sum()
Calculate a sum. *Assumes arguments are literal values.*

avg()
Calculate an average. *Assumes arguments are literal values.*

min()
Calculate the min of a column. *Assumes arguments are literal values.*

max()
Calculate the max of a column. *Assumes arguments are literal values.*

count()
Calculate the count. *Assumes arguments are literal values.*

cast()
Convert a field or expression from one data type to another.

concat()
Concatenate two values together. *Assumes arguments are bound parameters.*

coalesce()
Coalesce values. *Assumes arguments are bound parameters.*

dateDiff()
Get the difference between two dates/times. *Assumes arguments are bound parameters.*

now()
Defaults to returning date and time, but accepts 'time' or 'date' to return only those values.

extract()
Returns the specified date part from the SQL expression.

dateAdd()
Add the time unit to the date expression.

dayOfWeek()
Returns a FunctionExpression representing a call to SQL WEEKDAY function.

Window-Only Functions

These window-only functions contain a window expression by default:

rowNumber()
Returns an Aggregate expression for the `ROW_NUMBER()` SQL function.

lag()
Returns an Aggregate expression for the `LAG()` SQL function.

lead()
Returns an Aggregate expression for the `LEAD()` SQL function.

When providing arguments for SQL functions, there are two kinds of parameters you can use, literal arguments and bound parameters. Identifier/Literal parameters allow you to reference columns or other SQL literals. Bound parameters can be used to safely add user data to SQL functions. For example:

```
$query = $articles->find()->innerJoinWith('Categories');
$concat = $query->func()->concat([
    'Articles.title' => 'identifier',
    ' - CAT: ',
```

(continues on next page)

(continued from previous page)

```
    'Categories.name' => 'identifier',
    ' - Age: ',
    $query->func()->dateDiff([
        'NOW()' => 'literal',
        'Articles.created' => 'identifier',
    ])
]);
$query->select(['link_title' => $concat]);
```

Both `literal` and `identifier` arguments allow you to reference other columns and SQL literals while `identifier` will be appropriately quoted if auto-quoting is enabled. If not marked as literal or identifier, arguments will be bound parameters allowing you to safely pass user data to the function.

The above example generates something like this in MYSQL.

```
SELECT CONCAT(
    Articles.title,
    :c0,
    Categories.name,
    :c1,
    (DATEDIFF(NOW(), Articles.created))
) FROM articles;
```

The `:c0` argument will have `' - CAT:'` text bound when the query is executed. The `dateDiff` expression was translated to the appropriate SQL.

Custom Functions

If `func()` does not already wrap the SQL function you need, you can call it directly through `func()` and still safely pass arguments and user data as described. Make sure you pass the appropriate argument type for custom functions or they will be treated as bound parameters:

```
$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
]);
```

These custom function would generate something like this in MYSQL:

```
SELECT YEAR(created) as yearCreated,
       DATE_FORMAT(created, '%H:%i') as timeCreated
FROM articles;
```

Note: Use `func()` to pass untrusted user data to any SQL function.

Ordering Results

To apply ordering, you can use the `order` method:

```
$query = $articles->find()
    ->order(['title' => 'ASC', 'id' => 'ASC']);
```

When calling `order()` multiple times on a query, multiple clauses will be appended. However, when using finders you may sometimes need to overwrite the `ORDER BY`. Set the second parameter of `order()` (as well as `orderAsc()` or `orderDesc()`) to `SelectQuery::OVERWRITE` or to `true`:

```
$query = $articles->find()
    ->order(['title' => 'ASC']);
// Later, overwrite the ORDER BY clause instead of appending to it.
$query = $articles->find()
    ->order(['created' => 'DESC'], SelectQuery::OVERWRITE);
```

The `orderAsc` and `orderDesc` methods can be used when you need to sort on complex expressions:

```
$query = $articles->find();
$concat = $query->func()->concat([
    'title' => 'identifier',
    'synopsis' => 'identifier'
]);
$query->orderAsc($concat);
```

To build complex order clauses, use a Closure to build order expressions:

```
$query->orderAsc(function (QueryExpression $exp, SelectQuery $query) {
    return $exp->addCase(...);
});
```

Limiting Results

To limit the number of rows or set the row offset you can use the `limit()` and `page()` methods:

```
// Fetch rows 50 to 100
$query = $articles->find()
    ->limit(50)
    ->page(2);
```

As you can see from the examples above, all the methods that modify the query provide a fluent interface, allowing you to build a query through chained method calls.

Aggregates - Group and Having

When using aggregate functions like `count` and `sum` you may want to use `group by` and `having` clauses:

```
$query = $articles->find();
$query->select([
    'count' => $query->func()->count('view_count'),
    'published_date' => 'DATE(created)'
])
->groupBy('published_date')
->having(['count >' => 3]);
```

Case Statements

The ORM also offers the SQL `case` expression. The `case` expression allows for implementing `if ... then ... else` logic inside your SQL. This can be useful for reporting on data where you need to conditionally sum or count data, or where you need to specific data based on a condition.

If we wished to know how many published articles are in our database, we could use the following SQL:

```
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
```

To do this with the query builder, we'd use the following code:

```
$query = $articles->find();
$publishedCase = $query->newExpr()
    ->case()
    ->when(['published' => 'Y'])
    ->then(1);
$unpublishedCase = $query->newExpr()
    ->case()
    ->when(['published' => 'N'])
    ->then(1);

$query->select([
    'number_published' => $query->func()->count($publishedCase),
    'number_unpublished' => $query->func()->count($unpublishedCase)
]);
```

The `when()` method accepts SQL snippets, array conditions, and `Closure` for when you need additional logic to build the cases. If we wanted to classify cities into SMALL, MEDIUM, or LARGE based on population size, we could do the following:

```
$query = $cities->find();
$sizing = $query->newExpr()->case()
    ->when(['population <' => 100000])
    ->then('SMALL')
    ->when($query->newExpr()->between('population', 100000, 999000))
    ->then('MEDIUM')
    ->when(['population >=' => 999001])
```

(continues on next page)

(continued from previous page)

```
    ->then('LARGE');
$query = $query->select(['size' => $sizing]);
# SELECT CASE
#   WHEN population < 100000 THEN 'SMALL'
#   WHEN population BETWEEN 100000 AND 999000 THEN 'MEDIUM'
#   WHEN population >= 999001 THEN 'LARGE'
#   END AS size
```

You need to be careful when including user provided data into case expressions as it can create SQL injection vulnerabilities:

```
// Unsafe do *not* use
$case->when($requestData['published']);

// Instead pass user data as values to array conditions
$case->when(['published' => $requestData['published']]);
```

For more complex scenarios you can use `QueryExpression` objects and bound values:

```
$userValue = $query->newExpr()
    ->case()
    ->when($query->newExpr('population >= :userData'))
    ->then(123, 'integer');

$query->select(['val' => $userValue])
    ->bind(':userData', $requestData['value'], 'integer');
```

By using bindings you can safely embed user data into complex raw SQL snippets.

`then()`, `when()` and `else()` will try to infer the value type based on the parameter type. If you need to bind a value as a different type you can declare the desired type:

```
$case->when(['published' => true])->then('1', 'integer');
```

You can create `if ... then ... else` conditions by using `else()`:

```
$published = $query->newExpr()
    ->case()
    ->when(['published' => true])
    ->then('Y');
    ->else('N');

# CASE WHEN published = true THEN 'Y' ELSE 'N' END;
```

Also, it's possible to create the simple variant by passing a value to `case()`:

```
$published = $query->newExpr()
    ->case($query->identifier('published'))
    ->when(true)
    ->then('Y');
    ->else('N');

# CASE published WHEN true THEN 'Y' ELSE 'N' END;
```

The `addCase` function can also chain together multiple statements to create `if .. then .. [elseif .. then .. ] [ .. else ]` logic inside your SQL.

If we wanted to classify cities into SMALL, MEDIUM, or LARGE based on population size, we could do the following:

```
$query = $cities->find()
    ->where(function (QueryExpression $exp, SelectQuery $q) {
        return $exp->addCase(
            [
                $q->newExpr()->lt('population', 100000),
                $q->newExpr()->between('population', 100000, 999000),
                $q->newExpr()->gte('population', 999001),
            ],
            ['SMALL',  'MEDIUM', 'LARGE'], # values matching conditions
            ['string', 'string', 'string'] # type of each value
        );
    });
# WHERE CASE
#   WHEN population < 100000 THEN 'SMALL'
#   WHEN population BETWEEN 100000 AND 999000 THEN 'MEDIUM'
#   WHEN population >= 999001 THEN 'LARGE'
#   END
```

Any time there are fewer case conditions than values, `addCase` will automatically produce an `if .. then .. else` statement:

```
$query = $cities->find()
    ->where(function (QueryExpression $exp, SelectQuery $q) {
        return $exp->addCase(
            [
                $q->newExpr()->eq('population', 0),
            ],
            ['DESERTED', 'INHABITED'], # values matching conditions
            ['string', 'string'] # type of each value
        );
    });
# WHERE CASE
#   WHEN population = 0 THEN 'DESERTED' ELSE 'INHABITED' END
```

Fetching Arrays Instead of Entities

While ORMs and object result sets are powerful, creating entities is sometimes unnecessary. For example, when accessing aggregated data, building an Entity may not make sense. The process of converting the database results to entities is called hydration. If you wish to disable this process you can do this:

```
$query = $articles->find();
$query->enableHydration(false); // Results as arrays instead of entities
$result = $query->toList(); // Execute the query and return the array
```

After executing those lines, your result should look similar to this:

```
[
    ['id' => 1, 'title' => 'First Article', 'body' => 'Article 1 body' ...],
```

(continues on next page)

(continued from previous page)

```
    ['id' => 2, 'title' => 'Second Article', 'body' => 'Article 2 body' ...],
    ...
]
```

Adding Calculated Fields

After your queries, you may need to do some post-processing. If you need to add a few calculated fields or derived data, you can use the `formatResults()` method. This is a lightweight way to map over the result sets. If you need more control over the process, or want to reduce results you should use the *Map/Reduce* feature instead. If you were querying a list of people, you could calculate their age with a result formatter:

```
// Assuming we have built the fields, conditions and containments.
$query->formatResults(function (\Cake\Collection\CollectionInterface $results) {
    return $results->map(function ($row) {
        $row['age'] = $row['birth_date']->diff(new \DateTime)->y;
        return $row;
    });
});
```

As you can see in the example above, formatting callbacks will get a `ResultSetDecorator` as their first argument. The second argument will be the Query instance the formatter was attached to. The `$results` argument can be traversed and modified as necessary.

Result formatters are required to return an iterator object, which will be used as the return value for the query. Formatter functions are applied after all the Map/Reduce routines have been executed. Result formatters can be applied from within contained associations as well. CakePHP will ensure that your formatters are properly scoped. For example, doing the following would work as you may expect:

```
// In a method in the Articles table
$query->contain(['Authors' => function ($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;
        });
    });
}]);

// Get results
$results = $query->all();

// Outputs 29
echo $results->first()->author->age;
```

As seen above, the formatters attached to associated query builders are scoped to operate only on the data in the association. CakePHP will ensure that computed values are inserted into the correct entity.

Advanced Conditions

The query builder makes it simple to build complex `where` clauses. Grouped conditions can be expressed by providing combining `where()` and expression objects. For simple queries, you can build conditions using an array of conditions:

```
$query = $articles->find()
    ->where([
        'author_id' => 3,
        'OR' => [['view_count' => 2], ['view_count' => 3]],
    ]);
```

The above would generate SQL like

```
SELECT * FROM articles WHERE author_id = 3 AND (view_count = 2 OR view_count = 3)
```

If you'd prefer to avoid deeply nested arrays, you can use the callback form of `where()` to build your queries. The callback accepts a QueryExpression which allows you to use the expression builder interface to build more complex conditions without arrays. For example:

```
$query = $articles->find()->where(function (QueryExpression $exp, SelectQuery $query) {
    // Use add() to add multiple conditions for the same field.
    $author = $query->newExpr()->or(['author_id' => 3])->add(['author_id' => 2]);
    $published = $query->newExpr()->and(['published' => true, 'view_count' => 10]);

    return $exp->or([
        'promoted' => true,
        $query->newExpr()->and([$author, $published])
    ]);
});
```

The above generates SQL similar to:

```
SELECT *
FROM articles
WHERE (
    (
        (author_id = 2 OR author_id = 3)
        AND
        (published = 1 AND view_count = 10)
    )
    OR promoted = 1
)
```

The `QueryExpression` passed to the callback allows you to use both **combinators** and **conditions** to build the full expression.

Combinators
: These create new `QueryExpression` objects and set how the conditions added to that expression are joined together.

  - `and()` creates new expression objects that joins all conditions with `AND`.
  - `or()` creates new expression objects that joins all conditions with `OR`.

Conditions
: These are added to the expression and automatically joined together depending on which combinator was used.

The `QueryExpression` passed to the callback function defaults to `and()`:

```
$query = $articles->find()
    ->where(function (QueryExpression $exp) {
        return $exp
            ->eq('author_id', 2)
            ->eq('published', true)
            ->notEq('spam', true)
            ->gt('view_count', 10);
    });
```

Since we started off using `where()`, we don't need to call `and()`, as that happens implicitly. The above shows a few new condition methods being combined with `AND`. The resulting SQL would look like:

```
SELECT *
FROM articles
WHERE (
author_id = 2
AND published = 1
AND spam != 1
AND view_count > 10)
```

However, if we wanted to use both `AND` & `OR` conditions we could do the following:

```
$query = $articles->find()
    ->where(function (QueryExpression $exp) {
        $orConditions = $exp->or(['author_id' => 2])
            ->eq('author_id', 5);
        return $exp
            ->add($orConditions)
            ->eq('published', true)
            ->gte('view_count', 10);
    });
```

Which would generate the SQL similar to:

```
SELECT *
FROM articles
WHERE (
    (author_id = 2 OR author_id = 5)
    AND published = 1
    AND view_count >= 10
)
```

The **combinators** also allow you pass in a callback which takes the new expression object as a parameter if you want to separate the method chaining:

```
$query = $articles->find()
    ->where(function (QueryExpression $exp) {
        $orConditions = $exp->or(function (QueryExpression $or) {
            return $or->eq('author_id', 2)
                ->eq('author_id', 5);
        });
        return $exp
```

(continues on next page)

(continued from previous page)

```
            ->not($orConditions)
            ->lte('view_count', 10);
    });
```

You can negate sub-expressions using `not()`:

```
$query = $articles->find()
    ->where(function (QueryExpression $exp) {
        $orConditions = $exp->or(['author_id' => 2])
            ->eq('author_id', 5);
        return $exp
            ->not($orConditions)
            ->lte('view_count', 10);
    });
```

Which will generate the following SQL looking like:

```
SELECT *
FROM articles
WHERE (
    NOT (author_id = 2 OR author_id = 5)
    AND view_count <= 10
)
```

It is also possible to build expressions using SQL functions:

```
$query = $articles->find()
    ->where(function (QueryExpression $exp, SelectQuery $q) {
        $year = $q->func()->year([
            'created' => 'identifier'
        ]);
        return $exp
            ->gte($year, 2014)
            ->eq('published', true);
    });
```

Which will generate the following SQL looking like:

```
SELECT *
FROM articles
WHERE (
    YEAR(created) >= 2014
    AND published = 1
)
```

When using the expression objects you can use the following methods to create conditions:

- `eq()` Creates an equality condition:

  ```
  $query = $cities->find()
      ->where(function (QueryExpression $exp, SelectQuery $q) {
          return $exp->eq('population', '10000');
      });
  # WHERE population = 10000
  ```

- `notEq()` Creates an inequality condition:

```
$query = $cities->find()
    ->where(function (QueryExpression $exp, SelectQuery $q) {
        return $exp->notEq('population', '10000');
    });
# WHERE population != 10000
```

- `like()` Creates a condition using the `LIKE` operator:

```
$query = $cities->find()
    ->where(function (QueryExpression $exp, SelectQuery $q) {
        return $exp->like('name', '%A%');
    });
# WHERE name LIKE "%A%"
```

- `notLike()` Creates a negated `LIKE` condition:

```
$query = $cities->find()
    ->where(function (QueryExpression $exp, SelectQuery $q) {
        return $exp->notLike('name', '%A%');
    });
# WHERE name NOT LIKE "%A%"
```

- `in()` Create a condition using `IN`:

```
$query = $cities->find()
    ->where(function (QueryExpression $exp, SelectQuery $q) {
        return $exp->in('country_id', ['AFG', 'USA', 'EST']);
    });
# WHERE country_id IN ('AFG', 'USA', 'EST')
```

- `notIn()` Create a negated condition using `IN`:

```
$query = $cities->find()
    ->where(function (QueryExpression $exp, SelectQuery $q) {
        return $exp->notIn('country_id', ['AFG', 'USA', 'EST']);
    });
# WHERE country_id NOT IN ('AFG', 'USA', 'EST')
```

- `gt()` Create a > condition:

```
$query = $cities->find()
    ->where(function (QueryExpression $exp, SelectQuery $q) {
        return $exp->gt('population', '10000');
    });
# WHERE population > 10000
```

- `gte()` Create a >= condition:

```
$query = $cities->find()
    ->where(function (QueryExpression $exp, SelectQuery $q) {
        return $exp->gte('population', '10000');
    });
# WHERE population >= 10000
```

- `lt()` Create a < condition:

```
$query = $cities->find()
    ->where(function (QueryExpression $exp, SelectQuery $q) {
        return $exp->lt('population', '10000');
    });
# WHERE population < 10000
```

- `lte()` Create a <= condition:

```
$query = $cities->find()
    ->where(function (QueryExpression $exp, SelectQuery $q) {
        return $exp->lte('population', '10000');
    });
# WHERE population <= 10000
```

- `isNull()` Create an `IS NULL` condition:

```
$query = $cities->find()
    ->where(function (QueryExpression $exp, SelectQuery $q) {
        return $exp->isNull('population');
    });
# WHERE (population) IS NULL
```

- `isNotNull()` Create a negated `IS NULL` condition:

```
$query = $cities->find()
    ->where(function (QueryExpression $exp, SelectQuery $q) {
        return $exp->isNotNull('population');
    });
# WHERE (population) IS NOT NULL
```

- `between()` Create a `BETWEEN` condition:

```
$query = $cities->find()
    ->where(function (QueryExpression $exp, SelectQuery $q) {
        return $exp->between('population', 999, 5000000);
    });
# WHERE population BETWEEN 999 AND 5000000,
```

- `exists()` Create a condition using `EXISTS`:

```
$subquery = $cities->find()
    ->select(['id'])
    ->where(function (QueryExpression $exp, SelectQuery $q) {
        return $exp->equalFields('countries.id', 'cities.country_id');
    })
    ->andWhere(['population >' => 5000000]);

$query = $countries->find()
    ->where(function (QueryExpression $exp, SelectQuery $q) use ($subquery) {
        return $exp->exists($subquery);
    });
# WHERE EXISTS (SELECT id FROM cities WHERE countries.id = cities.country_id AND␣
↪population > 5000000)
```

- `notExists()` Create a negated condition using `EXISTS`:

```
$subquery = $cities->find()
    ->select(['id'])
    ->where(function (QueryExpression $exp, SelectQuery $q) {
        return $exp->equalFields('countries.id', 'cities.country_id');
    })
    ->andWhere(['population >' => 5000000]);

$query = $countries->find()
    ->where(function (QueryExpression $exp, SelectQuery $q) use ($subquery) {
        return $exp->notExists($subquery);
    });
# WHERE NOT EXISTS (SELECT id FROM cities WHERE countries.id = cities.country_id␣
↪AND population > 5000000)
```

Expression objects should cover many commonly used functions and expressions. If you find yourself unable to create the required conditions with expressions you can may be able to use `bind()` to manually bind parameters into conditions:

```
$query = $cities->find()
    ->where([
        'start_date BETWEEN :start AND :end',
    ])
    ->bind(':start', '2014-01-01', 'date')
    ->bind(':end',   '2014-12-31', 'date');
```

In situations when you can't get, or don't want to use the builder methods to create the conditions you want you can also use snippets of SQL in where clauses:

```
// Compare two fields to each other
$query->where(['Categories.parent_id != Parents.id']);
```

Warning: The field names used in expressions, and SQL snippets should **never** contain untrusted content as you will create SQL Injection vectors. See the *Using SQL Functions* section for how to safely include unsafe data into function calls.

Using Identifiers in Expressions

When you need to reference a column or SQL identifier in your queries you can use the `identifier()` method:

```
$query = $countries->find();
$query->select([
        'year' => $query->func()->year([$query->identifier('created')])
    ])
    ->where(function ($exp, $query) {
        return $exp->gt('population', 100000);
    });
```

You can use `identifier()` in comparisons to aggregations too:

```
$query = $this->Orders->find();
$query->select(['Customers.customer_name', 'total_orders' => $query->func()->count(
↪'Orders.order_id')])
    ->contain('Customers')
    ->group(['Customers.customer_name'])
    ->having(['total_orders >=' => $query->identifier('Customers.minimum_order_count')]);
```

Warning: To prevent SQL injections, Identifier expressions should never have untrusted data passed into them.

Collation

In situations that you need to deal with accented characters, multilingual data or case-sensitive comparisons, you can use the `$collation` parameter of `IdentifierExpression` or `StringExpression` to apply a character expression to a certain collation:

```
use Cake\Database\Expression\IdentifierExpression;

$collation = 'Latin1_general_CI_AI'; //sql server example
$query = $cities->find()
    ->where(function (QueryExpression $exp, Query $q) use ($collation) {
        return $exp->like(new IdentifierExpression('name', $collation), '%São José%');
    });
# WHERE name COLLATE LIKE Latin1_general_CI_AI "%São José%"
```

Automatically Creating IN Clauses

When building queries using the ORM, you will generally not have to indicate the data types of the columns you are interacting with, as CakePHP can infer the types based on the schema data. If in your queries you'd like CakePHP to automatically convert equality to `IN` comparisons, you'll need to indicate the column data type:

```
$query = $articles->find()
    ->where(['id' => $ids], ['id' => 'integer[]']);

// Or include IN to automatically cast to an array.
$query = $articles->find()
    ->where(['id IN' => $ids]);
```

The above will automatically create `id IN (...)` instead of `id = ?`. This can be useful when you do not know whether you will get a scalar or array of parameters. The `[]` suffix on any data type name indicates to the query builder that you want the data handled as an array. If the data is not an array, it will first be cast to an array. After that, each value in the array will be cast using the *type system*. This works with complex types as well. For example, you could take a list of DateTime objects using:

```
$query = $articles->find()
    ->where(['post_date' => $dates], ['post_date' => 'date[]']);
```

Automatic IS NULL Creation

When a condition value is expected to be `null` or any other value, you can use the `IS` operator to automatically create the correct expression:

```
$query = $categories->find()
    ->where(['parent_id IS' => $parentId]);
```

The above will generate``parent_id = :c1`` or `parent_id IS NULL` depending on the type of `$parentId`

Automatic IS NOT NULL Creation

When a condition value is expected not to be `null` or any other value, you can use the `IS NOT` operator to automatically create the correct expression:

```
$query = $categories->find()
    ->where(['parent_id IS NOT' => $parentId]);
```

The above will generate``parent_id != :c1`` or `parent_id IS NOT NULL` depending on the type of `$parentId`

Raw Expressions

When you cannot construct the SQL you need using the query builder, you can use expression objects to add snippets of SQL to your queries:

```
$query = $articles->find();
$expr = $query->newExpr()->add('1 + 1');
$query->select(['two' => $expr]);
```

`Expression` objects can be used with any query builder methods like `where()`, `limit()`, `groupBy()`, `select()` and many other methods.

> **Warning:** Using expression objects leaves you vulnerable to SQL injection. You should never use untrusted data into expressions.

Using Connection Roles

If you have configured *Read and Write Connections* in your application, you can have a query run on the `read` connection using one of the role methods:

```
// Run a query on the read connection
$query->useReadRole();

// Run a query on the write connection (default)
$query->useWriteRole();
```

New in version 4.5.0: Query role methods were added in 4.5.0

Expression Conjuction

It is possible to change the conjunction used to join conditions in a query expression using the method `setConjunction`:

```
$query = $articles->find();
$expr = $query->newExpr(['1','1'])->setConjunction('+');
$query->select(['two' => $expr]);
```

And can be used combined with aggregations too:

```
$query = $products->find();
$query->select(function ($query) {
        $stockQuantity = $query->func()->sum('Stocks.quantity');
        $totalStockValue = $query->func()->sum(
                $query->newExpr(['Stocks.quantity', 'Products.unit_price'])
                    ->setConjunction('*')
        );
        return [
            'Products.name',
            'stock_quantity' => $stockQuantity,
            'Products.unit_price',
            'total_stock_value' => $totalStockValue
        ];
    })
    ->innerJoinWith('Stocks')
    ->groupBy(['Products.id', 'Products.name', 'Products.unit_price']);
```

Tuple Comparison

Tuple comparison involves comparing two rows of data (tuples) element by element, typically using comparison operators like `<`, `>`, `=`:

```
$products->find()
    ->where([
        'OR' => [
            ['unit_price <' => 20],
            ['unit_price' => 20, 'tax_percentage <=' => 5],
        ]
    ]);

# WHERE (unit_price < 20 OR (unit_price = 20 AND tax_percentage <= 5))
```

The same result can be achieved using `TupleComparison`:

```
use Cake\Database\Expression\TupleComparison;

$products->find()
    ->where(
        new TupleComparison(
            ['unit_price', 'tax_percentage'],
            [20, 5],
```

(continues on next page)

(continued from previous page)

```
            ['integer', 'integer'], # type of each value
            '<='
        )
    );

# WHERE (unit_price, tax_percentage) <= (20, 5))
```

Tuple Comparison can also be used with `IN` and the result can be transformed even on DBMS that does not natively support it:

```
$articles->find()
        ->where(
            new TupleComparison(
                ['articles.id', 'articles.author_id'],
                [[10, 10], [30, 10]],
                ['integer', 'integer'],
                'IN'
            ),
        );

# WHERE (1) = ( SELECT (1) WHERE ( ( articles.id = : 10 AND articles.author_id = : 10 )␣
↪OR ( articles.id = : 30 AND articles.author_id = : 30 ) ) )
```

Note: Tuple comparison transform only supports the `IN` and = operators

Getting Results

Once you've made your query, you'll want to retrieve rows from it. There are a few ways of doing this:

```
// Iterate the query
foreach ($query as $row) {
    // Do stuff.
}

// Get the results
$results = $query->all();
```

You can use *any of the collection* methods on your query objects to pre-process or transform the results:

```
// Use one of the collection methods.
$ids = $query->map(function ($row) {
    return $row->id;
});

$maxAge = $query->max(function ($max) {
    return $max->age;
});
```

You can use `first` or `firstOrFail` to retrieve a single record. These methods will alter the query adding a `LIMIT 1` clause:

```
// Get just the first row
$row = $query->first();

// Get the first row or an exception.
$row = $query->firstOrFail();
```

Returning the Total Count of Records

Using a single query object, it is possible to obtain the total number of rows found for a set of conditions:

```
$total = $articles->find()->where(['is_active' => true])->count();
```

The `count()` method will ignore the `limit`, `offset` and `page` clauses, thus the following will return the same result:

```
$total = $articles->find()->where(['is_active' => true])->limit(10)->count();
```

This is useful when you need to know the total result set size in advance, without having to construct another `SelectQuery` object. Likewise, all result formatting and map-reduce routines are ignored when using the `count()` method.

Moreover, it is possible to return the total count for a query containing group by clauses without having to rewrite the query in any way. For example, consider this query for retrieving article ids and their comments count:

```
$query = $articles->find();
$query->select(['Articles.id', $query->func()->count('Comments.id')])
    ->matching('Comments')
    ->groupBy(['Articles.id']);
$total = $query->count();
```

After counting, the query can still be used for fetching the associated records:

```
$list = $query->all();
```

Sometimes, you may want to provide an alternate method for counting the total records of a query. One common use case for this is providing a cached value or an estimate of the total rows, or to alter the query to remove expensive unneeded parts such as left joins. This becomes particularly handy when using the CakePHP built-in pagination system which calls the `count()` method:

```
$query = $query->where(['is_active' => true])->counter(function ($query) {
    return 100000;
});
$query->count(); // Returns 100000
```

In the example above, when the pagination component calls the count method, it will receive the estimated hard-coded number of rows.

Caching Loaded Results

When fetching entities that don't change often you may want to cache the results. The `SelectQuery` class makes this simple:

```
$query->cache('recent_articles');
```

Will enable caching on the query's result set. If only one argument is provided to `cache()` then the 'default' cache configuration will be used. You can control which caching configuration is used with the second parameter:

```
// String config name.
$query->cache('recent_articles', 'dbResults');

// Instance of CacheEngine
$query->cache('recent_articles', $memcache);
```

In addition to supporting static keys, the `cache()` method accepts a function to generate the key. The function you give it will receive the query as an argument. You can then read aspects of the query to dynamically generate the cache key:

```
// Generate a key based on a simple checksum
// of the query's where clause
$query->cache(function ($q) {
    return 'articles-' . md5(serialize($q->clause('where')));
});
```

The cache method makes it simple to add cached results to your custom finders or through event listeners.

When the results for a cached query are fetched the following happens:

1. If the query has results set, those will be returned.
2. The cache key will be resolved and cache data will be read. If the cache data is not empty, those results will be returned.
3. If the cache misses, the query will be executed, the `Model.beforeFind` event will be triggered, and a new `ResultSet` will be created. This `ResultSet` will be written to the cache and returned.

Note: You cannot cache a streaming query result.

Loading Associations

The builder can help you retrieve data from multiple tables at the same time with the minimum amount of queries possible. To be able to fetch associated data, you first need to setup associations between the tables as described in the *Associations - Linking Tables Together* section. This technique of combining queries to fetch associated data from other tables is called **eager loading**.

Eager loading helps avoid many of the potential performance problems surrounding lazy-loading in an ORM. The queries generated by eager loading can better leverage joins, allowing more efficient queries to be made. In CakePHP you state which associations should be eager loaded using the 'contain' method:

```
// In a controller or table method.

// As an option to find()
$query = $articles->find('all', contain: ['Authors', 'Comments']);
```

(continues on next page)

(continued from previous page)

```
// As a method on the query object
$query = $articles->find('all');
$query->contain(['Authors', 'Comments']);
```

The above will load the related author and comments for each article in the result set. You can load nested associations using nested arrays to define the associations to be loaded:

```
$query = $articles->find()->contain([
    'Authors' => ['Addresses'], 'Comments' => ['Authors']
]);
```

Alternatively, you can express nested associations using the dot notation:

```
$query = $articles->find()->contain([
    'Authors.Addresses',
    'Comments.Authors'
]);
```

You can eager load associations as deep as you like:

```
$query = $products->find()->contain([
    'Shops.Cities.Countries',
    'Shops.Managers'
]);
```

Which is equivalent to calling:

```
$query = $products->find()->contain([
    'Shops' => ['Cities.Countries', 'Managers']
]);
```

You can select fields from all associations with multiple `contain()` statements:

```
$query = $this->find()->select([
    'Realestates.id',
    'Realestates.title',
    'Realestates.description'
])
->contain([
    'RealestateAttributes' => [
        'Attributes' => [
            'fields' => [
                // Aliased fields in contain() must include
                // the model prefix to be mapped correctly.
                'Attributes__name' => 'attr_name',
            ],
        ],
    ],
])
->contain([
    'RealestateAttributes' => [
        'fields' => [
```

(continues on next page)

(continued from previous page)

```
            'RealestateAttributes.realestate_id',
            'RealestateAttributes.value',
        ],
    ],
])
->where($condition);
```

If you need to reset the containments on a query you can set the second argument to `true`:

```
$query = $articles->find();
$query->contain(['Authors', 'Comments'], true);
```

Note: Association names in `contain()` calls should use the same association casing as in your association definitions, not the property name used to hold the association record(s). For example, if you have declared an association as `belongsTo('Users')` then you must use `contain('Users')` and not `contain('users')` or `contain('user')`.

Passing Conditions to Contain

When using `contain()` you are able to restrict the data returned by the associations and filter them by conditions. To specify conditions, pass an anonymous function that receives as the first argument a query object, `\Cake\ORM\Query\SelectQuery`:

```
// In a controller or table method.
$query = $articles->find()->contain('Comments', function (SelectQuery $q) {
    return $q
        ->select(['body', 'author_id'])
        ->where(['Comments.approved' => true]);
});
```

This also works for pagination at the Controller level:

```
$this->paginate['contain'] = [
    'Comments' => function (SelectQuery $query) {
        return $query->select(['body', 'author_id'])
        ->where(['Comments.approved' => true]);
    }
];
```

Warning: If the results are missing association entities, make sure the foreign key columns are selected in the query. Without the foreign keys, the ORM cannot find matching rows.

It is also possible to restrict deeply-nested associations using the dot notation:

```
$query = $articles->find()->contain([
    'Comments',
    'Authors.Profiles' => function (SelectQuery $q) {
        return $q->where(['Profiles.is_published' => true]);
```

(continues on next page)

(continued from previous page)

```
    }
]);
```

In the above example, you'll still get authors even if they don't have a published profile. To only get authors with a published profile use *matching()*. If you have defined custom finders in your associations, you can use them inside `contain()`:

```
// Bring all articles, but only bring the comments that are approved and
// popular.
$query = $articles->find()->contain('Comments', function (SelectQuery $q) {
    return $q->find('approved')->find('popular');
});
```

Note: With `BelongsTo` and `HasOne` associations only `select` and `where` clauses are valid in the `contain()` query. With `HasMany` and `BelongsToMany` all clauses such as `order()` are valid.

You can control more than just the query clauses used by `contain()`. If you pass an array with the association, you can override the `foreignKey`, `joinType` and `strategy`. See *Associations - Linking Tables Together* for details on the default value and options for each association type.

You can pass `false` as the new `foreignKey` to disable foreign key constraints entirely. Use the `queryBuilder` option to customize the query when using an array:

```
$query = $articles->find()->contain([
    'Authors' => [
        'foreignKey' => false,
        'queryBuilder' => function (SelectQuery $q) {
            return $q->where(...); // Full conditions for filtering
        }
    ]
]);
```

If you have limited the fields you are loading with `select()` but also want to load fields off of contained associations, you can pass the association object to `select()`:

```
// Select id & title from articles, but all fields off of Users.
$query = $articles->find()
    ->select(['id', 'title'])
    ->select($articles->Users)
    ->contain(['Users']);
```

Alternatively, you can use `enableAutoFields()` in an anonymous function:

```
// Select id & title from articles, but all fields off of Users.
$query = $articles->find()
    ->select(['id', 'title'])
    ->contain(['Users' => function(SelectQuery $q) {
        return $q->enableAutoFields();
    }]);
```

Sorting Contained Associations

When loading HasMany and BelongsToMany associations, you can use the `sort` option to sort the data in those associations:

```
$query->contain([
    'Comments' => [
        'sort' => ['Comments.created' => 'DESC']
    ]
]);
```

Filtering by Associated Data

A fairly common query case with associations is finding records 'matching' specific associated data. For example if you have 'Articles belongsToMany Tags' you will probably want to find Articles that have the CakePHP tag. This is extremely simple to do with the ORM in CakePHP:

```
// In a controller or table method.

$query = $articles->find();
$query->matching('Tags', function ($q) {
    return $q->where(['Tags.name' => 'CakePHP']);
});
```

You can apply this strategy to HasMany associations as well. For example if 'Authors HasMany Articles', you could find all the authors with recently published articles using the following:

```
$query = $authors->find();
$query->matching('Articles', function ($q) {
    return $q->where(['Articles.created >=' => new DateTime('-10 days')]);
});
```

Filtering by deep associations uses the same predictable syntax from `contain()`:

```
// In a controller or table method.
$query = $products->find()->matching(
    'Shops.Cities.Countries', function ($q) {
        return $q->where(['Countries.name' => 'Japan']);
    }
);

// Bring unique articles that were commented by `markstory` using passed variable
// Dotted matching paths should be used over nested matching() calls
$username = 'markstory';
$query = $articles->find()->matching('Comments.Users', function ($q) use ($username) {
    return $q->where(['username' => $username]);
});
```

Note: As this function will create an `INNER JOIN`, you might want to consider calling `distinct` on the find query as you might get duplicate rows if your conditions don't exclude them already. This might be the case, for example, when the same users comments more than once on a single article.

The data from the association that is 'matched' will be available on the `_matchingData` property of entities. If both match and contain the same association, you can expect to get both the `_matchingData` and standard association properties in your results.

Using innerJoinWith

Sometimes you need to match specific associated data but without actually loading the matching records like `matching()`. You can create just the `INNER JOIN` that `matching()` uses with `innerJoinWith()`:

```
$query = $articles->find();
$query->innerJoinWith('Tags', function ($q) {
    return $q->where(['Tags.name' => 'CakePHP']);
});
```

`innerJoinWith()` allows you to the same parameters and dot notation:

```
$query = $products->find()->innerJoinWith(
    'Shops.Cities.Countries', function ($q) {
        return $q->where(['Countries.name' => 'Japan']);
    }
);
```

You can combine `innerJoinWith()` and `contain()` with the same association when you want to match specific records and load the associated data together. The example below matches Articles that have specific Tags and loads the same Tags:

```
$filter = ['Tags.name' => 'CakePHP'];
$query = $articles->find()
    ->distinct($articles->getPrimaryKey())
    ->contain('Tags', function (SelectQuery $q) use ($filter) {
        return $q->where($filter);
    })
    ->innerJoinWith('Tags', function (SelectQuery $q) use ($filter) {
        return $q->where($filter);
    });
```

Note: If you use `innerJoinWith()` and want to `select()` fields from that association, you need to use an alias for the field:

```
$query
    ->select(['country_name' => 'Countries.name'])
    ->innerJoinWith('Countries');
```

If you don't use an alias, you will see the data in `_matchingData` as described by `matching()` above. This is an edge case from `matching()` not knowing you manually selected the field.

Warning: You should not combine `innerJoinWith()` and `matching()` with the same association. This will produce multiple `INNER JOIN` statements and might not create the query you expected.

Using notMatching

The opposite of `matching()` is `notMatching()`. This function will change the query so that it filters results that have no relation to the specified association:

```
// In a controller or table method.

$query = $articlesTable
    ->find()
    ->notMatching('Tags', function ($q) {
        return $q->where(['Tags.name' => 'boring']);
    });
```

The above example will find all articles that were not tagged with the word `boring`. You can apply this method to HasMany associations as well. You could, for example, find all the authors with no published articles in the last 10 days:

```
$query = $authorsTable
    ->find()
    ->notMatching('Articles', function ($q) {
        return $q->where(['Articles.created >=' => new \DateTime('-10 days')]);
    });
```

It is also possible to use this method for filtering out records not matching deep associations. For example, you could find articles that have not been commented on by a certain user:

```
$query = $articlesTable
    ->find()
    ->notMatching('Comments.Users', function ($q) {
        return $q->where(['username' => 'jose']);
    });
```

Since articles with no comments at all also satisfy the condition above, you may want to combine `matching()` and `notMatching()` in the same query. The following example will find articles having at least one comment, but not commented by a certain user:

```
$query = $articlesTable
    ->find()
    ->notMatching('Comments.Users', function ($q) {
        return $q->where(['username' => 'jose']);
    })
    ->matching('Comments');
```

Note: As `notMatching()` will create a `LEFT JOIN`, you might want to consider calling `distinct` on the find query as you can get duplicate rows otherwise.

Keep in mind that contrary to the `matching()` function, `notMatching()` will not add any data to the `_matchingData` property in the results.

Using leftJoinWith

On certain occasions you may want to calculate a result based on an association, without having to load all the records for it. For example, if you wanted to load the total number of comments an article has along with all the article data, you can use the `leftJoinWith()` function:

```
$query = $articlesTable->find();
$query->select(['total_comments' => $query->func()->count('Comments.id')])
    ->leftJoinWith('Comments')
    ->groupBy(['Articles.id'])
    ->enableAutoFields(true);
```

The results for the above query will contain the article data and the `total_comments` property for each of them.

`leftJoinWith()` can also be used with deeply nested associations. This is useful, for example, for bringing the count of articles tagged with a certain word, per author:

```
$query = $authorsTable
    ->find()
    ->select(['total_articles' => $query->func()->count('Articles.id')])
    ->leftJoinWith('Articles.Tags', function ($q) {
        return $q->where(['Tags.name' => 'awesome']);
    })
    ->groupBy(['Authors.id'])
    ->enableAutoFields(true);
```

This function will not load any columns from the specified associations into the result set.

Adding Joins

In addition to loading related data with `contain()`, you can also add additional joins with the query builder:

```
$query = $articles->find()
    ->join([
        'table' => 'comments',
        'alias' => 'c',
        'type' => 'LEFT',
        'conditions' => 'c.article_id = articles.id',
    ]);
```

You can append multiple joins at the same time by passing an associative array with multiple joins:

```
$query = $articles->find()
    ->join([
        'c' => [
            'table' => 'comments',
            'type' => 'LEFT',
            'conditions' => 'c.article_id = articles.id',
        ],
        'u' => [
            'table' => 'users',
            'type' => 'INNER',
            'conditions' => 'u.id = articles.user_id',
```

(continues on next page)

(continued from previous page)

```
        ]
    ]);
```

As seen above, when adding joins the alias can be the outer array key. Join conditions can also be expressed as an array of conditions:

```
$query = $articles->find()
    ->join([
        'c' => [
            'table' => 'comments',
            'type' => 'LEFT',
            'conditions' => [
                'c.created >' => new DateTime('-5 days'),
                'c.moderated' => true,
                'c.article_id = articles.id'
            ]
        ],
    ], ['c.created' => 'datetime', 'c.moderated' => 'boolean']);
```

When creating joins by hand and using array based conditions, you need to provide the datatypes for each column in the join conditions. By providing datatypes for the join conditions, the ORM can correctly convert data types into SQL. In addition to `join()` you can use `rightJoin()`, `leftJoin()` and `innerJoin()` to create joins:

```
// Join with an alias and string conditions
$query = $articles->find();
$query->leftJoin(
    ['Authors' => 'authors'],
    ['Authors.id = Articles.author_id']);

// Join with an alias, array conditions, and 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',
    ]
);
```

It should be noted that if you set the `quoteIdentifiers` option to `true` when defining your `Connection`, join conditions between table fields should be set as follow:

```
$query = $articles->find()
    ->join([
        'c' => [
            'table' => 'comments',
            'type' => 'LEFT',
            'conditions' => [
```

(continues on next page)

(continued from previous page)

```
                'c.article_id' => new \Cake\Database\Expression\IdentifierExpression(
↪'articles.id'),
            ],
        ],
    ]);
```

This ensures that all of your identifiers will be quoted across the Query, avoiding errors with some database Drivers (PostgreSQL notably)

Inserting Data

Unlike earlier examples, you should can't use `find()` to create insert queries. Instead, create a new `InsertQuery` object using `insertQuery()`:

```
$query = $articles->insertQuery();
$query->insert(['title', 'body'])
    ->values([
        'title' => 'First post',
        'body' => 'Some body text',
    ])
    ->execute();
```

To insert multiple rows with only one query, you can chain the `values()` method as many times as you need:

```
$query = $articles->insertQuery();
$query->insert(['title', 'body'])
    ->values([
        'title' => 'First post',
        'body' => 'Some body text',
    ])
    ->values([
        'title' => 'Second post',
        'body' => 'Another body text',
    ])
    ->execute();
```

Generally, it is easier to insert data using entities and *save()*. By composing a `SELECT` and `INSERT` query together, you can create `INSERT INTO ... SELECT` style queries:

```
$select = $articles->find()
    ->select(['title', 'body', 'published'])
    ->where(['id' => 3]);

$query = $articles->insertQuery()
    ->insert(['title', 'body', 'published'])
    ->values($select)
    ->execute();
```

Note: Inserting records with the query builder will not trigger events such as `Model.afterSave`. Instead you should use the *ORM to save data*.

Updating Data

As with insert queries, you should not use `find()` to create update queries. Instead, create new a `Query` object using `updateQuery()`:

```
$query = $articles->updateQuery();
$query->set(['published' => true])
    ->where(['id' => $id])
    ->execute();
```

Generally, it is easier to update data using entities and `patchEntity()`.

Note: Updating records with the query builder will not trigger events such as `Model.afterSave`. Instead you should use the *ORM to save data*.

Deleting Data

As with insert queries, you can't use `find()` to create delete queries. Instead, create new a query object using `deleteQuery()`:

```
$query = $articles->deleteQuery();
$query->where(['id' => $id])
    ->execute();
```

Generally, it is easier to delete data using entities and *delete()*.

SQL Injection Prevention

While the ORM and database abstraction layers prevent most SQL injections issues, it is still possible to leave yourself vulnerable through improper use.

When using condition arrays, the key/left-hand side as well as single value entries must not contain user data:

```
$query->where([
    // Data on the key/left-hand side is unsafe, as it will be
    // inserted into the generated query as-is
    $userData => $value,

    // The same applies to single value entries, they are not
    // safe to use with user data in any form
    $userData,
    "MATCH (comment) AGAINST ($userData)",
    'created < NOW() - ' . $userData
]);
```

When using the expression builder, column names must not contain user data:

```
$query->where(function (QueryExpression $exp) use ($userData, $values) {
    // Column names in all expressions are not safe.
    return $exp->in($userData, $values);
});
```

When building function expressions, function names should never contain user data:

```
// Not safe.
$query->func()->{$userData}($arg1);

// Also not safe to use an array of
// user data in a function expression
$query->func()->coalesce($userData);
```

Raw expressions are never safe:

```
$expr = $query->newExpr()->add($userData);
$query->select(['two' => $expr]);
```

Binding values

It is possible to protect against many unsafe situations by using bindings. Values can be bound to queries using the `Cake\Database\Query::bind()` method.

The following example would be a safe variant of the unsafe, SQL injection prone example given above:

```
$query
    ->where([
        'MATCH (comment) AGAINST (:userData)',
        'created < NOW() - :moreUserData',
    ])
    ->bind(':userData', $userData, 'string')
    ->bind(':moreUserData', $moreUserData, 'datetime');
```

Note: Unlike `Cake\Database\StatementInterface::bindValue()`, `Query::bind()` requires to pass the named placeholders including the colon!

More Complex Queries

If your application requires using more complex queries, you can express many complex queries using the ORM query builder.

Unions

Unions are created by composing one or more select queries together:

```
$inReview = $articles->find()
    ->where(['need_review' => true]);

$unpublished = $articles->find()
    ->where(['published' => false]);

$unpublished->union($inReview);
```

You can create `UNION ALL` queries using the `unionAll()` method:

```
$inReview = $articles->find()
    ->where(['need_review' => true]);

$unpublished = $articles->find()
    ->where(['published' => false]);

$unpublished->unionAll($inReview);
```

Subqueries

Subqueries enable you to compose queries together and build conditions and results based on the results of other queries:

```
$matchingComment = $articles->getAssociation('Comments')->find()
    ->select(['article_id'])
    ->distinct()
    ->where(['comment LIKE' => '%CakePHP%']);

// Use a subquery to create conditions
$query = $articles->find()
    ->where(['id IN' => $matchingComment]);

// Join the results of a subquery into another query.
// Giving the subquery an alias provides a way to reference
// results in subquery.
$query = $articles->find();
$query->from(['matches' => $matchingComment])
    ->innerJoin(
        ['Articles' =>  'articles'],
        ['Articles.id' => $query->identifier('matches.id') ]
    );
```

Subqueries are accepted anywhere a query expression can be used. For example, in the `select()`, `from()` and `join()` methods. The above example uses a standard `ORM\Query\SelectQuery` object that will generate aliases, these aliases can make referencing results in the outer query more complex. As of 4.2.0 you can use `Table::subquery()` to create a specialized query instance that will not generate aliases:

```
$comments = $articles->getAssociation('Comments')->getTarget();

$matchingComment = $comments->subquery()
    ->select(['article_id'])
    ->distinct()
    ->where(['comment LIKE' => '%CakePHP%']);

$query = $articles->find()
    ->where(['id IN' => $matchingComment]);
```

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()`.

Window Functions

Window functions allow you to perform calculations using rows related to the current row. They are commonly used to calculate totals or offsets on partial sets of rows in the query. For example if we wanted to find the date of the earliest and latest comment on each article we could use window functions:

```
$query = $articles->find();
$query->select([
    'Articles.id',
    'Articles.title',
    'Articles.user_id'
    'oldest_comment' => $query->func()
        ->min('Comments.created')
        ->partition('Comments.article_id'),
    'latest_comment' => $query->func()
        ->max('Comments.created')
        ->partition('Comments.article_id'),
])
->innerJoinWith('Comments');
```

The above would generate SQL similar to:

```
SELECT
    Articles.id,
    Articles.title,
    Articles.user_id
    MIN(Comments.created) OVER (PARTITION BY Comments.article_id) AS oldest_comment,
    MAX(Comments.created) OVER (PARTITION BY Comments.article_id) AS latest_comment,
FROM articles AS Articles
INNER JOIN comments AS Comments
```

Window expressions can be applied to most aggregate functions. Any aggregate function that cake abstracts with a wrapper in `FunctionsBuilder` will return an `AggregateExpression` which lets you attach window expressions. You can create custom aggregate functions through `FunctionsBuilder::aggregate()`.

These are the most commonly supported window features. Most features are provided by `AggregateExpresion`, but make sure you follow your database documentation on use and restrictions.

- `order($fields)` Order the aggregate group the same as a query ORDER BY.
- `partition($expressions)` Add one or more partitions to the window based on column names.
- `rows($start, $end)` Define a offset of rows that precede and/or follow the current row that should be included in the aggregate function.

- `range($start, $end)` Define a range of row values that precede and/or follow the current row that should be included in the aggregate function. This evaluates values based on the `order()` field.

If you need to re-use the same window expression multiple times you can create named windows using the `window()` method:

```
$query = $articles->find();

// Define a named window
$query->window('related_article', function ($window, $query) {
    $window->partition('Comments.article_id');

    return $window;
});

$query->select([
    'Articles.id',
    'Articles.title',
    'Articles.user_id'
    'oldest_comment' => $query->func()
        ->min('Comments.created')
        ->over('related_article'),
    'latest_comment' => $query->func()
        ->max('Comments.created')
        ->over('related_article'),
]);
```

Common Table Expressions

Common Table Expressions or CTE are useful when building reporting queries where you need to compose the results of several smaller query results together. They can serve a similar purpose to database views or subquery results. Common Table Expressions differ from derived tables and views in a couple ways:

1. Unlike views, you don't have to maintain schema for common table expressions. The schema is implicitly based on the result set of the table expression.
2. You can reference the results of a common table expression multiple times without incurring performance penalties unlike subquery joins.

As an example lets fetch a list of customers and the number of orders each of them has made. In SQL we would use:

```
WITH orders_per_customer AS (
    SELECT COUNT(*) AS order_count, customer_id FROM orders GROUP BY customer_id
)
SELECT name, orders_per_customer.order_count
FROM customers
INNER JOIN orders_per_customer ON orders_per_customer.customer_id = customers.id
```

To build that query with the ORM query builder we would use:

```
// Start the final query
$query = $this->Customers->find();

// Attach a common table expression
```

(continues on next page)

(continued from previous page)

```
$query->with(function ($cte) {
    // Create a subquery to use in our table expression
    $q = $this->Orders->subquery();
    $q->select([
        'order_count' => $q->func()->count('*'),
        'customer_id',
    ])
    ->groupBy('customer_id');

    // Attach the new query to the table expression
    return $cte
        ->name('orders_per_customer')
        ->query($q);
});

// Finish building the final query
$query->select([
    'name',
    'order_count' => 'orders_per_customer.order_count',
])
->join([
    // Define the join with our table expression
    'orders_per_customer' => [
        'table' => 'orders_per_customer',
        'conditions' => 'orders_per_customer.customer_id = Customers.id',
    ],
]);
```

If you need to build a recursive query (`WITH RECURSIVE ...`), chain `->recursive()` onto `return $cte`.

Executing Complex Queries

While the query builder makes most queries possible through builder methods, very complex queries can be tedious and complicated to build. You may want to *execute the desired SQL directly*.

Executing SQL directly allows you to fine tune the query that will be run. However, doing so doesn't let you use `contain` or other higher level ORM features.

Table Objects

class Cake\ORM**Table**

Table objects provide access to the collection of entities stored in a specific table. Each table in your application should have an associated Table class which is used to interact with a given table. If you do not need to customize the behavior of a given table CakePHP will generate a Table instance for you to use.

Before trying to use Table objects and the ORM, you should ensure that you have configured your *database connection*.

Basic Usage

To get started, create a Table class. These classes live in **src/Model/Table**. Tables are a type model collection specific to relational databases, and the main interface to your database in CakePHP's ORM. The most basic table class would look like:

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

use Cake\ORM\Table;

class ArticlesTable extends Table
{
}
```

Note that we did not tell the ORM which table to use for our class. By convention table objects will use a table that matches the lower cased and underscored version of the class name. In the above example the `articles` table will be used. If our table class was named `BlogPosts` your table should be named `blog_posts`. You can specify the table to use by using the `setTable()` method:

```
namespace App\Model\Table;

use Cake\ORM\Table;

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

No inflection conventions will be applied when specifying a table. By convention the ORM also expects each table to have a primary key with the name of `id`. If you need to modify this you can use the `setPrimaryKey()` method:

```
namespace App\Model\Table;

use Cake\ORM\Table;

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

Customizing the Entity Class a Table Uses

By default table objects use an entity class based on naming conventions. For example if your table class is called `ArticlesTable` the entity would be `Article`. If the table class was `PurchaseOrdersTable` the entity would be `PurchaseOrder`. If however, you want to use an entity that doesn't follow the conventions you can use the `setEntityClass()` method to change things up:

```
class PurchaseOrdersTable extends Table
{
    public function initialize(array $config): void
    {
        $this->setEntityClass('App\Model\Entity\PO');
    }
}
```

As seen in the examples above Table objects have an `initialize()` method which is called at the end of the constructor. It is recommended that you use this method to do initialization logic instead of overriding the constructor.

Getting Instances of a Table Class

Before you can query a table, you'll need to get an instance of the table. You can do this by using the `TableLocator` class:

```
// In a controller

$articles = $this->fetchTable('Articles');
```

`TableLocator` provides the various dependencies for constructing a table, and maintains a registry of all the constructed table instances making it easier to build relations and configure the ORM. See *Using the TableLocator* for more information.

If your table class is in a plugin, be sure to use the correct name for your table class. Failing to do so can result in validation rules, or callbacks not being triggered as a default class is used instead of your actual class. To correctly load plugin table classes use the following:

```
// Plugin table
$articlesTable = $this->fetchTable('PluginName.Articles');

// Vendor prefixed plugin table
$articlesTable = $this->fetchTable('VendorName/PluginName.Articles');
```

Lifecycle Callbacks

As you have seen above table objects trigger a number of events. Events are useful if you want to hook into the ORM and add logic in without subclassing or overriding methods. Event listeners can be defined in table or behavior classes. You can also use a table's event manager to bind listeners in.

When using callback methods behaviors attached in the `initialize()` method will have their listeners fired **before** the table callback methods are triggered. This follows the same sequencing as controllers & components.

To add an event listener to a Table class or Behavior simply implement the method signatures as described below. See the *Events System* for more detail on how to use the events subsystem:

```php
// In a controller
$articles->save($article, ['customVariable1' => 'yourValue1']);

// In ArticlesTable.php
public function afterSave(Event $event, EntityInterface $entity, ArrayObject $options)
{
    $customVariable = $options['customVariable1'];  // 'yourValue1'
    $options['customVariable2'] = 'yourValue2';
}

public function afterSaveCommit(Event $event, EntityInterface $entity, ArrayObject
↪$options)
{
    $customVariable = $options['customVariable1'];  // 'yourValue1'
    $customVariable = $options['customVariable2'];  // 'yourValue2'
}
```

Event List

- `Model.initialize`
- `Model.beforeMarshal`
- `Model.afterMarshal`
- `Model.beforeFind`
- `Model.buildValidator`
- `Model.buildRules`
- `Model.beforeRules`
- `Model.afterRules`
- `Model.beforeSave`
- `Model.afterSave`
- `Model.afterSaveCommit`
- `Model.beforeDelete`
- `Model.afterDelete`
- `Model.afterDeleteCommit`

initialize

`Cake\ORM\Table::`**`initialize`**(*EventInterface $event*, *ArrayObject $data*, *ArrayObject $options*)

The `Model.initialize` event is fired after the constructor and initialize methods are called. The `Table` classes do not listen to this event by default, and instead use the `initialize` hook method.

To respond to the `Model.initialize` event you can create a listener class which implements `EventListenerInterface`:

```
use Cake\Event\EventListenerInterface;
class ModelInitializeListener implements EventListenerInterface
{
    public function implementedEvents()
    {
        return [
            'Model.initialize' => 'initializeEvent',
        ];
    }

    public function initializeEvent($event): void
    {
        $table = $event->getSubject();
        // do something here
    }
}
```

and attach the listener to the `EventManager` as below:

```
use Cake\Event\EventManager;
$listener = new ModelInitializeListener();
EventManager::instance()->attach($listener);
```

This will call the `initializeEvent` when any `Table` class is constructed.

beforeMarshal

`Cake\ORM\Table::`**`beforeMarshal`**(*EventInterface $event*, *ArrayObject $data*, *ArrayObject $options*)

The `Model.beforeMarshal` event is fired before request data is converted into entities. See the *Modifying Request Data Before Building Entities* documentation for more information.

afterMarshal

`Cake\ORM\Table::`**`afterMarshal`**(*EventInterface $event*, *EntityInterface $entity*, *ArrayObject $data*, *ArrayObject $options*)

The `Model.afterMarshal` event is fired after request data is converted into entities. Event handlers will get the converted entities, original request data and the options provided to the `patchEntity()` or `newEntity()` call.

beforeFind

`Cake\ORM\Table::`**`beforeFind`**(*EventInterface $event*, *SelectQuery $query*, *ArrayObject $options*, *boolean $primary*)

The `Model.beforeFind` event is fired before each find operation. By stopping the event, and feeding the query with a custom result set, you can bypass the find operation entirely:

```
public function beforeFind(EventInterface $event, SelectQuery $query, ArrayObject
↪$options, $primary)
{
```

(continues on next page)

(continued from previous page)

```
    if (/* ... */) {
        $event->stopPropagation();
        $query->setResult(new \Cake\Datasource\ResultSetDecorator([]));

        return;
    }
    // ...
}
```

In this example, no further `beforeFind` events will be triggered on the related table or its attached behaviors (though behavior events are usually invoked earlier given their default priorities), and the query will return the empty result set that was passed via `SelectQuery::setResult()`.

Any changes done to the `$query` instance will be retained for the rest of the find. The `$primary` parameter indicates whether or not this is the root query, or an associated query. All associations participating in a query will have a `Model.beforeFind` event triggered. For associations that use joins, a dummy query will be provided. In your event listener you can set additional fields, conditions, joins or result formatters. These options/features will be copied onto the root query.

In previous versions of CakePHP there was an `afterFind` callback, this has been replaced with the *Modifying Results with Map/Reduce* features and entity constructors.

buildValidator

`Cake\ORM\Table::`**`buildValidator`**`(`*EventInterface $event*, *Validator $validator*, *$name*`)`

The `Model.buildValidator` event is fired when `$name` validator is created. Behaviors, can use this hook to add in validation methods.

buildRules

`Cake\ORM\Table::`**`buildRules`**`(`*EventInterface $event*, *RulesChecker $rules*`)`

The `Model.buildRules` event is fired after a rules instance has been created and after the table's `buildRules()` method has been called.

beforeRules

`Cake\ORM\Table::`**`beforeRules`**`(`*EventInterface $event*, *EntityInterface $entity*, *ArrayObject $options*, *$operation*`)`

The `Model.beforeRules` event is fired before an entity has had rules applied. By stopping this event, you can halt the rules checking and set the result of applying rules.

afterRules

Cake\ORM\Table::**afterRules**(*EventInterface $event*, *EntityInterface $entity*, *ArrayObject $options*, *$result*, *$operation*)

The `Model.afterRules` event is fired after an entity has rules applied. By stopping this event, you can return the final value of the rules checking operation.

beforeSave

Cake\ORM\Table::**beforeSave**(*EventInterface $event*, *EntityInterface $entity*, *ArrayObject $options*)

The `Model.beforeSave` event is fired before each entity is saved. Stopping this event will abort the save operation. When the event is stopped the result of the event will be returned.

afterSave

Cake\ORM\Table::**afterSave**(*EventInterface $event*, *EntityInterface $entity*, *ArrayObject $options*)

The `Model.afterSave` event is fired after an entity is saved.

afterSaveCommit

Cake\ORM\Table::**afterSaveCommit**(*EventInterface $event*, *EntityInterface $entity*, *ArrayObject $options*)

The `Model.afterSaveCommit` event is fired after the transaction in which the save operation is wrapped has been committed. It's also triggered for non atomic saves where database operations are implicitly committed. The event is triggered only for the primary table on which `save()` is directly called. The event is not triggered if a transaction is started before calling save.

beforeDelete

Cake\ORM\Table::**beforeDelete**(*EventInterface $event*, *EntityInterface $entity*, *ArrayObject $options*)

The `Model.beforeDelete` event is fired before an entity is deleted. By stopping this event you will abort the delete operation. When the event is stopped the result of the event will be returned.

afterDelete

Cake\ORM\Table::**afterDelete**(*EventInterface $event*, *EntityInterface $entity*, *ArrayObject $options*)

The `Model.afterDelete` event is fired after an entity has been deleted.

afterDeleteCommit

Cake\ORM\Table::**afterDeleteCommit**(*EventInterface $event*, *EntityInterface $entity*, *ArrayObject $options*)

The `Model.afterDeleteCommit` event is fired after the transaction in which the delete operation is wrapped has been is committed. It's also triggered for non atomic deletes where database operations are implicitly committed. The event is triggered only for the primary table on which `delete()` is directly called. The event is not triggered if a transaction is started before calling delete.

Stopping Table Events

To prevent the save from continuing, simply stop event propagation in your callback:

```
public function beforeSave(EventInterface $event, EntityInterface $entity, ArrayObject
↪$options)
{
    if (...) {
        $event->stopPropagation();
        $event->setResult(false);
        return;
    }
    ...
}
```

Alternatively, you can return false from the callback. This has the same effect as stopping event propagation.

Callback priorities

When using events on your tables and behaviors be aware of the priority and the order listeners are attached. Behavior events are attached before Table events are. With the default priorities this means that Behavior callbacks are triggered **before** the Table event with the same name.

As an example, if your Table is using `TreeBehavior` the `TreeBehavior::beforeDelete()` method will be called before your table's `beforeDelete()` method, and you will not be able to work wth the child nodes of the record being deleted in your Table's method.

You can manage event priorities in one of a few ways:

1. Change the `priority` of a Behavior's listeners using the `priority` option. This will modify the priority of **all** callback methods in the Behavior:

   ```
   // In a Table initialize() method
   $this->addBehavior('Tree', [
       // Default value is 10 and listeners are dispatched from the
       // lowest to highest priority.
       'priority' => 2,
   ]);
   ```

2. Modify the `priority` in your `Table` class by using the `Model.implementedEvents()` method. This allows you to assign a different priority per callback-function:

   ```
   // In a Table class.
   public function implementedEvents()
   ```

(continues on next page)

(continued from previous page)

```
{
    $events = parent::implementedEvents();
    $events['Model.beforeDelete'] = [
        'callable' => 'beforeDelete',
        'priority' => 3
    ];

    return $events;
}
```

Behaviors

Cake\ORM\Table::**addBehavior**(*$name*, *array $options = []*)

Behaviors provide a way to create horizontally re-usable pieces of logic related to table classes. You may be wondering why behaviors are regular classes and not traits. The primary reason for this is event listeners. While traits would allow for re-usable pieces of logic, they would complicate binding events.

To add a behavior to your table you can call the `addBehavior()` method. Generally the best place to do this is in the `initialize()` method:

```
namespace App\Model\Table;

use Cake\ORM\Table;

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

As with associations, you can use *plugin syntax* and provide additional configuration options:

```
namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{
    public function initialize(array $config): void
    {
        $this->addBehavior('Timestamp', [
            'events' => [
                'Model.beforeSave' => [
                    'created_at' => 'new',
                    'modified_at' => 'always'
                ]
            ]
        ]);
```

(continues on next page)

(continued from previous page)

```
    }
}
```

You can find out more about behaviors, including the behaviors provided by CakePHP in the chapter on *Behaviors*.

Configuring Connections

By default all table instances use the `default` database connection. If your application uses multiple database connections you will want to configure which tables use which connections. This is the `defaultConnectionName()` method:

```
namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{
    public static function defaultConnectionName(): string {
        return 'replica_db';
    }
}
```

Note: The `defaultConnectionName()` method **must** be static.

Using the TableLocator

`class` Cake\ORM**TableLocator**

As we've seen earlier, the TableLocator class provides a way to use a factory/registry for accessing your applications table instances. It provides a few other useful features as well.

Configuring Table Objects

Cake\ORM\TableLocator::**get**(*$alias*, *$config*)

When loading tables from the registry you can customize their dependencies, or use mock objects by providing an `$options` array:

```
$articles = FactoryLocator::get('Table')->get('Articles', [
    'className' => 'App\Custom\ArticlesTable',
    'table' => 'my_articles',
    'connection' => $connectionObject,
    'schema' => $schemaObject,
    'entityClass' => 'Custom\EntityClass',
    'eventManager' => $eventManager,
    'behaviors' => $behaviorRegistry
]);
```

Pay attention to the connection and schema configuration settings, they aren't string values but objects. The connection will take an object of `Cake\Database\Connection` and schema `Cake\Database\Schema\Collection`.

Note: If your table also does additional configuration in its `initialize()` method, those values will overwrite the ones provided to the registry.

You can also pre-configure the registry using the `setConfig()` method. Configuration data is stored *per alias*, and can be overridden by an object's `initialize()` method:

```
FactoryLocator::get('Table')->setConfig('Users', ['table' => 'my_users']);
```

Note: You can only configure a table before or during the **first** time you access that alias. Doing it after the registry is populated will have no effect.

Flushing the Registry

`Cake\ORM\TableLocator::clear()`

During test cases you may want to flush the registry. Doing so is often useful when you are using mock objects, or modifying a table's dependencies:

```
FactoryLocator::get('Table')->clear();
```

Configuring the Namespace to Locate ORM classes

If you have not followed the conventions it is likely that your Table or Entity classes will not be detected by CakePHP. In order to fix this, you can set a namespace with the `Cake\Core\Configure::write` method. As an example:

```
/src
    /App
        /My
            /Namespace
                /Model
                    /Entity
                    /Table
```

Would be configured with:

```
Cake\Core\Configure::write('App.namespace', 'App\My\Namespace');
```

Entities

class Cake\ORM**Entity**

While *Table Objects* represent and provide access to a collection of objects, entities represent individual rows or domain objects in your application. Entities contain methods to manipulate and access the data they contain. Fields can also be accessed as properties on the object.

Entities are created for you each time you iterate the query instance returned by `find()` of a table object or when you call `all()` or `first()` method of the query instance.

Creating Entity Classes

You don't need to create entity classes to get started with the ORM in CakePHP. However, if you want to have custom logic in your entities you will need to create classes. By convention entity classes live in **src/Model/Entity/**. If our application had an `articles` table we could create the following entity:

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

use Cake\ORM\Entity;

class Article extends Entity
{
}
```

Right now this entity doesn't do very much. However, when we load data from our articles table, we'll get instances of this class.

Note: If you don't define an entity class CakePHP will use the basic Entity class.

Creating Entities

Entities can be directly instantiated:

```
use App\Model\Entity\Article;

$article = new Article();
```

When instantiating an entity you can pass the fields with the data you want to store in them:

```
use App\Model\Entity\Article;

$article = new Article([
    'id' => 1,
    'title' => 'New Article',
    'created' => new DateTime('now')
]);
```

The preferred way of getting new entities is using the `newEmptyEntity()` method from the `Table` objects:

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

$article = $this->fetchTable('Articles')->newEmptyEntity();

$article = $this->fetchTable('Articles')->newEntity([
    'id' => 1,
    'title' => 'New Article',
    'created' => new DateTime('now')
]);
```

`$article` will be an instance of `App\Model\Entity\Article` or fallback to `Cake\ORM\Entity` instance if you haven't created the `Article` class.

Note: Prior to CakePHP 4.3 you need to use `$this->getTableLocator->get('Articles')` to get the table instance.

Accessing Entity Data

Entities provide a few ways to access the data they contain. Most commonly you will access the data in an entity using object notation:

```
use App\Model\Entity\Article;

$article = new Article;
$article->title = 'This is my first post';
echo $article->title;
```

You can also use the `get()` and `set()` methods.

`Cake\ORM\Entity::set`(*$field*, *$value = null*, *array $options = []*)

`Cake\ORM\Entity::get`(*$field*)

For example:

```
$article->set('title', 'This is my first post');
echo $article->get('title');
```

When using `set()` you can update multiple fields at once using an array:

```
$article->set([
    'title' => 'My first post',
    'body' => 'It is the best ever!'
]);
```

Warning: When updating entities with request data you should configure which fields can be set with mass assignment.

You can check if fields are defined in your entities with `has()`:

```
$article = new Article([
    'title' => 'First post',
    'user_id' => null
]);
$article->has('title'); // true
$article->has('user_id'); // true
$article->has('undefined'); // false
```

The `has()` method will return `true` if a field is defined. You can use `isEmpty()` and `hasValue()` to check if a field contains a 'non-empty' value:

```
$article = new Article([
    'title' => 'First post',
    'user_id' => null,
    'text' => '',
    'links' => [],
]);
$article->has('title'); // true
$article->isEmpty('title');  // false
$article->hasValue('title'); // true

$article->has('user_id'); // false
$article->isEmpty('user_id');  // true
$article->hasValue('user_id'); // false

$article->has('text'); // true
$article->isEmpty('text');  // true
$article->hasValue('text'); // false

$article->has('links'); // true
$article->isEmpty('links');  // true
$article->hasValue('links'); // false
```

If you often partially load entities you should enable strict-property access behavior to ensure you're not using properties that haven't been loaded. On a per-entity basis you can enable this behavior:

```
$article->requireFieldPresence();
```

Once enabled, accessing properties that are not defined will raise a `CakeORMMissingPropertyException`.

Accessors & Mutators

In addition to the simple get/set interface, entities allow you to provide accessors and mutator methods. These methods let you customize how fields are read or set.

Accessors

Accessors let you customize how fields are read. They use the convention of `_get(FieldName)` with `(FieldName)` being the CamelCased version (multiple words are joined together to a single word with the first letter of each word capitalized) of the field name.

They receive the basic value stored in the `_fields` array as their only argument. For example:

```
namespace App\Model\Entity;

use Cake\ORM\Entity;

class Article extends Entity
{
    protected function _getTitle($title)
    {
        return strtoupper($title);
    }
}
```

The example above converts the value of the `title` field to an uppercase version each time it is read. It would be run when getting the field through any of these two ways:

```
echo $article->title; // returns FOO instead of foo
echo $article->get('title'); // returns FOO instead of foo
```

Note: Code in your accessors is executed each time you reference the field. You can use a local variable to cache it if you are performing a resource-intensive operation in your accessor like this: *$myEntityProp = $entity->my_property*.

Warning: Accessors will be used when saving entities, so be careful when defining methods that format data, as the formatted data will be persisted.

Mutators

You can customize how fields get set by defining a mutator. They use the convention of `_set(FieldName)` with `(FieldName)` being the CamelCased version of the field name.

Mutators should always return the value that should be stored in the field. You can also use mutators to set other fields. When doing this, be careful to not introduce any loops, as CakePHP will not prevent infinitely looping mutator methods. For example:

```
namespace App\Model\Entity;

use Cake\ORM\Entity;
```

(continues on next page)

(continued from previous page)

```
use Cake\Utility\Text;

class Article extends Entity
{
    protected function _setTitle($title)
    {
        $this->slug = Text::slug($title);

        return strtoupper($title);
    }
}
```

The example above is doing two things: It stores a modified version of the given value in the `slug` field and stores an uppercase version in the `title` field. It would be run when setting the field through any of these two ways:

```
$user->title = 'foo'; // sets slug field and stores FOO instead of foo
$user->set('title', 'foo'); // sets slug field and stores FOO instead of foo
```

> **Warning:** Accessors are also run before entities are persisted to the database. If you want to transform fields but not persist that transformation, we recommend using virtual fields as those are not persisted.

Creating Virtual Fields

By defining accessors you can provide access to fields that do not actually exist. For example if your users table has `first_name` and `last_name` you could create a method for the full name:

```
namespace App\Model\Entity;

use Cake\ORM\Entity;

class User extends Entity
{
    protected function _getFullName()
    {
        return $this->first_name . '  ' . $this->last_name;
    }
}
```

You can access virtual fields as if they existed on the entity. The property name will be the lower case and underscored version of the method (`full_name`):

```
echo $user->full_name;
echo $user->get('full_name');
```

Do bear in mind that virtual fields cannot be used in finds. If you want them to be part of JSON or array representations of your entities, see *Exposing Virtual Fields*.

Checking if an Entity Has Been Modified

Cake\ORM\Entity::**dirty**(*$field = null*, *$dirty = null*)

You may want to make code conditional based on whether or not fields have changed in an entity. For example, you may only want to validate fields when they change:

```
// See if the title has been modified.
$article->isDirty('title');
```

You can also flag fields as being modified. This is handy when appending into array fields as this wouldn't automatically mark the field as dirty, only exchanging completely would.:

```
// Add a comment and mark the field as changed.
$article->comments[] = $newComment;
$article->setDirty('comments', true);
```

In addition you can also base your conditional code on the original field values by using the `getOriginal()` method. This method will either return the original value of the field if it has been modified or its actual value.

You can also check for changes to any field in the entity:

```
// See if the entity has changed
$article->isDirty();
```

To remove the dirty mark from fields in an entity, you can use the `clean()` method:

```
$article->clean();
```

When creating a new entity, you can avoid the fields from being marked as dirty by passing an extra option:

```
$article = new Article(['title' => 'New Article'], ['markClean' => true]);
```

To get a list of all dirty fields of an `Entity` you may call:

```
$dirtyFields = $entity->getDirty();
```

Validation Errors

After you *save an entity* any validation errors will be stored on the entity itself. You can access any validation errors using the `getErrors()`, `getError()` or `hasErrors()` methods:

```
// Get all the errors
$errors = $user->getErrors();

// Get the errors for a single field.
$errors = $user->getError('password');

// Does the entity or any nested entity have an error.
$user->hasErrors();

// Does only the root entity have an error
$user->hasErrors(false);
```

The `setErrors()` or `setError()` method can also be used to set the errors on an entity, making it easier to test code that works with error messages:

```
$user->setError('password', ['Password is required']);
$user->setErrors([
    'password' => ['Password is required'],
    'username' => ['Username is required']
]);
```

Mass Assignment

While setting fields to entities in bulk is simple and convenient, it can create significant security issues. Bulk assigning user data from the request into an entity allows the user to modify any and all columns. When using anonymous entity classes or creating the entity class with the *Bake Console* CakePHP does not protect against mass-assignment.

The `_accessible` property allows you to provide a map of fields and whether or not they can be mass-assigned. The values `true` and `false` indicate whether a field can or cannot be mass-assigned:

```
namespace App\Model\Entity;

use Cake\ORM\Entity;

class Article extends Entity
{
    protected array $_accessible = [
        'title' => true,
        'body' => true
    ];
}
```

In addition to concrete fields there is a special * field which defines the fallback behavior if a field is not specifically named:

```
namespace App\Model\Entity;

use Cake\ORM\Entity;

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

Note: If the * field is not defined it will default to `false`.

Avoiding Mass Assignment Protection

When creating a new entity using the `new` keyword you can tell it to not protect itself against mass assignment:

```
use App\Model\Entity\Article;

$article = new Article(['id' => 1, 'title' => 'Foo'], ['guard' => false]);
```

Modifying the Guarded Fields at Runtime

You can modify the list of guarded fields at runtime using the `setAccess()` method:

```
// Make user_id accessible.
$article->setAccess('user_id', true);

// Make title guarded.
$article->setAccess('title', false);
```

Note: Modifying accessible fields affects only the instance the method is called on.

When using the `newEntity()` and `patchEntity()` methods in the `Table` objects you can customize mass assignment protection with options. Please refer to the *Changing Accessible Fields* section for more information.

Bypassing Field Guarding

There are some situations when you want to allow mass-assignment to guarded fields:

```
$article->set($fields, ['guard' => false]);
```

By setting the `guard` option to `false`, you can ignore the accessible field list for a single call to `set()`.

Checking if an Entity was Persisted

It is often necessary to know if an entity represents a row that is already in the database. In those situations use the `isNew()` method:

```
if (!$article->isNew()) {
    echo 'This article was saved already!';
}
```

If you are certain that an entity has already been persisted, you can use `setNew()`:

```
$article->setNew(false);

$article->setNew(true);
```

Lazy Loading Associations

While eager loading associations is generally the most efficient way to access your associations, there may be times when you need to lazily load associated data. Before we get into how to lazy load associations, we should discuss the differences between eager loading and lazy loading associations:

Eager loading
: Eager loading uses joins (where possible) to fetch data from the database in as *few* queries as possible. When a separate query is required, like in the case of a HasMany association, a single query is emitted to fetch *all* the associated data for the current set of objects.

Lazy loading
: Lazy loading defers loading association data until it is absolutely required. While this can save CPU time because possibly unused data is not hydrated into objects, it can result in many more queries being emitted to the database. For example looping over a set of articles & their comments will frequently emit N queries where N is the number of articles being iterated.

While lazy loading is not included by CakePHP's ORM, you can just use one of the community plugins to do so. We recommend the LazyLoad Plugin[127]

After adding the plugin to your entity, you will be able to do the following:

```
$article = $this->Articles->findById($id);

// The comments property was lazy loaded
foreach ($article->comments as $comment) {
    echo $comment->body;
}
```

Creating Re-usable Code with Traits

You may find yourself needing the same logic in multiple entity classes. PHP's traits are a great fit for this. You can put your application's traits in **src/Model/Entity**. By convention traits in CakePHP are suffixed with `Trait` so they can be discernible from classes or interfaces. Traits are often a good complement to behaviors, allowing you to provide functionality for the table and entity objects.

For example if we had SoftDeletable plugin, it could provide a trait. This trait could give methods for marking entities as 'deleted', the method `softDelete` could be provided by a trait:

```
// SoftDelete/Model/Entity/SoftDeleteTrait.php

namespace SoftDelete\Model\Entity;

trait SoftDeleteTrait
{
    public function softDelete()
    {
        $this->set('deleted', true);
    }
}
```

You could then use this trait in your entity class by importing it and including it:

[127] https://github.com/jeremyharris/cakephp-lazyload

```
namespace App\Model\Entity;

use Cake\ORM\Entity;
use SoftDelete\Model\Entity\SoftDeleteTrait;

class Article extends Entity
{
    use SoftDeleteTrait;
}
```

Converting to Arrays/JSON

When building APIs, you may often need to convert entities into arrays or JSON data. CakePHP makes this simple:

```
// Get an array.
// Associations will be converted with toArray() as well.
$array = $user->toArray();

// Convert to JSON
// Associations will be converted with jsonSerialize hook as well.
$json = json_encode($user);
```

When converting an entity to an JSON, the virtual & hidden field lists are applied. Entities are recursively converted to JSON as well. This means that if you eager loaded entities and their associations CakePHP will correctly handle converting the associated data into the correct format.

Exposing Virtual Fields

By default virtual fields are not exported when converting entities to arrays or JSON. In order to expose virtual fields you need to make them visible. When defining your entity class you can provide a list of virtual field that should be exposed:

```
namespace App\Model\Entity;

use Cake\ORM\Entity;

class User extends Entity
{
    protected $_virtual = ['full_name'];
}
```

This list can be modified at runtime using the `setVirtual()` method:

```
$user->setVirtual(['full_name', 'is_admin']);
```

Hiding Fields

There are often fields you do not want exported in JSON or array formats. For example it is often unwise to expose password hashes or account recovery questions. When defining an entity class, define which fields should be hidden:

```
namespace App\Model\Entity;

use Cake\ORM\Entity;

class User extends Entity
{
    protected $_hidden = ['password'];
}
```

This list can be modified at runtime using the `setHidden()` method:

```
$user->setHidden(['password', 'recovery_question']);
```

Storing Complex Types

Accessor & Mutator methods on entities are not intended to contain the logic for serializing and unserializing complex data coming from the database. Refer to the *Saving Complex Types* section to understand how your application can store more complex data types like arrays and objects.

Associations - Linking Tables Together

Defining relations between different objects in your application should be a natural process. For example, an article may have many comments, and belong to an author. Authors may have many articles and comments. The four association types in CakePHP are: hasOne, hasMany, belongsTo, and belongsToMany.

Relationship	Association Type	Example
one to one	hasOne	A user has one profile.
one to many	hasMany	A user can have multiple articles.
many to one	belongsTo	Many articles belong to a user.
many to many	belongsToMany	Tags belong to many articles.

Associations are defined during the `initialize()` method of your table object. Methods matching the association type allow you to define the associations in your application. For example if we wanted to define a belongsTo association in our ArticlesTable:

```
namespace App\Model\Table;

use Cake\ORM\Table;

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

The simplest form of any association setup takes the table alias you want to associate with. By default all of the details of an association will use the CakePHP conventions. If you want to customize how your associations are handled you can modify them with setters:

```
class ArticlesTable extends Table
{
    public function initialize(array $config): void
    {
        $this->belongsTo('Authors', [
                'className' => 'Publishing.Authors'
            ])
            ->setForeignKey('author_id')
            ->setProperty('author');
    }
}
```

The property name will be the property key (of the associated entity) on the entity object, in this case:

```
$authorEntity = $articleEntity->author;
```

You can also use arrays to customize your associations:

```
$this->belongsTo('Authors', [
    'className' => 'Publishing.Authors',
    'foreignKey' => 'author_id',
    'propertyName' => 'author'
]);
```

However, arrays do not offer the typehinting and autocomplete benefits that the fluent interface does.

The same table can be used multiple times to define different types of associations. For example consider a case where you want to separate approved comments and those that have not been moderated yet:

```
class ArticlesTable extends Table
{
    public function initialize(array $config): void
    {
        $this->hasMany('Comments')
            ->setFinder('approved');

        $this->hasMany('UnapprovedComments', [
                'className' => 'Comments'
            ])
            ->setFinder('unapproved')
            ->setProperty('unapproved_comments');
    }
}
```

As you can see, by specifying the `className` key, it is possible to use the same table as different associations for the same table. You can even create self-associated tables to create parent-child relationships:

```
class CategoriesTable extends Table
{
    public function initialize(array $config): void
    {
```

(continues on next page)

(continued from previous page)

```
        $this->hasMany('SubCategories', [
            'className' => 'Categories',
        ]);

        $this->belongsTo('ParentCategories', [
            'className' => 'Categories',
        ]);
    }
}
```

You can also setup associations in mass by making a single call to `Table::addAssociations()` which accepts an array containing a set of table names indexed by association type as an argument:

```
class PostsTable extends Table
{
    public function initialize(array $config): void
    {
       $this->addAssociations([
           'belongsTo' => [
               'Users' => ['className' => 'App\Model\Table\UsersTable'],
           ],
           'hasMany' => ['Comments'],
           'belongsToMany' => ['Tags'],
       ]);
    }
}
```

Each association type accepts multiple associations where the keys are the aliases, and the values are association config data. If numeric keys are used the values will be treated as association aliases.

HasOne Associations

Let's set up a Users table with a hasOne relationship to the Addresses table.

First, your database tables need to be keyed correctly. For a hasOne relationship to work, one table has to contain a foreign key that points to a record in the other table. In this case, the Addresses table will contain a field called 'user_id'. The basic pattern is:

hasOne: the *other* model contains the foreign key.

Relation	Schema
Users hasOne Addresses	addresses.user_id
Doctors hasOne Mentors	mentors.doctor_id

Note: It is not mandatory to follow CakePHP conventions, you can override the name of any `foreignKey` in your associations definitions. Nevertheless, sticking to conventions will make your code less repetitive, easier to read and to maintain.

Once you create the `UsersTable` and `AddressesTable` classes, you can make the association with the following code:

```
class UsersTable extends Table
{
    public function initialize(array $config): void
    {
        $this->hasOne('Addresses');
    }
}
```

If you need more control, you can define your associations using the setters. For example, you might want to limit the association to include only certain records:

```
class UsersTable extends Table
{
    public function initialize(array $config): void
    {
        $this->hasOne('Addresses')
            ->setName('Addresses')
            ->setFinder('primary')
            ->setDependent(true);
    }
}
```

If you want to break different addresses into multiple associations, you can do something like:

```
class UsersTable extends Table
{
    public function initialize(array $config): void
    {
        $this->hasOne('HomeAddresses', [
                'className' => 'Addresses'
            ])
            ->setProperty('home_address')
            ->setConditions(['HomeAddresses.label' => 'Home'])
            ->setDependent(true);

        $this->hasOne('WorkAddresses', [
                'className' => 'Addresses'
            ])
            ->setProperty('work_address')
            ->setConditions(['WorkAddresses.label' => 'Work'])
            ->setDependent(true);
    }
}
```

Note: If a column is shared by multiple hasOne associations, you must qualify it with the association alias. In the above example, the 'label' column is qualified with the 'HomeAddresses' and 'WorkAddresses' aliases.

Possible keys for hasOne association arrays include:

- **className**: The class name of the other table. This is the same name used when getting an instance of the table. In the 'Users hasOne Addresses' example, it should be 'Addresses'. The default value is the name of the association.

- **foreignKey**: The name of the foreign key column in the other table. The default value is the underscored, singular name of the current model, suffixed with '_id' such as 'user_id' in the above example.
- **bindingKey**: The name of the column in the current table used to match the `foreignKey`. The default value is the primary key of the current table such as 'id' of Users in the above example.
- **conditions**: An array of find() compatible conditions such as `['Addresses.primary' => true]`
- **joinType**: The type of the join used in the SQL query. Accepted values are 'LEFT' and 'INNER'. You can use 'INNER' to get results only where the association is set. The default value is 'LEFT'.
- **dependent**: When the dependent key is set to `true`, and an entity is deleted, the associated model records are also deleted. In this case we set it to `true` so that deleting a User will also delete her associated Address.
- **cascadeCallbacks**: When this and **dependent** are `true`, cascaded deletes will load and delete entities so that callbacks are properly triggered. When `false`, `deleteAll()` is used to remove associated data and no callbacks are triggered.
- **propertyName**: The property name that should be filled with data from the associated table into the source table results. By default this is the underscored & singular name of the association so `address` in our example.
- **strategy**: The query strategy used to load matching record from the other table. Accepted values are `'join'` and `'select'`. Using `'select'` will generate a separate query and can be useful when the other table is in different database. The default is `'join'`.
- **finder**: The finder method to use when loading associated records.

Once this association has been defined, find operations on the Users table can contain the Address record if it exists:

```
// In a controller or table method.
$query = $users->find('all')->contain(['Addresses'])->all();
foreach ($query as $user) {
    echo $user->address->street;
}
```

The above would emit SQL that is similar to:

```
SELECT * FROM users INNER JOIN addresses ON addresses.user_id = users.id;
```

BelongsTo Associations

Now that we have Address data access from the User table, let's define a belongsTo association in the Addresses table in order to get access to related User data. The belongsTo association is a natural complement to the hasOne and hasMany associations - it allows us to see related data from the other direction.

When keying your database tables for a belongsTo relationship, follow this convention:

belongsTo: the *current* model contains the foreign key.

Relation	Schema
Addresses belongsTo Users	addresses.user_id
Mentors belongsTo Doctors	mentors.doctor_id

Tip: If a table contains a foreign key, it belongs to the other table.

We can define the belongsTo association in our Addresses table as follows:

```
class AddressesTable extends Table
{
    public function initialize(array $config): void
    {
        $this->belongsTo('Users');
    }
}
```

We can also define a more specific relationship using the setters:

```
class AddressesTable extends Table
{
    public function initialize(array $config): void
    {
        $this->belongsTo('Users')
            ->setForeignKey('user_id')
            ->setJoinType('INNER');
    }
}
```

Possible keys for belongsTo association arrays include:

- **className**: The class name of the other table. This is the same name used when getting an instance of the table. In the 'Addresses belongsTo Users' example, it should be 'Users'. The default value is the name of the association.
- **foreignKey**: The name of the foreign key column in the current table. The default value is the underscored, singular name of the other model, suffixed with '_id' such as 'user_id' in the above example.
- **bindingKey**: The name of the column in the other table used to match the `foreignKey`. The default value is the primary key of the other table such as 'id' of Users in the above example.
- **conditions**: An array of find() compatible conditions or SQL strings such as `['Users.active' => true]`
- **joinType**: The type of the join used in the SQL query. Accepted values are 'LEFT' and 'INNER'. You can use 'INNER' to get results only where the association is set. The default value is 'LEFT'.
- **propertyName**: The property name that should be filled with data from the associated table into the source table results. By default this is the underscored & singular name of the association so `user` in our example.
- **strategy**: The query strategy used to load matching record from the other table. Accepted values are `'join'` and `'select'`. Using `'select'` will generate a separate query and can be useful when the other table is in different database. The default is `'join'`.
- **finder**: The finder method to use when loading associated records.

Once this association has been defined, find operations on the Addresses table can contain the User record if it exists:

```
// In a controller or table method.
$query = $addresses->find('all')->contain(['Users'])->all();
foreach ($query as $address) {
    echo $address->user->username;
}
```

The above would output SQL similar to:

```
SELECT * FROM addresses LEFT JOIN users ON addresses.user_id = users.id;
```

HasMany Associations

An example of a hasMany association is "Articles hasMany Comments". Defining this association will allow us to fetch an article's comments when the article is loaded.

When creating your database tables for a hasMany relationship, follow this convention:

hasMany: the *other* model contains the foreign key.

Relation	Schema
Articles hasMany Comments	Comments.article_id
Products hasMany Options	Options.product_id
Doctors hasMany Patients	Patients.doctor_id

We can define the hasMany association in our Articles model as follows:

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

We can also define a more specific relationship using the setters:

```
class ArticlesTable extends Table
{
    public function initialize(array $config): void
    {
        $this->hasMany('Comments')
            ->setForeignKey('article_id')
            ->setDependent(true);
    }
}
```

Sometimes you may want to configure composite keys in your associations:

```
// Within ArticlesTable::initialize() call
$this->hasMany('Comments')
    ->setForeignKey([
        'article_id',
        'article_hash',
    ]);
```

Relying on the example above, we have passed an array containing the desired composite keys to `setForeignKey()`. By default the `bindingKey` would be automatically defined as `id` and `hash` respectively, but let's assume that you need to specify different binding fields than the defaults. You can setup it manually with `setBindingKey()`:

```
// Within ArticlesTable::initialize() call
$this->hasMany('Comments')
    ->setForeignKey([
        'article_id',
        'article_hash',
```

(continues on next page)

(continued from previous page)

```
    ])
    ->setBindingKey([
        'whatever_id',
        'whatever_hash',
    ]);
```

Like hasOne associations, `foreignKey` is in the other (Comments) table and `bindingKey` is in the current (Articles) table.

Possible keys for hasMany association arrays include:

- **className**: The class name of the other table. This is the same name used when getting an instance of the table. In the 'Articles hasMany Comments' example, it should be 'Comments'. The default value is the name of the association.
- **foreignKey**: The name of the foreign key column in the other table. The default value is the underscored, singular name of the current model, suffixed with '_id' such as 'article_id' in the above example.
- **bindingKey**: The name of the column in the current table used to match the `foreignKey`. The default value is the primary key of the current table such as 'id' of Articles in the above example.
- **conditions**: an array of find() compatible conditions or SQL strings such as `['Comments.visible' => true]`. It is recommended to use the `finder` option instead.
- **sort**: an array of find() compatible order clauses or SQL strings such as `['Comments.created' => 'ASC']`
- **dependent**: When dependent is set to `true`, recursive model deletion is possible. In this example, Comment records will be deleted when their associated Article record has been deleted.
- **cascadeCallbacks**: When this and **dependent** are `true`, cascaded deletes will load and delete entities so that callbacks are properly triggered. When `false`, `deleteAll()` is used to remove associated data and no callbacks are triggered.
- **propertyName**: The property name that should be filled with data from the associated table into the source table results. By default this is the underscored & plural name of the association so `comments` in our example.
- **strategy**: Defines the query strategy to use. Defaults to 'select'. The other valid value is 'subquery', which replaces the `IN` list with an equivalent subquery.
- **saveStrategy**: Either 'append' or 'replace'. Defaults to 'append'. When 'append' the current records are appended to any records in the database. When 'replace' associated records not in the current set will be removed. If the foreign key is a nullable column or if `dependent` is true records will be orphaned.
- **finder**: The finder method to use when loading associated records. See the *Using Association Finders* section for more information.

Once this association has been defined, find operations on the Articles table can contain the Comment records if they exist:

```
// In a controller or table method.
$query = $articles->find('all')->contain(['Comments'])->all();
foreach ($query as $article) {
    echo $article->comments[0]->text;
}
```

The above would output SQL similar to:

```
SELECT * FROM articles;
SELECT * FROM comments WHERE article_id IN (1, 2, 3, 4, 5);
```

When the subquery strategy is used, SQL similar to the following will be generated:

```
SELECT * FROM articles;
SELECT * FROM comments WHERE article_id IN (SELECT id FROM articles);
```

You may want to cache the counts for your hasMany associations. This is useful when you often need to show the number of associated records, but don't want to load all the records just to count them. For example, the comment count on any given article is often cached to make generating lists of articles more efficient. You can use the *CounterCacheBehavior* to cache counts of associated records.

You should make sure that your database tables do not contain columns that match association property names. If for example you have counter fields that conflict with association properties, you must either rename the association property, or the column name.

BelongsToMany Associations

An example of a BelongsToMany association is "Article BelongsToMany Tags", where the tags from one article are shared with other articles. BelongsToMany is often referred to as "has and belongs to many", and is a classic "many to many" association.

The main difference between hasMany and BelongsToMany is that the link between the models in a BelongsToMany association is not exclusive. For example, we are joining our Articles table with a Tags table. Using 'funny' as a Tag for my Article, doesn't "use up" the tag. I can also use it on the next article I write.

Three database tables are required for a BelongsToMany association. In the example above we would need tables for `articles`, `tags` and `articles_tags`. The `articles_tags` table contains the data that links tags and articles together. The joining table is named after the two tables involved, separated with an underscore by convention. In its simplest form, this table consists of `article_id` and `tag_id`.

belongsToMany requires a separate join table that includes both *model* names.

Relationship	Join Table Fields
Articles belongsToMany Tags	articles_tags.id, articles_tags.tag_id, articles_tags.article_id
Patients belongsToMany Doctors	doctors_patients.id, doctors_patients.doctor_id, doctors_patients.patient_id.

We can define the belongsToMany association in both our models as follows:

```
// In src/Model/Table/ArticlesTable.php
class ArticlesTable extends Table
{
    public function initialize(array $config): void
    {
        $this->belongsToMany('Tags');
    }
}

// In src/Model/Table/TagsTable.php
class TagsTable extends Table
{
    public function initialize(array $config): void
    {
        $this->belongsToMany('Articles');
    }
}
```

We can also define a more specific relationship using configuration:

```
// In src/Model/Table/TagsTable.php
class TagsTable extends Table
{
    public function initialize(array $config): void
    {
        $this->belongsToMany('Articles', [
            'joinTable' => 'articles_tags',
        ]);
    }
}
```

Possible keys for belongsToMany association arrays include:

- **className**: The class name of the other table. This is the same name used when getting an instance of the table. In the 'Articles belongsToMany Tags' example, it should be 'Tags'. The default value is the name of the association.
- **joinTable**: The name of the join table used in this association (if the current table doesn't adhere to the naming convention for belongsToMany join tables). By default this table name will be used to load the Table instance for the join table.
- **foreignKey**: The name of the foreign key that references the current model found on the join table, or list in case of composite foreign keys. This is especially handy if you need to define multiple belongsToMany relationships. The default value for this key is the underscored, singular name of the current model, suffixed with '_id'.
- **bindingKey**: The name of the column in the current table, that will be used for matching the `foreignKey`. Defaults to the primary key.
- **targetForeignKey**: The name of the foreign key that references the target model found on the join model, or list in case of composite foreign keys. The default value for this key is the underscored, singular name of the target model, suffixed with '_id'.
- **conditions**: An array of `find()` compatible conditions. If you have conditions on an associated table, you should use a 'through' model, and define the necessary belongsTo associations on it. It is recommended to use the `finder` option instead.
- **sort**: an array of find() compatible order clauses.
- **dependent**: When the dependent key is set to `false`, and an entity is deleted, the data of the join table will not be deleted.
- **through**: Allows you to provide either the alias of the Table instance you want used on the join table, or the instance itself. This makes customizing the join table keys possible, and allows you to customize the behavior of the pivot table.
- **cascadeCallbacks**: When this is `true`, cascaded deletes will load and delete entities so that callbacks are properly triggered on join table records. When `false`, `deleteAll()` is used to remove associated data and no callbacks are triggered. This defaults to `false` to help reduce overhead.
- **propertyName**: The property name that should be filled with data from the associated table into the source table results. By default this is the underscored & plural name of the association, so `tags` in our example.
- **strategy**: Defines the query strategy to use. Defaults to 'select'. The other valid value is 'subquery', which replaces the `IN` list with an equivalent subquery.
- **saveStrategy**: Either 'append' or 'replace'. Defaults to 'replace'. Indicates the mode to be used for saving associated entities. The former will only create new links between both side of the relation and the latter will do a wipe and replace to create the links between the passed entities when saving.

- **finder**: The finder method to use when loading associated records. See the *Using Association Finders* section for more information.

Once this association has been defined, find operations on the Articles table can contain the Tag records if they exist:

```
// In a controller or table method.
$query = $articles->find('all')->contain(['Tags'])->all();
foreach ($query as $article) {
    echo $article->tags[0]->text;
}
```

The above would output SQL similar to:

```
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)
);
```

When the subquery strategy is used, SQL similar to the following will be generated:

```
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)
);
```

Using the 'through' Option

If you plan on adding extra information to the join/pivot table, or if you need to use join columns outside of the conventions, you will need to define the `through` option. The `through` option provides you full control over how the belongsToMany association will be created.

It is sometimes desirable to store additional data with a many to many association. Consider the following:

```
Student BelongsToMany Course
Course BelongsToMany Student
```

A Student can take many Courses and a Course can be taken by many Students. This is a simple many to many association. The following table would suffice:

```
id | student_id | course_id
```

Now what if we want to store the number of days that were attended by the student on the course and their final grade? The table we'd want would be:

```
id | student_id | course_id | days_attended | grade
```

The way to implement our requirement is to use a **join model**, otherwise known as a **hasMany through** association. That is, the association is a model itself. So, we can create a new model CoursesMemberships. Take a look at the following models:

```
class StudentsTable extends Table
{
    public function initialize(array $config): void
    {
        $this->belongsToMany('Courses', [
            'through' => 'CoursesMemberships',
        ]);
    }
}

class CoursesTable extends Table
{
    public function initialize(array $config): void
    {
        $this->belongsToMany('Students', [
            'through' => 'CoursesMemberships',
        ]);
    }
}

class CoursesMembershipsTable extends Table
{
    public function initialize(array $config): void
    {
        $this->belongsTo('Students');
        $this->belongsTo('Courses');
    }
}
```

The CoursesMemberships join table uniquely identifies a given Student's participation on a Course in addition to extra meta-information.

When using a query object with a BelongsToMany relationship with a `through` model, add contain and matching conditions for the association target table into your query object. The `through` table can then be referenced in other conditions such as a where condition by designating the through table name before the field you are filtering on:

```
$query = $this->find(
        'list',
        valueField: 'studentFirstName', order: 'students.id'
    )
    ->contain(['Courses'])
    ->matching('Courses')
    ->where(['CoursesMemberships.grade' => 'B']);
```

Using Association Finders

By default associations will load records based on the foreign key columns. If you want to define additional conditions for associations, you can use a `finder`. When an association is loaded the ORM will use your *custom finder* to load, update, or delete associated records. Using finders lets you encapsulate your queries and make them more reusable. There are some limitations when using finders to load data in associations that are loaded using joins (belongsTo/hasOne). Only the following aspects of the query will be applied to the root query:

- Where conditions.
- Additional joins.
- Contained associations.

Other aspects of the query, such as selected columns, order, group by, having and other sub-statements, will not be applied to the root query. Associations that are *not* loaded through joins (hasMany/belongsToMany), do not have the above restrictions and can also use result formatters or map/reduce functions.

Association Conventions

By default, associations should be configured and referenced using the CamelCase style. This enables property chains to related tables in the following way:

```
$this->MyTableOne->MyTableTwo->find()->...;
```

Association properties on entities do not use CamelCase conventions though. Instead for a hasOne/belongsTo relation like "User belongsTo Roles", you would get a *role* property instead of *Role* or *Roles*:

```
// A single entity (or null if not available)
$role = $user->role;
```

Whereas for the other direction "Roles hasMany Users" it would be:

```
// Collection of user entities (or null if not available)
$users = $role->users;
```

Loading Associations

Once you've defined your associations you can *eager load associations* when fetching results.

Retrieving Data & Results Sets

class Cake\ORM**Table**

While table objects provide an abstraction around a 'repository' or collection of objects, when you query for individual records you get 'entity' objects. While this section discusses the different ways you can find and load entities, you should read the *Entities* section for more information on entities.

Debugging Queries and ResultSets

Since the ORM now returns Collections and Entities, debugging these objects can be more complicated than in previous CakePHP versions. There are now various ways to inspect the data returned by the ORM.

- `debug($query)` Shows the SQL and bound parameters, does not show results.
- `sql($query)` Shows the final rendered SQL when DebugKit is installed.
- `debug($query->all())` Shows the ResultSet properties (not the results).
- `debug($query->toList())` Show results in an array.
- `debug(iterator_to_array($query))` Shows query results in an array format.
- `debug(json_encode($query, JSON_PRETTY_PRINT))` More human readable results.
- `debug($query->first())` Show the properties of a single entity.
- `debug((string)$query->first())` Show the properties of a single entity as JSON.

Getting a Single Entity by Primary Key

`Cake\ORM\Table::get`(*$id*, *$options = []*)

It is often convenient to load a single entity from the database when editing or viewing entities and their related data. You can do this by using `get()`:

```
// In a controller or table method.

// Get a single article
$article = $articles->get($id);

// Get a single article, and related comments
$article = $articles->get($id, contain: ['Comments']);
```

If the get operation does not find any results a `Cake\Datasource\Exception\RecordNotFoundException` will be raised. You can either catch this exception yourself, or allow CakePHP to convert it into a 404 error.

Like `find()`, `get()` also has caching integrated. You can use the `cache` option when calling `get()` to perform read-through caching:

```
// In a controller or table method.

// Use any cache config or CacheEngine instance & a generated key
$article = $articles->get($id, cache: 'custom');

// Use any cache config or CacheEngine instance & specific key
$article = $articles->get($id, cache: 'custom', key: 'mykey');

// Explicitly disable caching
$article = $articles->get($id, cache: false);
```

Optionally you can `get()` an entity using *Custom Finder Methods*. For example you may want to get all translations for an entity. You can achieve that by using the `finder` option:

```
$article = $articles->get($id, 'translations');
```

The list of options supported by get() are:

- `cache` cache config.
- `key` cache key.
- `finder` custom finder function.
- `conditions` provide conditions for the WHERE clause of your query.
- `limit` Set the number of rows you want.
- `offset` Set the page offset you want. You can also use `page` to make the calculation simpler.
- `contain` define the associations to eager load.
- `fields` limit the fields loaded into the entity. Only loading some fields can cause entities to behave incorrectly.
- `group` add a GROUP BY clause to your query. This is useful when using aggregating functions.
- `having` add a HAVING clause to your query.
- `join` define additional custom joins.

Using Finders to Load Data

`Cake\ORM\Table::`**`find`**(*$type*, *mixed ...$args*)

Before you can work with entities, you'll need to load them. The easiest way to do this is using the `find()` method. The find method provides a short and extensible way to find the data you are interested in:

```
// In a controller or table method.

// Find all the articles
$query = $articles->find('all');
```

The return value of any `find()` method is always a `Cake\ORM\Query\SelectQuery` object. The SelectQuery class allows you to further refine a query after creating it. SelectQuery objects are evaluated lazily, and do not execute until you start fetching rows, convert it to an array, or when the `all()` method is called:

```
// In a controller or table method.

// Find all the articles.
// At this point the query has not run.
$query = $articles->find('all');

// Calling all() will execute the query
// and return the result set.
$results = $query->all();

// Once we have a result set we can get all the rows
$data = $results->toList();

// Converting the query to a key-value array will also execute it.
$data = $query->toArray();
```

Note: Once you've started a query you can use the *Query Builder* interface to build more complex queries, adding additional conditions, limits, or include associations using the fluent interface.

```
// In a controller or table method.
$query = $articles->find('all')
    ->where(['Articles.created >' => new DateTime('-10 days')])
    ->contain(['Comments', 'Authors'])
    ->limit(10);
```

You can also provide many commonly used options to `find()`:

```
// In a controller or table method.
$query = $articles->find('all',
    conditions: ['Articles.created >' => new DateTime('-10 days')],
    contain: ['Authors', 'Comments'],
    limit: 10
);
```

The list of named arguments supported by find() by default are:

- `conditions` provide conditions for the WHERE clause of your query.
- `limit` Set the number of rows you want.
- `offset` Set the page offset you want. You can also use `page` to make the calculation simpler.
- `contain` define the associations to eager load.
- `fields` limit the fields loaded into the entity. Only loading some fields can cause entities to behave incorrectly.
- `group` add a GROUP BY clause to your query. This is useful when using aggregating functions.
- `having` add a HAVING clause to your query.
- `join` define additional custom joins.
- `order` order the result set.

Any options that are not in this list will be passed to beforeFind listeners where they can be used to modify the query object. You can use the `getOptions()` method on a query object to retrieve the options used. While you can pass query objects to your controllers, we recommend that you package your queries up as *Custom Finder Methods* instead. Using custom finder methods will let you re-use your queries and make testing easier.

By default queries and result sets will return *Entities* objects. You can retrieve basic arrays by disabling hydration:

```
$query->disableHydration();

// $data is ResultSet that contains array data.
$data = $query->all();
```

Getting the First Result

The `first()` method allows you to fetch only the first row from a query. If the query has not been executed, a `LIMIT 1` clause will be applied:

```
// In a controller or table method.
$query = $articles->find('all', order: ['Articles.created' => 'DESC']);
$row = $query->first();
```

This approach replaces `find('first')` in previous versions of CakePHP. You may also want to use the `get()` method if you are loading entities by primary key.

Note: The `first()` method will return `null` if no results are found.

Getting a Count of Results

Once you have created a query object, you can use the `count()` method to get a result count of that query:

```
// In a controller or table method.
$query = $articles->find('all', conditions: ['Articles.title LIKE' => '%Ovens%']);
$number = $query->count();
```

See *Returning the Total Count of Records* for additional usage of the `count()` method.

Finding Key/Value Pairs

It is often useful to generate an associative array of data from your application's data. For example, this is very useful when creating <select> elements. CakePHP provides a simple to use method for generating 'lists' of data:

```
// In a controller or table method.
$query = $articles->find('list');
$data = $query->toArray();

// Data now looks like
$data = [
    1 => 'First post',
    2 => 'Second article I wrote',
];
```

With no additional options the keys of `$data` will be the primary key of your table, while the values will be the 'displayField' of the table. The default 'displayField' of the table is `title` or `name`. While, you can use the `setDisplayField()` method on a table object to configure the display field of a table:

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

When calling `list` you can configure the fields used for the key and value with the `keyField` and `valueField` options respectively:

```
// In a controller or table method.
$query = $articles->find('list', keyField: 'slug', valueField: 'label');
$data = $query->toArray();

// Data now looks like
$data = [
    'first-post' => 'First post',
    'second-article-i-wrote' => 'Second article I wrote',
];
```

Results can be grouped into nested sets. This is useful when you want bucketed sets, or want to build <optgroup> elements with FormHelper:

```
// In a controller or table method.
$query = $articles->find('list', keyField: 'slug', valueField: 'label', groupField:
↪'author_id');
$data = $query->toArray();

// Data now looks like
$data = [
    1 => [
        'first-post' => 'First post',
        'second-article-i-wrote' => 'Second article I wrote',
    ],
    2 => [
        // More data.
    ]
];
```

You can also create list data from associations that can be reached with joins:

```
$query = $articles->find('list', keyField: 'id', valueField: 'author.name')
    ->contain(['Authors']);
```

The keyField, valueField, and groupField expression will operate on entity attribute paths not the database columns. This means that you can use virtual fields in the results of find(list).

Customize Key-Value Output

Lastly it is possible to use closures to access entity accessor methods in your list finds.

```
// In your Authors Entity create a virtual field to be used as the displayField:
protected function _getLabel()
{
    return $this->_fields['first_name'] . ' ' . $this->_fields['last_name']
      . ' / ' . __('User ID %s', $this->_fields['user_id']);
}
```

This example shows using the _getLabel() accessor method from the Author entity.

```
// In your finders/controller:
$query = $articles->find('list',
        keyField: 'id',
        valueField: function ($article) {
            return $article->author->get('label');
        }
    )
    ->contain('Authors');
```

You can also fetch the label in the list directly using.

```
// In AuthorsTable::initialize():
$this->setDisplayField('label'); // Will utilize Author::_getLabel()
```

(continues on next page)

(continued from previous page)

```
// In your finders/controller:
$query = $authors->find('list'); // Will utilize AuthorsTable::getDisplayField()
```

Finding Threaded Data

The `find('threaded')` finder returns nested entities that are threaded together through a key field. By default this field is `parent_id`. This finder allows you to access data stored in an 'adjacency list' style table. All entities matching a given `parent_id` are placed under the `children` attribute:

```
// In a controller or table method.
$query = $comments->find('threaded');

// Expanded default values
$query = $comments->find('threaded',
    keyField: $comments->primaryKey(),
    parentField: 'parent_id'
);
$results = $query->toArray();

echo count($results[0]->children);
echo $results[0]->children[0]->comment;
```

The `parentField` and `keyField` keys can be used to define the fields that threading will occur on.

Tip: If you need to manage more advanced trees of data, consider using *Tree* instead.

Custom Finder Methods

The examples above show how to use the built-in `all` and `list` finders. However, it is possible and recommended that you implement your own finder methods. Finder methods are the ideal way to package up commonly used queries, allowing you to abstract query details into a simple to use method. Finder methods are defined by creating methods following the convention of `findFoo` where `Foo` is the name of the finder you want to create. For example if we wanted to add a finder to our articles table for finding articles written by a given user, we would do the following:

```
use App\Model\Entity\User;
use Cake\ORM\Query\SelectQuery;
use Cake\ORM\Table;

class ArticlesTable extends Table
{
    public function findOwnedBy(SelectQuery $query, User $user)
    {
        return $query->where(['author_id' => $user->id]);
    }
}

$query = $articles->find('ownedBy', user: $userEntity);
```

Finder methods can modify the query as required, or use the `$options` to customize the finder operation with relevant application logic. You can also 'stack' finders, allowing you to express complex queries effortlessly. Assuming you

have both the 'published' and 'recent' finders, you could do the following:

```
$query = $articles->find('published')->find('recent');
```

While all the examples so far have shown finder methods on table classes, finder methods can also be defined on *Behaviors*.

If you need to modify the results after they have been fetched you should use a *Modifying Results with Map/Reduce* function to modify the results. The map reduce features replace the 'afterFind' callback found in previous versions of CakePHP.

Dynamic Finders

CakePHP's ORM provides dynamically constructed finder methods which allow you to express simple queries with no additional code. For example if you wanted to find a user by username you could do:

```
// In a controller
// The following two calls are equal.
$query = $this->Users->findByUsername('joebob');
$query = $this->Users->findAllByUsername('joebob');
```

When using dynamic finders you can constrain on multiple fields:

```
$query = $users->findAllByUsernameAndApproved('joebob', 1);
```

You can also create `OR` conditions:

```
$query = $users->findAllByUsernameOrEmail('joebob', 'joe@example.com');
```

While you can use either `OR` or `AND` conditions, you cannot combine the two in a single dynamic finder. Other query options like `contain` are also not supported with dynamic finders. You should use *Custom Finder Methods* to encapsulate more complex queries. Lastly, you can also combine dynamic finders with custom finders:

```
$query = $users->findTrollsByUsername('bro');
```

The above would translate into the following:

```
$users->find('trolls', conditions: ['username' => 'bro']);
```

Once you have a query object from a dynamic finder, you'll need to call `first()` if you want the first result.

Note: While dynamic finders make it simple to express queries, they add a small amount of overhead. You cannot call `findBy` methods from a query object. When using a finder chain the dynamic finder must be called first.

Retrieving Associated Data

When you want to grab associated data, or filter based on associated data, there are two ways:

- use CakePHP ORM query functions like `contain()` and `matching()`
- use join functions like `innerJoin()`, `leftJoin()`, and `rightJoin()`

You should use `contain()` when you want to load the primary model, and its associated data. While `contain()` will let you apply additional conditions to the loaded associations, you cannot constrain the primary model based on the associations. For more details on the `contain()`, look at *Eager Loading Associations Via Contain*.

You should use `matching()` when you want to restrict the primary model based on associations. For example, you want to load all the articles that have a specific tag on them. For more details on the `matching()`, look at *Filtering by Associated Data Via Matching And Joins*.

If you prefer to use join functions, you can look at *Adding Joins* for more information.

Eager Loading Associations Via Contain

By default CakePHP does not load **any** associated data when using `find()`. You need to 'contain' or eager-load each association you want loaded in your results.

Eager loading helps avoid many of the potential performance problems surrounding lazy-loading in an ORM. The queries generated by eager loading can better leverage joins, allowing more efficient queries to be made. In CakePHP you state which associations should be eager loaded using the 'contain' method:

```
// In a controller or table method.

// As an option to find()
$query = $articles->find('all', contain: ['Authors', 'Comments']);

// As a method on the query object
$query = $articles->find('all');
$query->contain(['Authors', 'Comments']);
```

The above will load the related author and comments for each article in the result set. You can load nested associations using nested arrays to define the associations to be loaded:

```
$query = $articles->find()->contain([
    'Authors' => ['Addresses'], 'Comments' => ['Authors']
]);
```

Alternatively, you can express nested associations using the dot notation:

```
$query = $articles->find()->contain([
    'Authors.Addresses',
    'Comments.Authors'
]);
```

You can eager load associations as deep as you like:

```
$query = $products->find()->contain([
    'Shops.Cities.Countries',
    'Shops.Managers'
]);
```

Which is equivalent to calling:

```
$query = $products->find()->contain([
    'Shops' => ['Cities.Countries', 'Managers']
]);
```

You can select fields from all associations with multiple `contain()` statements:

```
$query = $this->find()->select([
    'Realestates.id',
    'Realestates.title',
    'Realestates.description'
])
->contain([
    'RealestateAttributes' => [
        'Attributes' => [
            'fields' => [
                // Aliased fields in contain() must include
                // the model prefix to be mapped correctly.
                'Attributes__name' => 'attr_name',
            ],
        ],
    ],
])
->contain([
    'RealestateAttributes' => [
        'fields' => [
            'RealestateAttributes.realestate_id',
            'RealestateAttributes.value',
        ],
    ],
])
->where($condition);
```

If you need to reset the containments on a query you can set the second argument to `true`:

```
$query = $articles->find();
$query->contain(['Authors', 'Comments'], true);
```

Note: Association names in `contain()` calls should use the same association casing as in your association definitions, not the property name used to hold the association record(s). For example, if you have declared an association as `belongsTo('Users')` then you must use `contain('Users')` and not `contain('users')` or `contain('user')`.

Passing Conditions to Contain

When using `contain()` you are able to restrict the data returned by the associations and filter them by conditions. To specify conditions, pass an anonymous function that receives as the first argument a query object, `\Cake\ORM\Query\SelectQuery`:

```
// In a controller or table method.
$query = $articles->find()->contain('Comments', function (SelectQuery $q) {
    return $q
        ->select(['body', 'author_id'])
        ->where(['Comments.approved' => true]);
});
```

This also works for pagination at the Controller level:

```
$this->paginate['contain'] = [
    'Comments' => function (SelectQuery $query) {
        return $query->select(['body', 'author_id'])
        ->where(['Comments.approved' => true]);
    }
];
```

Warning: If the results are missing association entities, make sure the foreign key columns are selected in the query. Without the foreign keys, the ORM cannot find matching rows.

It is also possible to restrict deeply-nested associations using the dot notation:

```
$query = $articles->find()->contain([
    'Comments',
    'Authors.Profiles' => function (SelectQuery $q) {
        return $q->where(['Profiles.is_published' => true]);
    }
]);
```

In the above example, you'll still get authors even if they don't have a published profile. To only get authors with a published profile use *matching()*. If you have defined custom finders in your associations, you can use them inside `contain()`:

```
// Bring all articles, but only bring the comments that are approved and
// popular.
$query = $articles->find()->contain('Comments', function (SelectQuery $q) {
    return $q->find('approved')->find('popular');
});
```

Note: With `BelongsTo` and `HasOne` associations only `select` and `where` clauses are valid in the `contain()` query. With `HasMany` and `BelongsToMany` all clauses such as `order()` are valid.

You can control more than just the query clauses used by `contain()`. If you pass an array with the association, you can override the `foreignKey`, `joinType` and `strategy`. See *Associations - Linking Tables Together* for details on the default value and options for each association type.

You can pass `false` as the new `foreignKey` to disable foreign key constraints entirely. Use the `queryBuilder` option to customize the query when using an array:

```
$query = $articles->find()->contain([
    'Authors' => [
        'foreignKey' => false,
        'queryBuilder' => function (SelectQuery $q) {
            return $q->where(...); // Full conditions for filtering
        }
    ]
]);
```

If you have limited the fields you are loading with `select()` but also want to load fields off of contained associations, you can pass the association object to `select()`:

```
// Select id & title from articles, but all fields off of Users.
$query = $articles->find()
    ->select(['id', 'title'])
    ->select($articles->Users)
    ->contain(['Users']);
```

Alternatively, you can use `enableAutoFields()` in an anonymous function:

```
// Select id & title from articles, but all fields off of Users.
$query = $articles->find()
    ->select(['id', 'title'])
    ->contain(['Users' => function(SelectQuery $q) {
        return $q->enableAutoFields();
    }]);
```

Sorting Contained Associations

When loading HasMany and BelongsToMany associations, you can use the `sort` option to sort the data in those associations:

```
$query->contain([
    'Comments' => [
        'sort' => ['Comments.created' => 'DESC']
    ]
]);
```

Filtering by Associated Data Via Matching And Joins

A fairly common query case with associations is finding records 'matching' specific associated data. For example if you have 'Articles belongsToMany Tags' you will probably want to find Articles that have the CakePHP tag. This is extremely simple to do with the ORM in CakePHP:

```
// In a controller or table method.

$query = $articles->find();
$query->matching('Tags', function ($q) {
```

(continues on next page)

(continued from previous page)

```
    return $q->where(['Tags.name' => 'CakePHP']);
});
```

You can apply this strategy to HasMany associations as well. For example if 'Authors HasMany Articles', you could find all the authors with recently published articles using the following:

```
$query = $authors->find();
$query->matching('Articles', function ($q) {
    return $q->where(['Articles.created >=' => new DateTime('-10 days')]);
});
```

Filtering by deep associations uses the same predictable syntax from `contain()`:

```
// In a controller or table method.
$query = $products->find()->matching(
    'Shops.Cities.Countries', function ($q) {
        return $q->where(['Countries.name' => 'Japan']);
    }
);

// Bring unique articles that were commented by `markstory` using passed variable
// Dotted matching paths should be used over nested matching() calls
$username = 'markstory';
$query = $articles->find()->matching('Comments.Users', function ($q) use ($username) {
    return $q->where(['username' => $username]);
});
```

Note: As this function will create an `INNER JOIN`, you might want to consider calling `distinct` on the find query as you might get duplicate rows if your conditions don't exclude them already. This might be the case, for example, when the same users comments more than once on a single article.

The data from the association that is 'matched' will be available on the `_matchingData` property of entities. If both match and contain the same association, you can expect to get both the `_matchingData` and standard association properties in your results.

Using innerJoinWith

Sometimes you need to match specific associated data but without actually loading the matching records like `matching()`. You can create just the `INNER JOIN` that `matching()` uses with `innerJoinWith()`:

```
$query = $articles->find();
$query->innerJoinWith('Tags', function ($q) {
    return $q->where(['Tags.name' => 'CakePHP']);
});
```

`innerJoinWith()` allows you to the same parameters and dot notation:

```
$query = $products->find()->innerJoinWith(
    'Shops.Cities.Countries', function ($q) {
        return $q->where(['Countries.name' => 'Japan']);
```

(continues on next page)

(continued from previous page)

```
    }
);
```

You can combine `innerJoinWith()` and `contain()` with the same association when you want to match specific records and load the associated data together. The example below matches Articles that have specific Tags and loads the same Tags:

```
$filter = ['Tags.name' => 'CakePHP'];
$query = $articles->find()
    ->distinct($articles->getPrimaryKey())
    ->contain('Tags', function (SelectQuery $q) use ($filter) {
        return $q->where($filter);
    })
    ->innerJoinWith('Tags', function (SelectQuery $q) use ($filter) {
        return $q->where($filter);
    });
```

Note: If you use `innerJoinWith()` and want to `select()` fields from that association, you need to use an alias for the field:

```
$query
    ->select(['country_name' => 'Countries.name'])
    ->innerJoinWith('Countries');
```

If you don't use an alias, you will see the data in `_matchingData` as described by `matching()` above. This is an edge case from `matching()` not knowing you manually selected the field.

Warning: You should not combine `innerJoinWith()` and `matching()` with the same association. This will produce multiple `INNER JOIN` statements and might not create the query you expected.

Using notMatching

The opposite of `matching()` is `notMatching()`. This function will change the query so that it filters results that have no relation to the specified association:

```
// In a controller or table method.

$query = $articlesTable
    ->find()
    ->notMatching('Tags', function ($q) {
        return $q->where(['Tags.name' => 'boring']);
    });
```

The above example will find all articles that were not tagged with the word `boring`. You can apply this method to HasMany associations as well. You could, for example, find all the authors with no published articles in the last 10 days:

```
$query = $authorsTable
    ->find()
    ->notMatching('Articles', function ($q) {
        return $q->where(['Articles.created >=' => new \DateTime('-10 days')]);
    });
```

It is also possible to use this method for filtering out records not matching deep associations. For example, you could find articles that have not been commented on by a certain user:

```
$query = $articlesTable
    ->find()
    ->notMatching('Comments.Users', function ($q) {
        return $q->where(['username' => 'jose']);
    });
```

Since articles with no comments at all also satisfy the condition above, you may want to combine `matching()` and `notMatching()` in the same query. The following example will find articles having at least one comment, but not commented by a certain user:

```
$query = $articlesTable
    ->find()
    ->notMatching('Comments.Users', function ($q) {
        return $q->where(['username' => 'jose']);
    })
    ->matching('Comments');
```

Note: As `notMatching()` will create a `LEFT JOIN`, you might want to consider calling `distinct` on the find query as you can get duplicate rows otherwise.

Keep in mind that contrary to the `matching()` function, `notMatching()` will not add any data to the `_matchingData` property in the results.

Using leftJoinWith

On certain occasions you may want to calculate a result based on an association, without having to load all the records for it. For example, if you wanted to load the total number of comments an article has along with all the article data, you can use the `leftJoinWith()` function:

```
$query = $articlesTable->find();
$query->select(['total_comments' => $query->func()->count('Comments.id')])
    ->leftJoinWith('Comments')
    ->groupBy(['Articles.id'])
    ->enableAutoFields(true);
```

The results for the above query will contain the article data and the `total_comments` property for each of them.

`leftJoinWith()` can also be used with deeply nested associations. This is useful, for example, for bringing the count of articles tagged with a certain word, per author:

```
$query = $authorsTable
    ->find()
```

(continues on next page)

(continued from previous page)

```
    ->select(['total_articles' => $query->func()->count('Articles.id')])
    ->leftJoinWith('Articles.Tags', function ($q) {
        return $q->where(['Tags.name' => 'awesome']);
    })
    ->groupBy(['Authors.id'])
    ->enableAutoFields(true);
```

This function will not load any columns from the specified associations into the result set.

Changing Fetching Strategies

As mentioned in earlier, you can customize the `strategy` used by an association in a `contain()`.

If you look at `BelongsTo` and `HasOne` *association* options, the default 'join' strategy and 'INNER' `joinType` can be changed to 'select':

```
$query = $articles->find()->contain([
    'Comments' => [
        'strategy' => 'select',
    ]
]);
```

This can be useful when you need to add conditions that don't work well in a join. This also makes it possible to query tables that are not allowed in joins such as separate databases.

Usually, you set the strategy for an association when defining it in `Table::initialize()`, but you can permanently change the strategy manually:

```
$articles->Comments->setStrategy('select');
```

Fetching With The Subquery Strategy

As your tables grow in size, fetching associations from them can become slower, especially if you are querying big batches at once. A good way of optimizing association loading for `hasMany` and `belongsToMany` associations is by using the `subquery` strategy:

```
$query = $articles->find()->contain([
    'Comments' => [
            'strategy' => 'subquery',
            'queryBuilder' => function ($q) {
                return $q->where(['Comments.approved' => true]);
            }
    ]
]);
```

The result will remain the same as with using the default strategy, but this can greatly improve the query and fetching time in some databases, in particular it will allow to fetch big chunks of data at the same time in databases that limit the amount of bound parameters per query, such as **Microsoft SQL Server**.

Lazy Loading Associations

While CakePHP uses eager loading to fetch your associations, there may be cases where you need to lazy-load associations. You should refer to the *Lazy Loading Associations* and *Loading Additional Associations* sections for more information.

Working with Result Sets

Once a query is executed with `all()`, you will get an instance of `Cake\ORM\ResultSet`. This object offers powerful ways to manipulate the resulting data from your queries. ResultSets are a *Collection* and you can use any collection method on ResultSet objects.

Result set objects will lazily load rows from the underlying prepared statement. By default results will be buffered in memory allowing you to iterate a result set multiple times, or cache and iterate the results.

Result sets allow you to cache/serialize or JSON encode results for API results:

```
// In a controller or table method.
$results = $query->all();

// Serialized
$serialized = serialize($results);

// Json
$json = json_encode($results);
```

Both serializing and JSON encoding result sets work as you would expect. The serialized data can be unserialized into a working result set. Converting to JSON respects hidden & virtual field settings on all entity objects within a result set.

Result sets are a 'Collection' object and support the same methods that *collection objects* do. For example, you can extract a list of unique tags on a collection of articles by running:

```
// In a controller or table method.
$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, []);
```

Some other examples of the collection methods being used with result sets are:

```
// Filter the rows by a calculated property
$filtered = $results->filter(function ($row) {
    return $row->is_recent;
});

// Create an associative array from result properties
```

(continues on next page)

(continued from previous page)

```
$results = $articles->find()->contain(['Authors'])->all();

$authorList = $results->combine('id', 'author.name');
```

The *Collections* chapter has more detail on what can be done with result sets using the collections features. The *Adding Calculated Fields* section show how you can add calculated fields, or replace the result set.

Getting the First & Last Record From a ResultSet

You can use the `first()` and `last()` methods to get the respective records from a result set:

```
$result = $articles->find('all')->all();

// Get the first and/or last result.
$row = $result->first();
$row = $result->last();
```

Getting an Arbitrary Index From a ResultSet

You can use `skip()` and `first()` to get an arbitrary record from a ResultSet:

```
$result = $articles->find('all')->all();

// Get the 5th record
$row = $result->skip(4)->first();
```

Checking if a ResultSet is Empty

You can use the `isEmpty()` method on a ResultSet object to see if it has any rows in it.:

```
// Check results
$results = $query->all();
$results->isEmpty();
```

Loading Additional Associations

Once you've created a result set, you may need to load additional associations. This is the perfect time to lazily eager load data. You can load additional associations using `loadInto()`:

```
$articles = $this->Articles->find()->all();
$withMore = $this->Articles->loadInto($articles, ['Comments', 'Users']);
```

It is possible to restrict the data returned by the associations and filter them by conditions. To specify conditions, pass an anonymous function that receives as the first argument a query object, `\Cake\ORM\Query`:

```
$user = $this->Users->get($id);
$withMore = $this->Users->loadInto($user, ['Posts' => function (Query $query) {
```

(continues on next page)

(continued from previous page)

```
    return $query->where(['Posts.status' => 'published']);
}]);
```

You can eager load additional data into a single entity, or a collection of entities.

Modifying Results with Map/Reduce

More often than not, find operations require post-processing the data that is found in the database. While entities' getter methods can take care of most of the virtual field generation or special data formatting, sometimes you need to change the data structure in a more fundamental way.

For those cases, the `SelectQuery` object offers the `mapReduce()` method, which is a way of processing results once they are fetched from the database.

A common example of changing the data structure is grouping results together based on certain conditions. For this task we can use the `mapReduce()` function. We need two callable functions the `$mapper` and the `$reducer`. The `$mapper` callable receives the current result from the database as first argument, the iteration key as second argument and finally it receives an instance of the `MapReduce` routine it is running:

```
$mapper = function ($article, $key, $mapReduce) {
    $status = 'published';
    if ($article->isDraft() || $article->isInReview()) {
        $status = 'unpublished';
    }
    $mapReduce->emitIntermediate($article, $status);
};
```

In the above example `$mapper` is calculating the status of an article, either published or unpublished, then it calls `emitIntermediate()` on the `MapReduce` instance. This method stores the article in the list of articles labelled as either published or unpublished.

The next step in the map-reduce process is to consolidate the final results. For each status created in the mapper, the `$reducer` function will be called so you can do any extra processing. This function will receive the list of articles in a particular "bucket" as the first parameter, the name of the "bucket" it needs to process as the second parameter, and again, as in the `mapper()` function, the instance of the `MapReduce` routine as the third parameter. In our example, we did not have to do any extra processing, so we just `emit()` the final results:

```
$reducer = function ($articles, $status, $mapReduce) {
    $mapReduce->emit($articles, $status);
};
```

Finally, we can put these two functions together to do the grouping:

```
$articlesByStatus = $articles->find()
    ->where(['author_id' => 1])
    ->mapReduce($mapper, $reducer)
    ->all();

foreach ($articlesByStatus as $status => $articles) {
    echo sprintf("There are %d %s articles", count($articles), $status);
}
```

The above will output the following lines:

```
There are 4 published articles
There are 5 unpublished articles
```

Of course, this is a simplistic example that could actually be solved in another way without the help of a map-reduce process. Now, let's take a look at another example in which the reducer function will be needed to do something more than just emitting the results.

Calculating the most commonly mentioned words, where the articles contain information about CakePHP, as usual we need a mapper function:

```
$mapper = function ($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);
    }
};
```

It first checks for whether the "cakephp" word is in the article's body, and then breaks the body into individual words. Each word will create its own `bucket` where each article id will be stored. Now let's reduce our results to only extract the count:

```
$reducer = function ($occurrences, $word, $mapReduce) {
    $mapReduce->emit(count($occurrences), $word);
}
```

Finally, we put everything together:

```
$wordCount = $articles->find()
    ->where(['published' => true])
    ->andWhere(['published_date >=' => new DateTime('2014-01-01')])
    ->disableHydration()
    ->mapReduce($mapper, $reducer)
    ->all()
    ->toArray();
```

This could return a very large array if we don't clean stop words, but it could look something like this:

```
[
    'cakephp' => 100,
    'awesome' => 39,
    'impressive' => 57,
    'outstanding' => 10,
    'mind-blowing' => 83
]
```

One last example and you will be a map-reduce expert. Imagine you have a `friends` table and you want to find "fake friends" in our database, or better said, people who do not follow each other. Let's start with our `mapper()` function:

```
$mapper = function ($rel, $key, $mr) {
    $mr->emitIntermediate($rel['target_user_id'], $rel['source_user_id']);
```

(continues on next page)

(continued from previous page)

```
    $mr->emitIntermediate(-$rel['source_user_id'], $rel['target_user_id']);
};
```

The intermediate array will be like the following:

```
[
    1 => [2, 3, 4, 5, -3, -5],
    2 => [-1],
    3 => [-1, 1, 6],
    4 => [-1],
    5 => [-1, 1],
    6 => [-3],
    ...
]
```

Positive numbers mean that a user, indicated with the first-level key, is following them, and negative numbers mean that the user is followed by them.

Now it's time to reduce it. For each call to the reducer, it will receive a list of followers per user:

```
$reducer = function ($friends, $user, $mr) {
    $fakeFriends = [];

    foreach ($friends as $friend) {
        if ($friend > 0 && !in_array(-$friend, $friends)) {
            $fakeFriends[] = $friend;
        }
    }

    if ($fakeFriends) {
        $mr->emit($fakeFriends, $user);
    }
};
```

And we supply our functions to a query:

```
$fakeFriends = $friends->find()
    ->disableHydration()
    ->mapReduce($mapper, $reducer)
    ->all()
    ->toArray();
```

This would return an array similar to this:

```
[
    1 => [2, 4],
    3 => [6]
    ...
]
```

The resulting array means, for example, that user with id 1 follows users 2 and 4, but those do not follow 1 back.

Stacking Multiple Operations

Using `mapReduce` in a query will not execute it immediately. The operation will be registered to be run as soon as the first result is attempted to be fetched. This allows you to keep chaining additional methods and filters to the query even after adding a map-reduce routine:

```
$query = $articles->find()
    ->where(['published' => true])
    ->mapReduce($mapper, $reducer);

// At a later point in your app:
$query->where(['created >=' => new DateTime('1 day ago')]);
```

This is particularly useful for building custom finder methods as described in the *Custom Finder Methods* section:

```
public function findPublished(SelectQuery $query)
{
    return $query->where(['published' => true]);
}

public function findRecent(SelectQuery $query)
{
    return $query->where(['created >=' => new DateTime('1 day ago')]);
}

public function findCommonWords(SelectQuery $query)
{
    // 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');
```

Moreover, it is also possible to stack more than one `mapReduce` operation for a single query. For example, if we wanted to have the most commonly used words for articles, but then filter it to only return words that were mentioned more than 20 times across all articles:

```
$mapper = function ($count, $word, $mr) {
    if ($count > 20) {
        $mr->emit($count, $word);
    }
};

$articles->find('commonWords')->mapReduce($mapper)->all();
```

Removing All Stacked Map-reduce Operations

Under some circumstances you may want to modify a `SelectQuery` object so that no `mapReduce` operations are executed at all. This can be done by calling the method with both parameters as null and the third parameter (overwrite) as `true`:

```
$query->mapReduce(null, null, true);
```

Validating Data

Before you *save your data* you will probably want to ensure the data is correct and consistent. In CakePHP we have two stages of validation:

1. Before request data is converted into entities, validation rules around data types and formatting can be applied.
2. Before data is saved, domain or application rules can be applied. These rules help ensure that your application's data remains consistent.

Validating Data Before Building Entities

When marshalling data into entities, you can validate data. Validating data allows you to check the type, shape and size of data. By default request data will be validated before it is converted into entities. If any validation rules fail, the returned entity will contain errors. The fields with errors will not be present in the returned entity:

```
$article = $articles->newEntity($this->request->getData());
if ($article->getErrors()) {
    // Entity failed validation.
}
```

When building an entity with validation enabled the following occurs:

1. The validator object is created.
2. The `table` and `default` validation provider are attached.
3. The named validation method is invoked. For example `validationDefault`.
4. The `Model.buildValidator` event will be triggered.
5. Request data will be validated.
6. Request data will be type-cast into types that match the column types.
7. Errors will be set into the entity.
8. Valid data will be set into the entity, while fields that failed validation will be excluded.

If you'd like to disable validation when converting request data, set the `validate` option to false:

```
$article = $articles->newEntity(
    $this->request->getData(),
    ['validate' => false]
);
```

The same can be said about the `patchEntity()` method:

```
$article = $articles->patchEntity($article, $newData, [
    'validate' => false
]);
```

Creating A Default Validation Set

Validation rules are defined in the Table classes for convenience. This defines what data should be validated in conjunction with where it will be saved.

To create a default validation object in your table, create the `validationDefault()` function:

```
use Cake\ORM\Table;
use Cake\Validation\Validator;

class ArticlesTable extends Table
{
    public function validationDefault(Validator $validator): Validator
    {
        $validator
            ->requirePresence('title', 'create')
            ->notEmptyString('title');

        $validator
            ->allowEmptyString('link')
            ->add('link', 'valid-url', ['rule' => 'url']);

        ...

        return $validator;
    }
}
```

The available validation methods and rules come from the `Validator` class and are documented in the *Creating Validators* section.

Note: Validation objects are intended primarily for validating user input, i.e. forms and any other posted request data.

Using A Different Validation Set

In addition to disabling validation you can choose which validation rule set you want applied:

```
$article = $articles->newEntity(
    $this->request->getData(),
    ['validate' => 'update']
);
```

The above would call the `validationUpdate()` method on the table instance to build the required rules. By default the `validationDefault()` method will be used. An example validator for our articles table would be:

```
class ArticlesTable extends Table
{
    public function validationUpdate($validator)
    {
        $validator
            ->notEmptyString('title', __('You need to provide a title'))
            ->notEmptyString('body', __('A body is required'));
        return $validator;
    }
}
```

You can have as many validation sets as necessary. See the *validation chapter* for more information on building validation rule-sets.

Using A Different Validation Set For Associations

Validation sets can also be defined per association. When using the `newEntity()` or `patchEntity()` methods, you can pass extra options to each of the associations to be converted:

```
$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'],
     ],
 ]);
```

Combining Validators

Because of how validator objects are built, you can decompose their construction process into multiple reusable steps:

```
// UsersTable.php

public function validationDefault(Validator $validator): Validator
{
    $validator->notEmptyString('username');
    $validator->notEmptyString('password');
    $validator->add('email', 'valid-email', ['rule' => 'email']);
```

(continues on next page)

(continued from previous page)

```
    ...

    return $validator;
}

public function validationHardened(Validator $validator): Validator
{
    $validator = $this->validationDefault($validator);

    $validator->add('password', 'length', ['rule' => ['lengthBetween', 8, 100]]);
    return $validator;
}
```

Given the above setup, when using the `hardened` validation set, it will also contain the validation rules declared in the `default` set.

Validation Providers

Validation rules can use functions defined on any known providers. By default CakePHP sets up a few providers:

1. Methods on the table class or its behaviors are available on the `table` provider.
2. The core `Validation` class is setup as the `default` provider.

When a validation rule is created you can name the provider of that rule. For example, if your table has an `isValidRole` method you can use it as a validation rule:

```
use Cake\ORM\Table;
use Cake\Validation\Validator;

class UsersTable extends Table
{
    public function validationDefault(Validator $validator): Validator
    {
        $validator
            ->add('role', 'validRole', [
                'rule' => 'isValidRole',
                'message' => __('You need to provide a valid role'),
                'provider' => 'table',
            ]);
        return $validator;
    }

    public function isValidRole($value, array $context): bool
    {
        return in_array($value, ['admin', 'editor', 'author'], true);
    }

}
```

You can also use closures for validation rules:

```
$validator->add('name', 'myRule', [
    'rule' => function ($value, array $context) {
        if ($value > 1) {
            return true;
        }
        return 'Not a good value.';
    }
]);
```

Validation methods can return error messages when they fail. This is a simple way to make error messages dynamic based on the provided value.

Getting Validators From Tables

Once you have created a few validation sets in your table class, you can get the resulting object by name:

```
$defaultValidator = $usersTable->getValidator('default');

$hardenedValidator = $usersTable->getValidator('hardened');
```

Default Validator Class

As stated above, by default the validation methods receive an instance of `Cake\Validation\Validator`. Instead, if you want your custom validator's instance to be used each time, you can use table's `$_validatorClass` property:

```
// In your table class
public function initialize(array $config): void
{
    $this->_validatorClass = \FullyNamespaced\Custom\Validator::class;
}
```

Applying Application Rules

While basic data validation is done when *request data is converted into entities*, many applications also have more complex validation that should only be applied after basic validation has completed.

Where validation ensures the form or syntax of your data is correct, rules focus on comparing data against the existing state of your application and/or network.

These types of rules are often referred to as 'domain rules' or 'application rules'. CakePHP exposes this concept through 'RulesCheckers' which are applied before entities are persisted. Some example application rules are:

- Ensuring email uniqueness
- State transitions or workflow steps, for example, updating an invoice's status.
- Preventing the modification of soft deleted items.
- Enforcing usage/rate limit caps.

Application rules are checked when calling the Table `save()` and `delete()` methods.

Creating a Rules Checker

Rules checker classes are generally defined by the `buildRules()` method in your table class. Behaviors and other event subscribers can use the `Model.buildRules` event to augment the rules checker for a given Table class:

```
use Cake\ORM\RulesChecker;

// In a table class
public function buildRules(RulesChecker $rules): RulesChecker
{
    // Add a rule that is applied for create and update operations
    $rules->add(function ($entity, $options) {
        // Return a boolean to indicate pass/failure
    }, 'ruleName');

    // Add a rule for create.
    $rules->addCreate(function ($entity, $options) {
        // Return a boolean to indicate pass/failure
    }, 'ruleName');

    // Add a rule for update
    $rules->addUpdate(function ($entity, $options) {
        // Return a boolean to indicate pass/failure
    }, 'ruleName');

    // Add a rule for the deleting.
    $rules->addDelete(function ($entity, $options) {
        // Return a boolean to indicate pass/failure
    }, 'ruleName');

    return $rules;
}
```

Your rules functions can expect to get the Entity being checked and an array of options. The options array will contain `errorField`, `message`, and `repository`. The `repository` option will contain the table class the rules are attached to. Because rules accept any `callable`, you can also use instance functions:

```
$rules->addCreate([$this, 'uniqueEmail'], 'uniqueEmail');
```

or callable classes:

```
$rules->addCreate(new IsUnique(['email']), 'uniqueEmail');
```

When adding rules you can define the field the rule is for and the error message as options:

```
$rules->add([$this, 'isValidState'], 'validState', [
    'errorField' => 'status',
    'message' => 'This invoice cannot be moved to that status.'
]);
```

The error will be visible when calling the `getErrors()` method on the entity:

```
$entity->getErrors(); // Contains the domain rules error messages
```

Creating Unique Field Rules

Because unique rules are quite common, CakePHP includes a simple Rule class that allows you to define unique field sets:

```
use Cake\ORM\Rule\IsUnique;

// A single field.
$rules->add($rules->isUnique(['email']));

// A list of fields
$rules->add($rules->isUnique(
    ['username', 'account_id'],
    'This username & account_id combination has already been used.'
));
```

When setting rules on foreign key fields it is important to remember, that only the fields listed are used in the rule. The unique set of rules will be found with `find('all')`. This means that setting `$user->account->id` will not trigger the above rule.

Many database engines allow NULLs to be unique values in UNIQUE indexes. To simulate this, set the `allowMultipleNulls` options to true:

```
$rules->add($rules->isUnique(
    ['username', 'account_id'],
    ['allowMultipleNulls' => true]
));
```

Foreign Key Rules

While you could rely on database errors to enforce constraints, using rules code can help provide a nicer user experience. Because of this CakePHP includes an `ExistsIn` rule class:

```
// A single field.
$rules->add($rules->existsIn('article_id', 'Articles'));

// Multiple keys, useful for composite primary keys.
$rules->add($rules->existsIn(['site_id', 'article_id'], 'Articles'));
```

The fields to check existence against in the related table must be part of the primary key.

You can enforce `existsIn` to pass when nullable parts of your composite foreign key are null:

```
// Example: A composite primary key within NodesTable is (parent_id, site_id).
// A Node may reference a parent Node but does not need to. In latter case, parent_id is␣
↪null.
// Allow this rule to pass, even if fields that are nullable, like parent_id, are null:
$rules->add($rules->existsIn(
    ['parent_id', 'site_id'], // Schema: parent_id NULL, site_id NOT NULL
    'ParentNodes',
    ['allowNullableNulls' => true]
));
```

(continues on next page)

(continued from previous page)

```
// A Node however should in addition also always reference a Site.
$rules->add($rules->existsIn(['site_id'], 'Sites'));
```

In most SQL databases multi-column `UNIQUE` indexes allow multiple null values to exist as `NULL` is not equal to itself. While, allowing multiple null values is the default behavior of CakePHP, you can include null values in your unique checks using `allowMultipleNulls`:

```
// Only one null value can exist in `parent_id` and `site_id`
$rules->add($rules->existsIn(
    ['parent_id', 'site_id'],
    'ParentNodes',
    ['allowMultipleNulls' => false]
));
```

Association Count Rules

If you need to validate that a property or association contains the correct number of values, you can use the `validCount()` rule:

```
// In the ArticlesTable.php file
// No more than 5 tags on an article.
$rules->add($rules->validCount('tags', 5, '<=', 'You can only have 5 tags'));
```

When defining count based rules, the third parameter lets you define the comparison operator to use. ==, >=, <=, >, <, and `!=` are the accepted operators. To ensure a property's count is within a range, use two rules:

```
// In the ArticlesTable.php file
// Between 3 and 5 tags
$rules->add($rules->validCount('tags', 3, '>=', 'You must have at least 3 tags'));
$rules->add($rules->validCount('tags', 5, '<=', 'You must have at most 5 tags'));
```

Note that `validCount` returns `false` if the property is not countable or does not exist:

```
// The save operation will fail if tags is null.
$rules->add($rules->validCount('tags', 0, '<=', 'You must not have any tags'));
```

Association Link Constraint Rule

The `LinkConstraint` lets you emulate SQL constraints in databases that don't support them, or when you want to provide more user friendly error messages when constraints would fail. This rule enables you to check if an association does or does not have related records depending on the mode used:

```
// Ensure that each comment is linked to an Article during updates.
$rules->addUpdate($rules->isLinkedTo(
    'Articles',
    'article',
    'Requires an article'
));

// Ensure that an article has no linked comments during delete.
```

(continues on next page)

(continued from previous page)

```
$rules->addDelete($rules->isNotLinkedTo(
    'Comments',
    'comments',
    'Must have zero comments before deletion.'
));
```

Using Entity Methods as Rules

You may want to use entity methods as domain rules:

```
$rules->add(function ($entity, $options) {
    return $entity->isOkLooking();
}, 'ruleName');
```

Using Conditional Rules

You may want to conditionally apply rules based on entity data:

```
$rules->add(function ($entity, $options) use($rules) {
    if ($entity->role == 'admin') {
        $rule = $rules->existsIn('user_id', 'Admins');

        return $rule($entity, $options);
    }
    if ($entity->role == 'user') {
        $rule = $rules->existsIn('user_id', 'Users');

        return $rule($entity, $options);
    }

    return false;
}, 'userExists');
```

Conditional/Dynamic Error Messages

Rules, being it *custom callables*, or *rule objects*, can either return a boolean, indicating whether they passed, or they can return a string, which means that the rule did not pass, and that the returned string should be used as the error message.

Possible existing error messages defined via the `message` option will be overwritten by the ones returned from the rule:

```
$rules->add(
    function ($entity, $options) {
        if (!$entity->length) {
            return false;
        }

        if ($entity->length < 10) {
            return 'Error message when value is less than 10';
```

(continues on next page)

(continued from previous page)

```
        }

        if ($entity->length > 20) {
            return 'Error message when value is greater than 20';
        }

        return true;
    },
    'ruleName',
    [
        'errorField' => 'length',
        'message' => 'Generic error message used when `false` is returned',
    ]
);
```

Note: Note that in order for the returned message to be actually used, you *must* also supply the `errorField` option, otherwise the rule will just silently fail to pass, ie without an error message being set on the entity!

Creating Custom re-usable Rules

You may want to re-use custom domain rules. You can do so by creating your own invokable rule:

```
// Using a custom rule of the application
use App\ORM\Rule\IsUniqueWithNulls;
// ...
public function buildRules(RulesChecker $rules): RulesChecker
{
    $rules->add(new IsUniqueWithNulls(['parent_id', 'instance_id', 'name']),
↪'uniqueNamePerParent', [
        'errorField' => 'name',
        'message' => 'Name must be unique per parent.',
    ]);
    return $rules;
}
```

See the core rules for examples on how to create such rules.

Creating Custom Rule Objects

If your application has rules that are commonly reused, it is helpful to package those rules into re-usable classes:

```
// in src/Model/Rule/CustomRule.php
namespace App\Model\Rule;

use Cake\Datasource\EntityInterface;

class CustomRule
{
```

(continues on next page)

(continued from previous page)

```
    public function __invoke(EntityInterface $entity, array $options)
    {
        // Do work
        return false;
    }
}

// Add the custom rule
use App\Model\Rule\CustomRule;

$rules->add(new CustomRule(...), 'ruleName');
```

By creating custom rule classes you can keep your code DRY and test your domain rules in isolation.

Disabling Rules

When saving an entity, you can disable the rules if necessary:

```
$articles->save($article, ['checkRules' => false]);
```

Validation vs. Application Rules

The CakePHP ORM is unique in that it uses a two-layered approach to validation.

The first layer is validation. Validation rules are intended to operate in a stateless way. They are best leveraged to ensure that the shape, data types and format of data is correct.

The second layer is application rules. Application rules are best leveraged to check stateful properties of your entities. For example, validation rules could ensure that an email address is valid, while an application rule could ensure that the email address is unique.

As you already discovered, the first layer is done through the `Validator` objects when calling `newEntity()` or `patchEntity()`:

```
$validatedEntity = $articlesTable->newEntity(
    $unsafeData,
    ['validate' => 'customName']
);
$validatedEntity = $articlesTable->patchEntity(
    $entity,
    $unsafeData,
    ['validate' => 'customName']
);
```

In the above example, we'll use a 'custom' validator, which is defined using the `validationCustomName()` method:

```
public function validationCustomName($validator)
{
    $validator->add(
        // ...
    );
```

(continues on next page)

(continued from previous page)

```
    return $validator;
}
```

Validation assumes strings or array are passed since that is what is received from any request:

```
// In src/Model/Table/UsersTable.php
public function validatePasswords($validator)
{
    $validator->add('confirm_password', 'no-misspelling', [
        'rule' => ['compareWith', 'password'],
        'message' => 'Passwords are not equal',
    ]);

    // ...

    return $validator;
}
```

Validation is **not** triggered when directly setting properties on your entities:

```
$userEntity->email = 'not an email!!';
$usersTable->save($userEntity);
```

In the above example the entity will be saved as validation is only triggered for the `newEntity()` and `patchEntity()` methods. The second level of validation is meant to address this situation.

Application rules as explained above will be checked whenever `save()` or `delete()` are called:

```
// In src/Model/Table/UsersTable.php
public function buildRules(RulesChecker $rules): RulesChecker
{
    $rules->add($rules->isUnique(['email']));

    return $rules;
}

// Elsewhere in your application code
$userEntity->email = 'a@duplicated.email';
$usersTable->save($userEntity); // Returns false
```

While Validation is meant for direct user input, application rules are specific for data transitions generated inside your application:

```
// In src/Model/Table/OrdersTable.php
public function buildRules(RulesChecker $rules): RulesChecker
{
    $check = function($order) {
        if ($order->shipping_mode !== 'free') {
            return true;
        }

        return $order->price >= 100;
    };
```

(continues on next page)

(continued from previous page)

```
    $rules->add($check, [
        'errorField' => 'shipping_mode',
        'message' => 'No free shipping for orders under 100!',
    ]);

    return $rules;
}

// Elsewhere in application code
$order->price = 50;
$order->shipping_mode = 'free';
$ordersTable->save($order); // Returns false
```

Using Validation as Application Rules

In certain situations you may want to run the same data validation routines for data that was both generated by users and inside your application. This could come up when running a CLI script that directly sets properties on entities:

```
// In src/Model/Table/UsersTable.php
public function validationDefault(Validator $validator): Validator
{
    $validator->add('email', 'valid_email', [
        'rule' => 'email',
        'message' => 'Invalid email',
    ]);

    // ...

    return $validator;
}

public function buildRules(RulesChecker $rules): RulesChecker
{
    // Add validation rules
    $rules->add(function($entity) {
        $data = $entity->extract($this->getSchema()->columns(), true);
        if (!$entity->isNew() && !empty($data)) {
            $data += $entity->extract((array)$this->getPrimaryKey());
        }
        $validator = $this->getValidator('default');
        $errors = $validator->validate($data, $entity->isNew());
        $entity->setErrors($errors);

        return empty($errors);
    });

    // ...

    return $rules;
}
```

When executed the save will fail thanks to the new application rule that was added:

```
$userEntity->email = 'not an email!!!';
$usersTable->save($userEntity);
$userEntity->getError('email'); // Invalid email
```

The same result can be expected when using `newEntity()` or `patchEntity()`:

```
$userEntity = $usersTable->newEntity(['email' => 'not an email!!']);
$userEntity->getError('email'); // Invalid email
```

Saving Data

class Cake\ORM**Table**

After you have *loaded your data* you will probably want to update and save the changes.

A Glance Over Saving Data

Applications will usually have a couple of ways in which data is saved. The first one is obviously through web forms and the other is by directly generating or changing data in the code to be sent to the database.

Inserting Data

The easiest way to insert data in the database is by creating a new entity and passing it to the `save()` method in the `Table` class:

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

$articlesTable = $this->fetchTable('Articles');
$article = $articlesTable->newEmptyEntity();

$article->title = 'A New Article';
$article->body = 'This is the body of the article';

if ($articlesTable->save($article)) {
    // The $article entity contains the id now
    $id = $article->id;
}
```

Updating Data

Updating your data is achieved by using the `save()` method

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

$articlesTable = $this->fetchTable('Articles');
$article = $articlesTable->get(12); // Return article with id 12
```

(continues on next page)

(continued from previous page)

```
$article->title = 'CakePHP is THE best PHP framework!';
$articlesTable->save($article);
```

CakePHP will know whether to perform an insert or an update based on the return value of the `isNew()` method. Entities that were retrieved with `get()` or `find()` will always return `false` when `isNew()` is called on them.

Saving With Associations

By default the `save()` method will also save one level of associations:

```
$articlesTable = $this->fetchTable('Articles');
$author = $articlesTable->Authors->findByUserName('mark')->first();

$article = $articlesTable->newEmptyEntity();
$article->title = 'An article by mark';
$article->author = $author;

if ($articlesTable->save($article)) {
    // The foreign key value was set automatically.
    echo $article->author_id;
}
```

The `save()` method is also able to create new records for associations:

```
$firstComment = $articlesTable->Comments->newEmptyEntity();
$firstComment->body = 'The CakePHP features are outstanding';

$secondComment = $articlesTable->Comments->newEmptyEntity();
$secondComment->body = 'CakePHP performance is terrific!';

$tag1 = $articlesTable->Tags->findByName('cakephp')->first();
$tag2 = $articlesTable->Tags->newEmptyEntity();
$tag2->name = 'awesome';

$article = $articlesTable->get(12);
$article->comments = [$firstComment, $secondComment];
$article->tags = [$tag1, $tag2];

$articlesTable->save($article);
```

Associate Many To Many Records

The previous example demonstrates how to associate a few tags to an article. Another way of accomplishing the same thing is by using the `link()` method in the association:

```
$tag1 = $articlesTable->Tags->findByName('cakephp')->first();
$tag2 = $articlesTable->Tags->newEmptyEntity();
$tag2->name = 'awesome';

$articlesTable->Tags->link($article, [$tag1, $tag2]);
```

Unlink Many To Many Records

Unlinking many to many records is done via the `unlink()` method:

```
$tags = $articlesTable
    ->Tags
    ->find()
    ->where(['name IN' => ['cakephp', 'awesome']])
    ->toList();

$articlesTable->Tags->unlink($article, $tags);
```

When modifying records by directly setting or changing the properties no validation happens, which is a problem when accepting form data. The following sections will demonstrate how to efficiently convert form data into entities so that they can be validated and saved.

Converting Request Data into Entities

Before editing and saving data back to your database, you'll need to convert the request data from the array format held in the request, and the entities that the ORM uses. The Table class provides an efficient way to convert one or many entities from request data. You can convert a single entity using:

```
// In a controller

$articles = $this->fetchTable('Articles');

// Validate and convert to an Entity object
$entity = $articles->newEntity($this->request->getData());
```

Note: If you are using newEntity() and the resulting entities are missing some or all of the data they were passed, double check that the columns you want to set are listed in the `$_accessible` property of your entity. See *Mass Assignment*.

The request data should follow the structure of your entities. For example if you have an article, which belonged to a user, and had many comments, your request data should resemble:

```
$data = [
    'title' => 'CakePHP For the Win',
    'body' => 'Baking with CakePHP makes web development fun!',
    'user_id' => 1,
    'user' => [
        'username' => 'mark',
    ],
    'comments' => [
        ['body' => 'The CakePHP features are outstanding'],
        ['body' => 'CakePHP performance is terrific!'],
    ]
];
```

By default, the `newEntity()` method validates the data that gets passed to it, as explained in the *Validating Data Before Building Entities* section. If you wish to bypass data validation pass the `'validate' => false` option:

```
$entity = $articles->newEntity($data, ['validate' => false]);
```

When building forms that save nested associations, you need to define which associations should be marshalled:

```
// In a controller

$articles = $this->fetchTable('Articles');

// New entity with nested associations
$entity = $articles->newEntity($this->request->getData(), [
    'associated' => [
        'Tags', 'Comments' => ['associated' => ['Users']],
    ]
]);
```

The above indicates that the 'Tags', 'Comments' and 'Users' for the Comments should be marshalled. Alternatively, you can use dot notation for brevity:

```
// In a controller

$articles = $this->fetchTable('Articles');

// New entity with nested associations using dot notation
$entity = $articles->newEntity($this->request->getData(), [
    'associated' => ['Tags', 'Comments.Users'],
]);
```

You may also disable marshalling of possible nested associations like so:

```
$entity = $articles->newEntity($data, ['associated' => []]);
// or...
$entity = $articles->patchEntity($entity, $data, ['associated' => []]);
```

Associated data is also validated by default unless told otherwise. You may also change the validation set to be used per association:

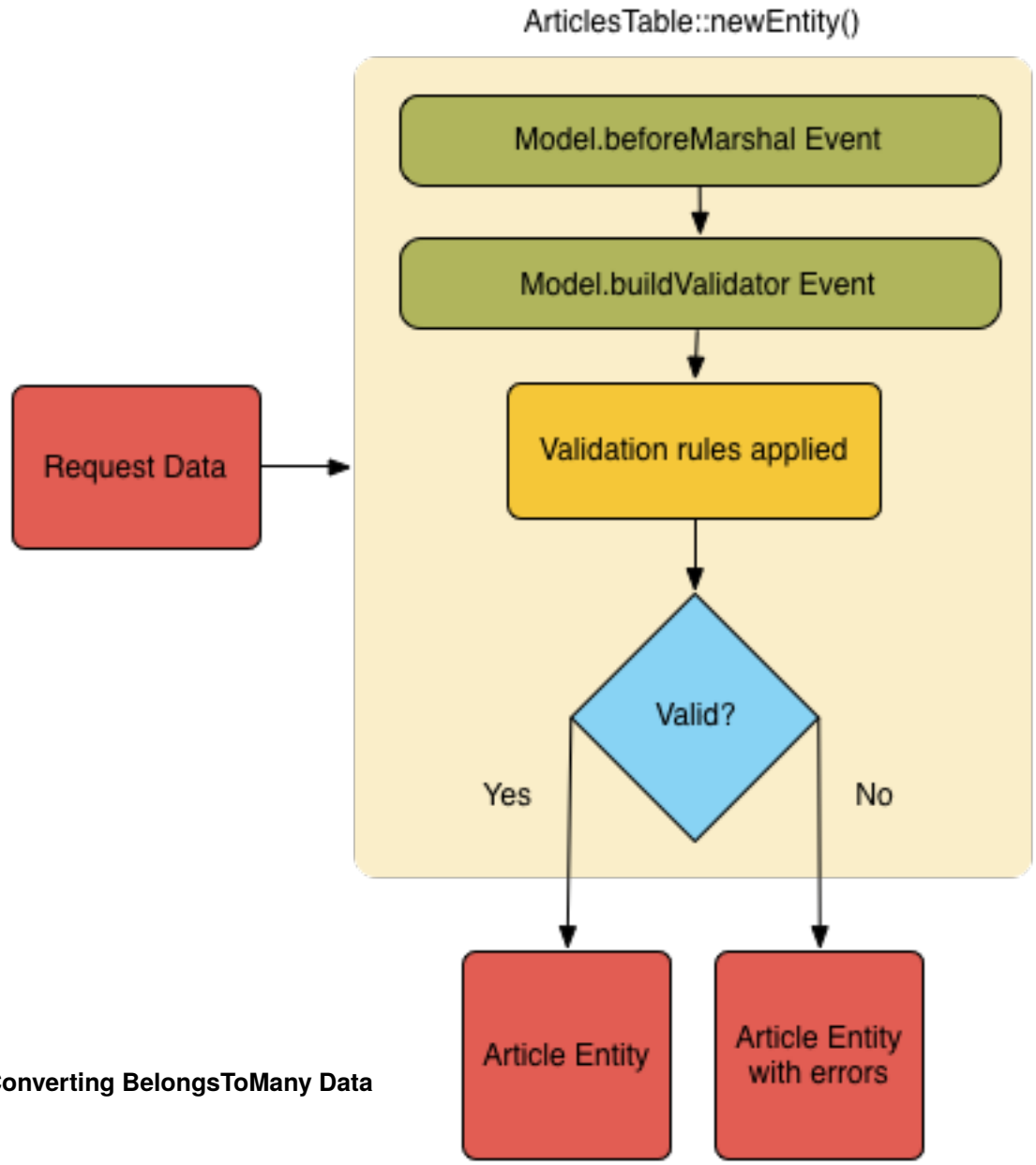
```
// In a controller

$articles = $this->fetchTable('Articles');

// Bypass validation on Tags association and
// Designate 'signup' validation set for Comments.Users
$entity = $articles->newEntity($this->request->getData(), [
    'associated' => [
        'Tags' => ['validate' => false],
        'Comments.Users' => ['validate' => 'signup'],
    ],
]);
```

The *Using A Different Validation Set For Associations* chapter has more information on how to use different validators for associated marshalling.

The following diagram gives an overview of what happens inside the `newEntity()` or `patchEntity()` method:



You can always count on getting an entity back from `newEntity()`. If validation fails your entity will contain errors, and any invalid fields will not be populated in the created entity.

Converting BelongsToMany Data

If you are saving belongsToMany associations you can either use a list of entity data or a list of ids. When using a list of entity data your request data should look like:

```
$data = [
    'title' => 'My title',
    'body' => 'The text',
    'user_id' => 1,
    'tags' => [
        ['name' => 'CakePHP'],
```

(continues on next page)

(continued from previous page)

```
        ['name' => 'Internet'],
    ],
];
```

The above will create 2 new tags. If you want to link an article with existing tags you can use a list of ids. Your request data should look like:

```
$data = [
    'title' => 'My title',
    'body' => 'The text',
    'user_id' => 1,
    'tags' => [
        '_ids' => [1, 2, 3, 4],
    ],
];
```

If you need to link against some existing belongsToMany records, and create new ones at the same time you can use an expanded format:

```
$data = [
    'title' => 'My title',
    'body' => 'The text',
    'user_id' => 1,
    'tags' => [
        ['name' => 'A new tag'],
        ['name' => 'Another new tag'],
        ['id' => 5],
        ['id' => 21],
    ],
];
```

When the above data is converted into entities, you will have 4 tags. The first two will be new objects, and the second two will be references to existing records.

When converting belongsToMany data, you can disable entity creation, by using the `onlyIds` option:

```
$result = $articles->patchEntity($entity, $data, [
    'associated' => ['Tags' => ['onlyIds' => true]],
]);
```

When used, this option restricts belongsToMany association marshalling to only use the `_ids` data.

Converting HasMany Data

If you want to update existing hasMany associations and update their properties, you should first ensure your entity is loaded with the hasMany association populated. You can then use request data similar to:

```
$data = [
    'title' => 'My Title',
    'body' => 'The text',
    'comments' => [
        ['id' => 1, 'comment' => 'Update the first comment'],
```

(continues on next page)

(continued from previous page)

```
        ['id' => 2, 'comment' => 'Update the second comment'],
        ['comment' => 'Create a new comment'],
    ],
];
```

If you are saving hasMany associations and want to link existing records to a new parent record you can use the `_ids` format:

```
$data = [
    'title' => 'My new article',
    'body' => 'The text',
    'user_id' => 1,
    'comments' => [
        '_ids' => [1, 2, 3, 4],
    ],
];
```

When converting hasMany data, you can disable the new entity creation, by using the `onlyIds` option. When enabled, this option restricts hasMany marshalling to only use the `_ids` key and ignore all other data.

Converting Multiple Records

When creating forms that create/update multiple records at once you can use `newEntities()`:

```
// In a controller.

$articles = $this->fetchTable('Articles');
$entities = $articles->newEntities($this->request->getData());
```

In this situation, the request data for multiple articles should look like:

```
$data = [
    [
        'title' => 'First post',
        'published' => 1,
    ],
    [
        'title' => 'Second post',
        'published' => 1,
    ],
];
```

Once you've converted request data into entities you can save:

```
// In a controller.
foreach ($entities as $entity) {
    // Save entity
    $articles->save($entity);
}
```

The above will run a separate transaction for each entity saved. If you'd like to process all the entities as a single transaction you can use `saveMany()` or `saveManyOrFail()`:

```
// Get a boolean indicating success
$articles->saveMany($entities);

// Get a PersistenceFailedException if any records fail to save.
$articles->saveManyOrFail($entities);
```

Changing Accessible Fields

It's also possible to allow `newEntity()` to write into non accessible fields. For example, `id` is usually absent from the `_accessible` property. In such case, you can use the `accessibleFields` option. It could be useful to keep ids of associated entities:

```
// In a controller

$articles = $this->fetchTable('Articles');
$entity = $articles->newEntity($this->request->getData(), [
    'associated' => [
        'Tags', 'Comments' => [
            'associated' => [
                'Users' => [
                    'accessibleFields' => ['id' => true],
                ],
            ],
        ],
    ],
]);
```

The above will keep the association unchanged between Comments and Users for the concerned entity.

Note: If you are using newEntity() and the resulting entities are missing some or all of the data they were passed, double check that the columns you want to set are listed in the `$_accessible` property of your entity. See *Mass Assignment*.

Merging Request Data Into Entities

In order to update entities you may choose to apply request data directly to an existing entity. This has the advantage that only the fields that actually changed will be saved, as opposed to sending all fields to the database to be persisted. You can merge an array of raw data into an existing entity using the `patchEntity()` method:

```
// In a controller.

$articles = $this->fetchTable('Articles');
$article = $articles->get(1);
$articles->patchEntity($article, $this->request->getData());
$articles->save($article);
```

Validation and patchEntity

Similar to `newEntity()`, the `patchEntity` method will validate the data before it is copied to the entity. The mechanism is explained in the *Validating Data Before Building Entities* section. If you wish to disable validation while patching an entity, pass the `validate` option as follows:

```
// In a controller.

$articles = $this->fetchTable('Articles');
$article = $articles->get(1);
$articles->patchEntity($article, $data, ['validate' => false]);
```

You may also change the validation set used for the entity or any of the associations:

```
$articles->patchEntity($article, $this->request->getData(), [
    'validate' => 'custom',
    'associated' => ['Tags', 'Comments.Users' => ['validate' => 'signup']]
]);
```

Patching HasMany and BelongsToMany

As explained in the previous section, the request data should follow the structure of your entity. The `patchEntity()` method is equally capable of merging associations, by default only the first level of associations are merged, but if you wish to control the list of associations to be merged or merge deeper to deeper levels, you can use the third parameter of the method:

```
// In a controller.
$associated = ['Tags', 'Comments.Users'];
// or using nested arrays
$associated = ['Tags', 'Comments' => ['associated' => ['Users']]];
$article = $articles->get(1, ['contain' => $associated]);
$articles->patchEntity($article, $this->request->getData(), [
    'associated' => $associated,
]);
$articles->save($article);
```

Associations are merged by matching the primary key field in the source entities to the corresponding fields in the data array. Associations will construct new entities if no previous entity is found for the association's target property.

For example give some request data like the following:

```
$data = [
    'title' => 'My title',
    'user' => [
        'username' => 'mark',
    ],
];
```

Trying to patch an entity without an entity in the user property will create a new user entity:

```
// In a controller.
$entity = $articles->patchEntity(new Article, $data);
echo $entity->user->username; // Echoes 'mark'
```

The same can be said about hasMany and belongsToMany associations, with an important caveat:

Note: For belongsToMany associations, ensure the relevant entity has a property accessible for the associated entity.

If a Product belongsToMany Tag:

```
// in the Product Entity
protected array $_accessible = [
    // .. other properties
   'tags' => true,
];
```

Note: For hasMany and belongsToMany associations, if there were any entities that could not be matched by primary key to a record in the data array, then those records will be discarded from the resulting entity.

Remember that using either `patchEntity()` or `patchEntities()` does not persist the data, it just edits (or creates) the given entities. In order to save the entity you will have to call the table's `save()` method.

For example, consider the following case:

```
$data = [
    'title' => 'My title',
    'body' => 'The text',
    'comments' => [
        ['body' => 'First comment', 'id' => 1],
        ['body' => 'Second comment', 'id' => 2],
    ],
];
$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);
```

At the end, if the entity is converted back to an array you will obtain the following result:

```
[
    'title' => 'My title',
    'body' => 'The text',
    'comments' => [
        ['body' => 'Changed comment', 'id' => 1],
        ['body' => 'A new comment'],
    ],
];
```

As you can see, the comment with id 2 is no longer there, as it could not be matched to anything in the `$newData` array. This happens because CakePHP is reflecting the new state described in the request data.

Some additional advantages of this approach is that it reduces the number of operations to be executed when persisting the entity again.

Please note that this does not mean that the comment with id 2 was removed from the database, if you wish to remove the comments for that article that are not present in the entity, you can collect the primary keys and execute a batch delete for those not in the list:

```
// In a controller.
use Cake\Collection\Collection;

$comments = $this->fetchTable('Comments');
$present = (new Collection($entity->comments))->extract('id')->filter()->toList();
$comments->deleteAll([
    'article_id' => $article->id,
    'id NOT IN' => $present,
]);
```

As you can see, this also helps creating solutions where an association needs to be implemented like a single set.

You can also patch multiple entities at once. The consideration made for patching hasMany and belongsToMany associations apply for patching multiple entities: Matches are done by the primary key field value and missing matches in the original entities array will be removed and not present in the result:

```
// In a controller.

$articles = $this->fetchTable('Articles');
$list = $articles->find('popular')->toList();
$patched = $articles->patchEntities($list, $this->request->getData());
foreach ($patched as $entity) {
    $articles->save($entity);
}
```

Similarly to using `patchEntity()`, you can use the third argument for controlling the associations that will be merged in each of the entities in the array:

```
// In a controller.
$patched = $articles->patchEntities(
    $list,
    $this->request->getData(),
    ['associated' => ['Tags', 'Comments.Users']]
);
```

Modifying Request Data Before Building Entities

If you need to modify request data before it is converted into entities, you can use the `Model.beforeMarshal` event. This event lets you manipulate the request data just before entities are created:

```
// Include use statements at the top of your file.
use Cake\Event\EventInterface;
use ArrayObject;

// In a table or behavior class
public function beforeMarshal(EventInterface $event, ArrayObject $data, ArrayObject
↪$options)
```

(continues on next page)

(continued from previous page)

```
{
    if (isset($data['username'])) {
        $data['username'] = mb_strtolower($data['username']);
    }
}
```

The `$data` parameter is an `ArrayObject` instance, so you don't have to return it to change the data used to create entities.

The main purpose of `beforeMarshal` is to assist the users to pass the validation process when simple mistakes can be automatically resolved, or when data needs to be restructured so it can be put into the right fields.

The `Model.beforeMarshal` event is triggered just at the start of the validation process, one of the reasons is that `beforeMarshal` is allowed to change the validation rules and the saving options, such as the field list. Validation is triggered just after this event is finished. A common example of changing the data before it is validated is trimming all fields before saving:

```
// Include use statements at the top of your file.
use Cake\Event\EventInterface;
use ArrayObject;

// In a table or behavior class
public function beforeMarshal(EventInterface $event, ArrayObject $data, ArrayObject
↪$options)
{
    foreach ($data as $key => $value) {
        if (is_string($value)) {
            $data[$key] = trim($value);
        }
    }
}
```

Because of how the marshalling process works, if a field does not pass validation it will automatically be removed from the data array and not be copied into the entity. This is to prevent inconsistent data from entering the entity object.

Moreover, the data in `beforeMarshal` is a copy of the passed data. This is because it is important to preserve the original user input, as it may be used elsewhere.

Modifying Entities After Updating From Request Data

The `Model.afterMarshal` event allows you to modify entities after they have been created or updated from request data. It can be useful to apply additional validation logic that you cannot easily express through Validator methods:

```
// Include use statements at the top of your file.
use Cake\Event\EventInterface;
use Cake\ORM\EntityInterface;
use ArrayObject;

// In a table or behavior class
public function afterMarshal(
    EventInterface $event,
    EntityInterface $entity,
    ArrayObject $data,
```

(continues on next page)

(continued from previous page)

```
    ArrayObject $options
) {
    // Don't accept people who have a name starting with J on the 20th
    // of each month.
    if (mb_substr($entity->name, 1) === 'J' && (int)date('d') === 20) {
        $entity->setError('name', 'No J people today sorry.');
    }
}
```

Validating Data Before Building Entities

The *Validating Data* chapter has more information on how to use the validation features of CakePHP to ensure your data stays correct and consistent.

Avoiding Property Mass Assignment Attacks

When creating or merging entities from request data you need to be careful of what you allow your users to change or add in the entities. For example, by sending an array in the request containing the `user_id` an attacker could change the owner of an article, causing undesirable effects:

```
// Contains ['user_id' => 100, 'title' => 'Hacked!'];
$data = $this->request->getData();
$entity = $this->patchEntity($entity, $data);
$this->save($entity);
```

There are two ways of protecting you against this problem. The first one is by setting the default columns that can be safely set from a request using the *Mass Assignment* feature in the entities.

The second way is by using the `fields` option when creating or merging data into an entity:

```
// Contains ['user_id' => 100, 'title' => 'Hacked!'];
$data = $this->request->getData();

// Only allow title to be changed
$entity = $this->patchEntity($entity, $data, [
    'fields' => ['title']
]);
$this->save($entity);
```

You can also control which properties can be assigned for associations:

```
// Only allow changing the title and tags
// and the tag name is the only column that can be set
$entity = $this->patchEntity($entity, $data, [
    'fields' => ['title', 'tags'],
    'associated' => ['Tags' => ['fields' => ['name']]]
]);
$this->save($entity);
```

Using this feature is handy when you have many different functions your users can access and you want to let your users edit different data based on their privileges.

Saving Entities

`Cake\ORM\Table::save(`*Entity $entity*, *array $options = []*`)`

When saving request data to your database you need to first hydrate a new entity using `newEntity()` for passing into `save()`. For example:

```
// In a controller

$articles = $this->fetchTable('Articles');
$article = $articles->newEntity($this->request->getData());
if ($articles->save($article)) {
    // ...
}
```

The ORM uses the `isNew()` method on an entity to determine whether or not an insert or update should be performed. If the `isNew()` method returns `true` and the entity has a primary key value, an 'exists' query will be issued. The 'exists' query can be suppressed by passing `'checkExisting' => false` in the `$options` argument:

```
$articles->save($article, ['checkExisting' => false]);
```

Once you've loaded some entities you'll probably want to modify them and update your database. This is a pretty simple exercise in CakePHP:

```
$articles = $this->fetchTable('Articles');
$article = $articles->find('all')->where(['id' => 2])->first();

$article->title = 'My new title';
$articles->save($article);
```

When saving, CakePHP will *apply your rules*, and wrap the save operation in a database transaction. It will also only update properties that have changed. The above `save()` call would generate SQL like:

```
UPDATE articles SET title = 'My new title' WHERE id = 2;
```

If you had a new entity, the following SQL would be generated:

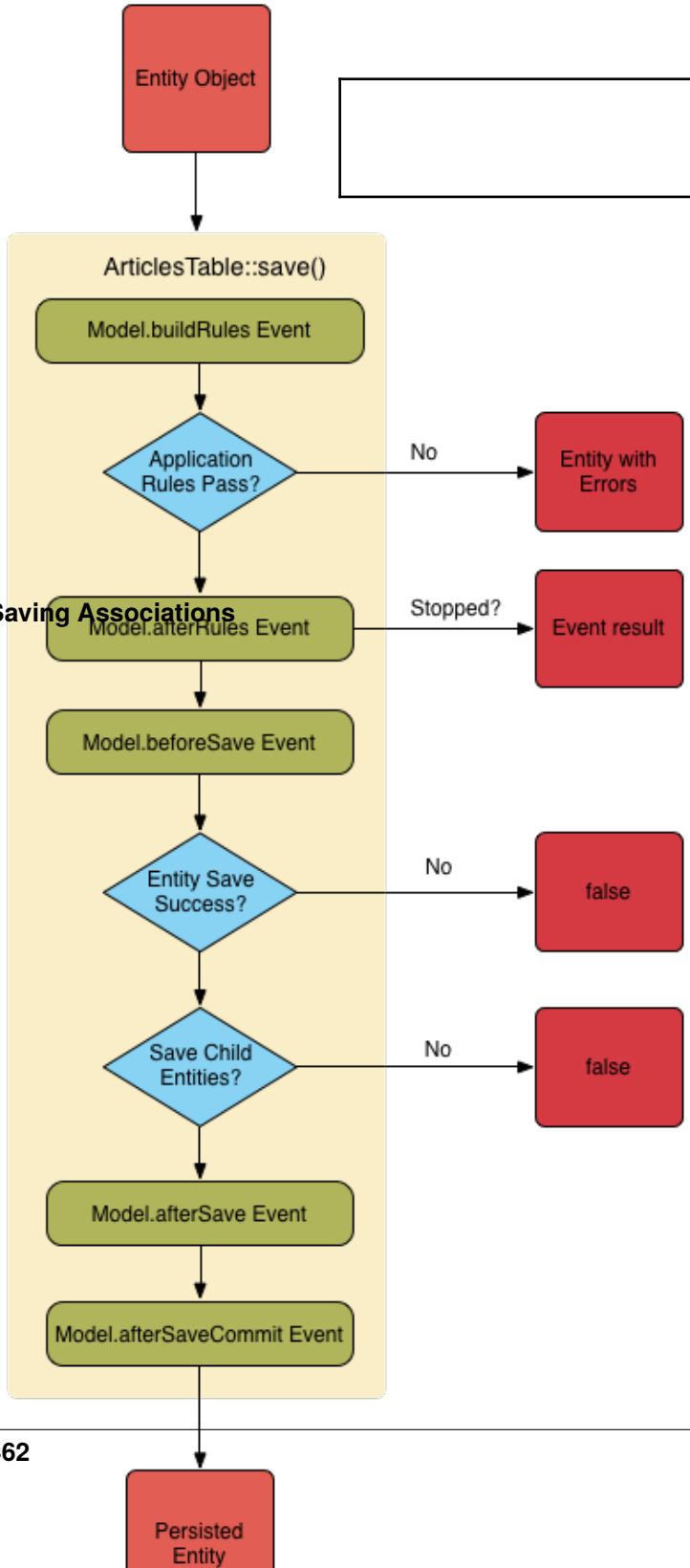
```
INSERT INTO articles (title) VALUES ('My new title');
```

When an entity is saved a few things happen:

1. Rule checking will be started if not disabled.
2. Rule checking will trigger the `Model.beforeRules` event. If this event is stopped, the save operation will fail and return `false`.
3. Rules will be checked. If the entity is being created, the `create` rules will be used. If the entity is being updated, the `update` rules will be used.
4. The `Model.afterRules` event will be triggered.
5. The `Model.beforeSave` event is dispatched. If it is stopped, the save will be aborted, and save() will return `false`.
6. Parent associations are saved. For example, any listed belongsTo associations will be saved.
7. The modified fields on the entity will be saved.

8. Child associations are saved. For example, any listed hasMany, hasOne, or belongsToMany associations will be saved.
9. The `Model.afterSave` event will be dispatched.
10. The `Model.afterSaveCommit` event will be dispatched.

The following diagram illustrates the above process:



See the *Applying Application Rules* section for more information on creating and using rules.

Warning: If no changes a
tity when it is saved, the cal
because no save is perform

The `save()` method will return the modified entity on success, and `false` on failure. You can disable rules and/or transactions using the `$options` argument for save:

```
//␣
↪In a controller or table method.
$articles-
↪>save($article, ['checkRules
↪' => false, 'atomic' => false]);
```

Saving Associations

When you are saving an entity, you can also choose to save some or all of the associated entities. By default all first level entities will be saved. For example saving an Article, will also automatically update any dirty entities that are directly related to articles table.

You can fine tune which associations are saved by using the `associated` option:

```
// In a controller.

// Only␣
↪save the comments association
$articles->save($entity,␣
↪['associated' => ['Comments']]);
```

You can define save distant or deeply nested associations by using dot notation:

```
// Save the company,
↪ the employees and related␣
↪addresses for each of them.
```

(continues on next page)

(continued from previous page)

```
$companies-
↪>save($entity, ['associated
↪' => ['Employees.Addresses']]);
```

Moreover, you can combine the dot notation for associations with the options array:

```
$companies->save($entity, [
  'associated' => [
    'Employees',
    'Employees.Addresses'
  ]
]);
```

Your entities should be structured in the same way as they are when loaded from the database. See the form helper documentation for *how to build inputs for associations*.

If you are building or modifying association data after building your entities you will have to mark the association property as modified with `setDirty()`:

```
$company-
↪>author->name = 'Master Chef';
$company->setDirty('author',␣
↪true);
```

Saving BelongsTo Associations

When saving belongsTo associations, the ORM expects a single nested entity named with the singular, *underscored* version of the association name. For example:

```
// In a controller.
$data = [
    'title' => 'First Post',
    'user' => [
        'id' => 1,
        'username' => 'mark',
    ],
];

$articles = $this->fetchTable('Articles');
$article = $articles->newEntity($data, [
    'associated' => ['Users']
]);

$articles->save($article);
```

Saving HasOne Associations

When saving hasOne associations, the ORM expects a single nested entity named with the singular, *underscored* version of the association name. For example:

```
// In a controller.
$data = [
    'id' => 1,
    'username' => 'cakephp',
    'profile' => [
        'twitter' => '@cakephp',
    ],
];

$users = $this->fetchTable('Users');
$user = $users->newEntity($data, [
    'associated' => ['Profiles']
]);
$users->save($user);
```

Saving HasMany Associations

When saving hasMany associations, the ORM expects an array of entities named with the plural, *underscored* version of the association name. For example:

```
// In a controller.
$data = [
    'title' => 'First Post',
    'comments' => [
        ['body' => 'Best post ever'],
        ['body' => 'I really like this.']
    ]
];

$articles = $this->fetchTable('Articles');
$article = $articles->newEntity($data, [
    'associated' => ['Comments']
]);
$articles->save($article);
```

When saving hasMany associations, associated records will either be updated, or inserted. For the case that the record already has associated records in the database, you have the choice between two saving strategies:

append
: Associated records are updated in the database or, if not matching any existing record, inserted.

replace
: Any existing records that do not match the records provided will be deleted from the database. Only provided records will remain (or be inserted).

By default the `append` saving strategy is used. See *HasMany Associations* for details on defining the `saveStrategy`.

Whenever you add new records to an existing association you should always mark the association property as 'dirty'. This lets the ORM know that the association property has to be persisted:

```
$article->comments[] = $comment;
$article->setDirty('comments', true);
```

Without the call to `setDirty()` the updated comments will not be saved.

If you are creating a new entity, and want to add existing records to a has many/belongs to many association you need to initialize the association property first:

```
$article->comments = [];
```

Without initialization calling `$article->comments[] = $comment;` will have no effect.

Saving BelongsToMany Associations

When saving belongsToMany associations, the ORM expects an array of entities named with the plural, *underscored* version of the association name. For example:

```
// In a controller.
$data = [
    'title' => 'First Post',
    'tags' => [
        ['tag' => 'CakePHP'],
        ['tag' => 'Framework']
    ]
];

$articles = $this->fetchTable('Articles');
$article = $articles->newEntity($data, [
    'associated' => ['Tags']
]);
$articles->save($article);
```

When converting request data into entities, the `newEntity()` and `newEntities()` methods will handle both arrays of properties, as well as a list of ids at the `_ids` key. Using the `_ids` key makes it possible to building a select box or checkbox based form controls for belongs to many associations. See the *Converting Request Data into Entities* section for more information.

When saving belongsToMany associations, you have the choice between two saving strategies:

append
: Only new links will be created between each side of this association. This strategy will not destroy existing links even though they may not be present in the array of entities to be saved.

replace
: When saving, existing links will be removed and new links will be created in the junction table. If there are existing link in the database to some of the entities intended to be saved, those links will be updated, not deleted and then re-saved.

See *BelongsToMany Associations* for details on defining the `saveStrategy`.

By default the `replace` strategy is used. Whenever you add new records into an existing association you should always mark the association property as 'dirty'. This lets the ORM know that the association property has to be persisted:

```
$article->tags[] = $tag;
$article->setDirty('tags', true);
```

Without the call to `setDirty()` the updated tags will not be saved.

Often you'll find yourself wanting to make an association between two existing entities, eg. a user coauthoring an article. This is done by using the method `link()`, like this:

```
$article = $this->Articles->get($articleId);
$user = $this->Users->get($userId);

$this->Articles->Users->link($article, [$user]);
```

When saving belongsToMany Associations, it can be relevant to save some additional data to the junction Table. In the previous example of tags, it could be the `vote_type` of person who voted on that article. The `vote_type` can be either `upvote` or `downvote` and is represented by a string. The relation is between Users and Articles.

Saving that association, and the `vote_type` is done by first adding some data to `_joinData` and then saving the association with `link()`, example:

```
$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]);
```

Saving Additional Data to the Join Table

In some situations the table joining your BelongsToMany association, will have additional columns on it. CakePHP makes it simple to save properties into these columns. Each entity in a belongsToMany association has a `_joinData` property that contains the additional columns on the junction table. This data can be either an array or an Entity instance. For example if Students BelongsToMany Courses, we could have a junction table that looks like:

```
id | student_id | course_id | days_attended | grade
```

When saving data you can populate the additional columns on the junction table by setting data to the `_joinData` property:

```
$student->courses[0]->_joinData->grade = 80.12;
$student->courses[0]->_joinData->days_attended = 30;

$studentsTable->save($student);
```

The example above will only work if the property `_joinData` is already a reference to a Join Table Entity. If you don't already have a `_joinData` entity, you can create one using `newEntity()`:

```
$coursesMembershipsTable = $this->fetchTable('CoursesMemberships');
$student->courses[0]->_joinData = $coursesMembershipsTable->newEntity([
    'grade' => 80.12,
    'days_attended' => 30
]);

$studentsTable->save($student);
```

The `_joinData` property can be either an entity, or an array of data if you are saving entities built from request data. When saving junction table data from request data your POST data should look like:

```
$data = [
    'first_name' => 'Sally',
    'last_name' => 'Parker',
    'courses' => [
        [
            'id' => 10,
            '_joinData' => [
                'grade' => 80.12,
                'days_attended' => 30
            ]
        ],
        // Other courses.
    ]
];
$student = $this->Students->newEntity($data, [
    'associated' => ['Courses._joinData']
]);
```

See the *Creating Inputs for Associated Data* documentation for how to build inputs with `FormHelper` correctly.

Saving Complex Types

Tables are capable of storing data represented in basic types, like strings, integers, floats, booleans, etc. But It can also be extended to accept more complex types such as arrays or objects and serialize this data into simpler types that can be saved in the database.

This functionality is achieved by using the custom types system. See the *Adding Custom Types* section to find out how to build custom column Types:

```
use Cake\Database\TypeFactory;

TypeFactory::map('json', 'Cake\Database\Type\JsonType');

// In src/Model/Table/UsersTable.php

class UsersTable extends Table
{
    public function getSchema(): TableSchemaInterface
    {
        $schema = parent::getSchema();
        $schema->setColumnType('preferences', 'json');

        return $schema;
    }
}
```

The code above maps the `preferences` column to the `json` custom type. This means that when retrieving data for that column, it will be unserialized from a JSON string in the database and put into an entity as an array.

Likewise, when saved, the array will be transformed back into its JSON representation:

```
$user = new User([
    'preferences' => [
```

(continues on next page)

(continued from previous page)

```
        'sports' => ['football', 'baseball'],
        'books' => ['Mastering PHP', 'Hamlet']
    ]
]);
$usersTable->save($user);
```

When using complex types it is important to validate that the data you are receiving from the end user is the correct type. Failing to correctly handle complex data could result in malicious users being able to store data they would not normally be able to.

Strict Saving

`Cake\ORM\Table::saveOrFail`(*$entity*, *$options = []*)

Using this method will throw an *Cake\ORM\Exception\PersistenceFailedException* if:

- the application rules checks failed
- the entity contains errors
- the save was aborted by a callback.

Using this can be helpful when you performing complex database operations without human monitoring, for example, inside a Shell task.

Note: If you use this method in a controller, be sure to catch the `PersistenceFailedException` that could be raised.

If you want to track down the entity that failed to save, you can use the `Cake\ORMException\PersistenceFailedException::getEntity()` method:

```
try {
    $table->saveOrFail($entity);
} catch (\Cake\ORM\Exception\PersistenceFailedException $e) {
    echo $e->getEntity();
}
```

As this internally performs a *Cake\ORM\Table::save()* call, all corresponding save events will be triggered.

Find or Create an Entity

`Cake\ORM\Table::findOrCreate`(*$search*, *$callback = null*, *$options = []*)

Find an existing record based on `$search` or create a new record using the properties in `$search` and calling the optional `$callback`. This method is ideal in scenarios where you need to reduce the chance of duplicate records:

```
$record = $table->findOrCreate(
    ['email' => 'bobbi@example.com'],
    function ($entity) use ($otherData) {
        // Only called when a new record is created.
        $entity->name = $otherData['name'];
    }
);
```

If your find conditions require custom order, associations or conditions, then the `$search` parameter can be a callable or `SelectQuery` object. If you use a callable, it should take a `SelectQuery` as its argument.

The returned entity will have been saved if it was a new record. The supported options for this method are:

- `atomic` Should the find and save operation be done inside a transaction.
- `defaults` Set to `false` to not set `$search` properties into the created entity.

Creating with an existing primary key

When handling UUID primary keys you often want to provide an externally generated value, and not have an an identifier generated for you.

In this case make sure you are not passing the primary key as part of the marshalled data. Instead, assign the primary key and then patch in the remaining entity data:

```
$record = $table->newEmptyEntity();
$record->id = $existingUuid;
$record = $table->patchEntity($record, $existingData);
$table->saveOrFail($record);
```

Saving Multiple Entities

`Cake\ORM\Table::`**`saveMany`**(*$entities*, *$options = []*)

Using this method you can save multiple entities atomically. `$entities` can be an array of entities created using `newEntities()` / `patchEntities()`. `$options` can have the same options as accepted by `save()`:

```
$data = [
    [
        'title' => 'First post',
        'published' => 1
    ],
    [
        'title' => 'Second post',
        'published' => 1
    ],
];

$articles = $this->fetchTable('Articles');
$entities = $articles->newEntities($data);
$result = $articles->saveMany($entities);
```

The result will be updated entities on success or `false` on failure.

Bulk Updates

`Cake\ORM\Table::updateAll(`*$fields*, *$conditions*`)`

There may be times when updating rows individually is not efficient or necessary. In these cases it is more efficient to use a bulk-update to modify many rows at once, by assigning the new field values, and conditions for the update:

```
// Publish all the unpublished articles.
function publishAllUnpublished()
{
    $this->updateAll(
        [  // fields
            'published' => true,
            'publish_date' => DateTime::now()
        ],
        [  // conditions
            'published' => false
        ]
    );
}
```

If you need to do bulk updates and use SQL expressions, you will need to use an expression object as `updateAll()` uses prepared statements under the hood:

```
use Cake\Database\Expression\QueryExpression;

...

function incrementCounters()
{
    $expression = new QueryExpression('view_count = view_count + 1');
    $this->updateAll([$expression], ['published' => true]);
}
```

A bulk-update will be considered successful if 1 or more rows are updated.

Warning: updateAll will *not* trigger beforeSave/afterSave events. If you need those first load a collection of records and update them.

`updateAll()` is for convenience only. You can use this more flexible interface as well:

```
// Publish all the unpublished articles.
function publishAllUnpublished()
{
    $this->updateQuery()
        ->set(['published' => true])
        ->where(['published' => false])
        ->execute();
}
```

Also see: *Updating Data*.

Deleting Data

class Cake\ORM**Table**

Cake\ORM\Table::**delete**(*Entity $entity*, *$options = []*)

Once you've loaded an entity you can delete it by calling the originating table's delete method:

```
// In a controller.
$entity = $this->Articles->get(2);
$result = $this->Articles->delete($entity);
```

When deleting entities a few things happen:

1. The *delete rules* will be applied. If the rules fail, deletion will be prevented.
2. The `Model.beforeDelete` event is triggered. If this event is stopped, the delete will be aborted and the event's result will be returned.
3. The entity will be deleted.
4. All dependent associations will be deleted. If associations are being deleted as entities, additional events will be dispatched.
5. Any junction table records for BelongsToMany associations will be removed.
6. The `Model.afterDelete` event will be triggered.

By default all deletes happen within a transaction. You can disable the transaction with the atomic option:

```
$result = $this->Articles->delete($entity, ['atomic' => false]);
```

The `$options` parameter supports the following options:

- `atomic` Defaults to true. When true the deletion happens within a transaction.
- `checkRules` Defaults to true. Check deletion rules before deleting records.

Cascading Deletes

When deleting entities, associated data can also be deleted. If your HasOne and HasMany associations are configured as `dependent`, delete operations will 'cascade' to those entities as well. By default entities in associated tables are removed using *Cake\ORM\Table::deleteAll()*. You can elect to have the ORM load related entities, and delete them individually by setting the `cascadeCallbacks` option to `true`. A sample HasMany association with both these options enabled would be:

```
// In a Table's initialize method.
$this->hasMany('Comments', [
    'dependent' => true,
    'cascadeCallbacks' => true,
]);
```

Note: Setting `cascadeCallbacks` to `true`, results in considerably slower deletes when compared to bulk deletes. The cascadeCallbacks option should only be enabled when your application has important work handled by event listeners.

Bulk Deletes

`Cake\ORM\Table::deleteMany`(*$entities*, *$options = []*)

If you have an array of entities you want to delete you can use `deleteMany()` to delete them in a single transaction:

```
// Get a boolean indicating success
$success = $this->Articles->deleteMany($entities);

// Will throw a PersistenceFailedException if any entity cannot be deleted.
$this->Articles->deleteManyOrFail($entities);
```

The `$options` for these methods are the same as `delete()`. Deleting records with these method **will** trigger events.

`Cake\ORM\Table::deleteAll`(*$conditions*)

There may be times when deleting rows one by one is not efficient or useful. In these cases it is more performant to use a bulk-delete to remove many rows at once:

```
// Delete all the spam
function destroySpam()
{
    return $this->deleteAll(['is_spam' => true]);
}
```

A bulk-delete will be considered successful if 1 or more rows are deleted. The function returns the number of deleted records as an integer.

> **Warning:** deleteAll will *not* trigger beforeDelete/afterDelete events. If you need callbacks triggered, first load the entities with `find()` and delete them in a loop.

Strict Deletes

`Cake\ORM\Table::deleteOrFail`(*$entity*, *$options = []*)

Using this method will throw an *Cake\ORM\Exception\PersistenceFailedException* if:

- the entity is new
- the entity has no primary key value
- application rules checks failed
- the delete was aborted by a callback.

If you want to track down the entity that failed to delete, you can use the `Cake\ORMException\PersistenceFailedException::getEntity()` method:

```
try {
    $table->deleteOrFail($entity);
} catch (\Cake\ORM\Exception\PersistenceFailedException $e) {
    echo $e->getEntity();
}
```

As this internally performs a *Cake\ORM\Table::delete()* call, all corresponding delete events will be triggered.

Behaviors

Behaviors are a way to organize and enable horizontal re-use of Model layer logic. Conceptually they are similar to traits. However, behaviors are implemented as separate classes. This allows them to hook into the life-cycle callbacks that models emit, while providing trait-like features.

Behaviors provide a convenient way to package up behavior that is common across many models. For example, CakePHP includes a `TimestampBehavior`. Many models will want timestamp fields, and the logic to manage these fields is not specific to any one model. It is these kinds of scenarios that behaviors are a perfect fit for.

Using Behaviors

Behaviors provide a way to create horizontally re-usable pieces of logic related to table classes. You may be wondering why behaviors are regular classes and not traits. The primary reason for this is event listeners. While traits would allow for re-usable pieces of logic, they would complicate binding events.

To add a behavior to your table you can call the `addBehavior()` method. Generally the best place to do this is in the `initialize()` method:

```
namespace App\Model\Table;

use Cake\ORM\Table;

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

As with associations, you can use *plugin syntax* and provide additional configuration options:

```
namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{
    public function initialize(array $config): void
    {
        $this->addBehavior('Timestamp', [
            'events' => [
                'Model.beforeSave' => [
                    'created_at' => 'new',
                    'modified_at' => 'always'
                ]
            ]
        ]);
    }
}
```

Core Behaviors

CounterCache

class Cake\ORM\Behavior**CounterCacheBehavior**

Often times web applications need to display counts of related objects. For example, when showing a list of articles you may want to display how many comments it has. Or when showing a user you might want to show how many friends/followers she has. The CounterCache behavior is intended for these situations. CounterCache will update a field in the associated models assigned in the options when it is invoked. The fields should exist in the database and be of the type INT.

Basic Usage

You enable the CounterCache behavior like any other behavior, but it won't do anything until you configure some relations and the field counts that should be stored on each of them. Using our example below, we could cache the comment count for each article with the following:

```
class CommentsTable extends Table
{
    public function initialize(array $config): void
    {
        $this->addBehavior('CounterCache', [
            'Articles' => ['comment_count']
        ]);
    }
}
```

Note: The column `comment_count` should exist in the `articles` table.

The CounterCache configuration should be a map of relation names and the specific configuration for that relation.

As you see you need to add the behavior on the "other side" of the association where you actually want the field to be updated. In this example the behavior is added to the `CommentsTable` even though it updates the `comment_count` field in the `ArticlesTable`.

The counter's value will be updated each time an entity is saved or deleted. The counter **will not** be updated when you

- save the entity without changing data or
- use `updateAll()` or
- use `deleteAll()` or
- execute SQL you have written

Advanced Usage

If you need to keep a cached counter for less than all of the related records, you can supply additional conditions or finder methods to generate a counter value:

```
// Use a specific find method.
// In this case find(published)
$this->addBehavior('CounterCache', [
    'Articles' => [
        'comment_count' => [
            'finder' => 'published',
        ],
    ],
]);
```

If you don't have a custom finder method you can provide an array of conditions to find records instead:

```
$this->addBehavior('CounterCache', [
    'Articles' => [
        'comment_count' => [
            'conditions' => ['Comments.spam' => false],
        ],
    ],
]);
```

If you want CounterCache to update multiple fields, for example both showing a conditional count and a basic count you can add these fields in the array:

```
$this->addBehavior('CounterCache', [
    'Articles' => ['comment_count',
        'published_comment_count' => [
            'finder' => 'published',
        ],
    ],
]);
```

If you want to calculate the CounterCache field value on your own, you can set the `ignoreDirty` option to `true`. This will prevent the field from being recalculated if you've set it dirty before:

```
$this->addBehavior('CounterCache', [
    'Articles' => [
        'comment_count' => [
            'ignoreDirty' => true,
        ],
    ],
]);
```

Lastly, if a custom finder and conditions are not suitable you can provide a callback function. Your function must return the count value to be stored:

```
$this->addBehavior('CounterCache', [
    'Articles' => [
        'rating_avg' => function ($event, $entity, $table, $original) {
            return 4.5;
```

(continues on next page)

(continued from previous page)

```
        }
    ],
]);
```

Your function can return `false` to skip updating the counter column, or a `SelectQuery` object that produced the count value. If you return a `SelectQuery` object, your query will be used as a subquery in the update statement. The `$table` parameter refers to the table object holding the behavior (not the target relation) for convenience. The callback is invoked at least once with `$original` set to `false`. If the entity-update changes the association then the callback is invoked a *second* time with `true`, the return value then updates the counter of the *previously* associated item.

Note: The CounterCache behavior works for `belongsTo` associations only. For example for "Comments belongsTo Articles", you need to add the CounterCache behavior to the `CommentsTable` in order to generate `comment_count` for Articles table.

Belongs to many Usage

It is possible to use the CounterCache behavior in a `belongsToMany` association. First, you need to add the `through` and `cascadeCallbacks` options to the `belongsToMany` association:

```
'through'          => 'CommentsArticles',
'cascadeCallbacks' => true
```

Also see *Using the 'through' Option* how to configure a custom join table.

The `CommentsArticles` is the name of the junction table classname. If you don't have it you should create it with the bake CLI tool.

In this `src/Model/Table/CommentsArticlesTable.php` you then need to add the behavior with the same code as described above.:

```
$this->addBehavior('CounterCache', [
    'Articles' => ['comments_count'],
]);
```

Finally clear all caches with `bin/cake cache clear_all` and try it out.

Timestamp

class Cake\ORM\Behavior**TimestampBehavior**

The timestamp behavior allows your table objects to update one or more timestamps on each model event. This is primarily used to populate data into `created` and `modified` fields. However, with some additional configuration, you can update any timestamp/datetime column on any event a table publishes.

Basic Usage

You enable the timestamp behavior like any other behavior:

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

The default configuration will do the following:

- When a new entity is saved the `created` and `modified` fields will be set to the current time.
- When an entity is updated, the `modified` field is set to the current time.

Using and Configuring the Behavior

If you need to modify fields with different names, or want to update additional timestamp fields on custom events you can use some additional configuration:

```
class OrdersTable extends Table
{
    public function initialize(array $config): void
    {
        $this->addBehavior('Timestamp', [
            'events' => [
                'Model.beforeSave' => [
                    'created_at' => 'new',
                    'updated_at' => 'always',
                ],
                'Orders.completed' => [
                    'completed_at' => 'always'
                ]
            ]
        ]);
    }
}
```

As you can see above, in addition to the standard `Model.beforeSave` event, we are also updating the `completed_at` column when orders are completed.

Updating Timestamps on Entities

Sometimes you'll want to update just the timestamps on an entity without changing any other properties. This is sometimes referred to as 'touching' a record. In CakePHP you can use the `touch()` method to do exactly this:

```
// Touch based on the Model.beforeSave event.
$articles->touch($article);

// Touch based on a specific event.
$orders->touch($order, 'Orders.completed');
```

After you have saved the entity, the field is updated.

Touching records can be useful when you want to signal that a parent resource has changed when a child resource is created/updated. For example: updating an article when a new comment is added.

Saving Updates Without Modifying Timestamps

To disable the automatic modification of the `updated` timestamp column when saving an entity you can mark the attribute as 'dirty':

```
// Mark the modified column as dirty making
// the current value be set on update.
$order->setDirty('modified', true);
```

Translate

class Cake\ORM\Behavior**TranslateBehavior**

The Translate behavior allows you to create and retrieve translated copies of your entities in multiple languages.

> **Warning:** The TranslateBehavior does not support composite primary keys at this point in time.

Translation Strategies

The behavior offers two strategies for how the translations are stored.

1. Shadow table Strategy: This strategy uses a separate "shadow table" for each Table object to store translation of all translated fields of that table. This is the default strategy.
2. Eav Strategy: This strategy uses a `i18n` table where it stores the translation for each of the fields of any given Table object that it's bound to.

Shadow Table Strategy

Let's assume we have an `articles` table and we want it's `title` and `body` fields to be translated. For that we create a shadow table `articles_translations`:

```
CREATE TABLE `articles_translations` (
    `id` int(11) NOT NULL,
    `locale` varchar(5) NOT NULL,
    `title` varchar(255),
    `body` text,
    PRIMARY KEY (`id`,`locale`)
);
```

The shadow table needs `id` and `locale` columns which together form the primary key and other columns with same name as primary table which need to be translated.

A note on language abbreviations: The Translate Behavior doesn't impose any restrictions on the language identifier, the possible values are only restricted by the `locale` column type/size. `locale` is defined as `varchar(6)` in case you want to use abbreviations like `es-419` (Spanish for Latin America, language abbreviation with area code UN M.49[128]).

Tip: It's wise to use the same language abbreviations as required for *Internationalization and Localization*. Thus you are consistent and switching the language works identical for both, the `Translate` behavior and `Internationalization and Localization`.

So it's recommended to use either the two letter ISO code of the language like `en`, `fr`, `de` or the full locale name such as `fr_FR`, `es_AR`, `da_DK` which contains both the language and the country where it is spoken.

Eav Strategy

In order to use the Eav strategy, you need to create a `i18n` table with the correct schema. Currently the only way of loading the `i18n` table is by manually running the following SQL script in your database:

```
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)
);
```

The schema is also available as sql file in **/config/schema/i18n.sql**.

[128] https://en.wikipedia.org/wiki/UN_M.49

Attaching the Translate Behavior to Your Tables

Attaching the behavior can be done in the `initialize()` method in your Table class:

```
class ArticlesTable extends Table
{
    public function initialize(array $config): void
    {
        // By default ShadowTable will be used.
        $this->addBehavior('Translate', ['fields' => ['title', 'body']]);
    }
}
```

For shadow table strategy specifying the `fields` key is optional as the behavior can infer the fields from the shadow table columns.

If you want to use the `EavStrategy` then you can configure the behavior as:

```
class ArticlesTable extends Table
{
    public function initialize(array $config): void
    {
        $this->addBehavior('Translate', [
            'strategyClass' => \Cake\ORM\Behavior\Translate\EavStrategy::class,
            'fields' => ['title', 'body'],
        ]);
    }
}
```

For `EavStrategy` you are required to pass the `fields` key in the configuration array. This list of fields is needed to tell the behavior what columns will be able to store translations.

By default the locale specified in `App.defaultLocale` config is used as default locale for the `TranslateBehavior`. You can override that by setting `defaultLocale` config of the behavior:

```
class ArticlesTable extends Table
{
    public function initialize(array $config): void
    {
        $this->addBehavior('Translate', [
            'defaultLocale' => 'en_GB',
        ]);
    }
}
```

Quick tour

Regardless of the datastructure strategy you choose the behavior provides the same API to manage translations.

Now, select a language to be used for retrieving entities by changing the application language, which will affect all translations:

```
// In the Articles controller. Change the locale to Spanish, for example
I18n::setLocale('es');
```

Then, get an existing entity:

```
$article = $this->Articles->get(12);
echo $article->title; // Echoes 'A title', not translated yet
```

Next, translate your entity:

```
$article->title = 'Un Artículo';
$this->Articles->save($article);
```

You can try now getting your entity again:

```
$article = $this->Articles->get(12);
echo $article->title; // Echoes 'Un Artículo', yay piece of cake!
```

Working with multiple translations can be done by using a special trait in your Entity class:

```
use Cake\ORM\Behavior\Translate\TranslateTrait;
use Cake\ORM\Entity;

class Article extends Entity
{
    use TranslateTrait;
}
```

Now you can find all translations for a single entity:

```
$article = $this->Articles->find('translations')->first();
echo $article->translation('es')->title; // 'Un Artículo'

echo $article->translation('en')->title; // 'An Article';
```

And save multiple translations at once:

```
$article->translation('es')->title = 'Otro Título';
$article->translation('fr')->title = 'Un autre Titre';
$this->Articles->save($article);
```

If you want to go deeper on how it works or how to tune the behavior for your needs, keep on reading the rest of this chapter.

Using a Separate Translations Table for Eav strategy

If you wish to use a table other than `i18n` for translating a particular repository, you can specify the name of the table class name for your custom table in the behavior's configuration. This is common when you have multiple tables to translate and you want a cleaner separation of the data that is stored for each different table:

```
class ArticlesTable extends Table
{
    public function initialize(array $config): void
    {
        $this->addBehavior('Translate', [
            'fields' => ['title', 'body'],
            'translationTable' => 'ArticlesI18n',
        ]);
    }
}
```

You need to make sure that any custom table you use has the columns `field`, `foreign_key`, `locale` and `model`.

Reading Translated Content

As shown above you can use the `setLocale()` method to choose the active translation for entities that are loaded:

```
// Load I18n core functions at the beginning of your Articles Controller:
use Cake\I18n\I18n;

// Then you can change the language in your action:
I18n::setLocale('es');

// All entities in results will contain spanish translation
$results = $this->Articles->find()->all();
```

This method works with any finder in your tables. For example, you can use TranslateBehavior with `find('list')`:

```
I18n::setLocale('es');
$data = $this->Articles->find('list')->toArray();

// Data will contain
[1 => 'Mi primer artículo', 2 => 'El segundo artículo', 15 => 'Otro articulo' ...]

// Change the locale to french for a single find call
$data = $this->Articles->find('list', locale: 'fr')->toArray();
```

Retrieve All Translations For An Entity

When building interfaces for updating translated content, it is often helpful to show one or more translation(s) at the same time. You can use the `translations` finder for this:

```
// Find the first article with all corresponding translations
$article = $this->Articles->find('translations')->first();
```

In the example above you will get a list of entities back that have a `_translations` property set. This property will contain a list of translation data entities. For example the following properties would be accessible:

```
// Outputs 'en'
echo $article->_translations['en']->locale;

// Outputs 'title'
echo $article->_translations['en']->field;

// Outputs 'My awesome post!'
echo $article->_translations['en']->body;
```

A more elegant way for dealing with this data is by adding a trait to the entity class that is used for your table:

```
use Cake\ORM\Behavior\Translate\TranslateTrait;
use Cake\ORM\Entity;

class Article extends Entity
{
    use TranslateTrait;
}
```

This trait contains a single method called `translation`, which lets you access or create new translation entities on the fly:

```
// Outputs 'title'
echo $article->translation('en')->title;

// Adds a new translation data entity to the article
$article->translation('de')->title = 'Wunderbar';
```

Limiting the Translations to be Retrieved

You can limit the languages that are fetched from the database for a particular set of records:

```
$results = $this->Articles->find('translations', locales: ['en', 'es']);
$article = $results->first();
$spanishTranslation = $article->translation('es');
$englishTranslation = $article->translation('en');
```

Preventing Retrieval of Empty Translations

Translation records can contain any string, if a record has been translated and stored as an empty string (‘’) the translate behavior will take and use this to overwrite the original field value.

If this is undesired, you can ignore translations which are empty using the `allowEmptyTranslations` config key:

```
class ArticlesTable extends Table
{
    public function initialize(array $config): void
    {
        $this->addBehavior('Translate', [
            'fields' => ['title', 'body'],
            'allowEmptyTranslations' => false
        ]);
    }
}
```

The above would only load translated data that had content.

Retrieving All Translations For Associations

It is also possible to find translations for any association in a single find operation:

```
$article = $this->Articles->find('translations')->contain([
    'Categories' => function ($query) {
        return $query->find('translations');
    }
])->first();

// Outputs 'Programación'
echo $article->categories[0]->translation('es')->name;
```

This assumes that `Categories` has the TranslateBehavior attached to it. It simply uses the query builder function for the `contain` clause to use the `translations` custom finder in the association.

Retrieving one language without using I18n::setLocale

calling `I18n::setLocale('es');` changes the default locale for all translated finds, there may be times you wish to retrieve translated content without modifying the application’s state. For these scenarios use the behavior’s `setLocale()` method:

```
I18n::setLocale('en'); // reset for illustration

// specific locale.
$this->Articles->setLocale('es');

$article = $this->Articles->get(12);
echo $article->title; // Echoes 'Un Artículo', yay piece of cake!
```

Note that this only changes the locale of the Articles table, it would not affect the language of associated data. To affect associated data it’s necessary to call the method on each table, for example:

```
I18n::setLocale('en'); // reset for illustration

$this->Articles->setLocale('es');
$this->Articles->Categories->setLocale('es');

$data = $this->Articles->find('all', contain: ['Categories']);
```

This example also assumes that `Categories` has the TranslateBehavior attached to it.

Querying Translated Fields

TranslateBehavior does not substitute find conditions by default. You need to use `translationField()` method to compose find conditions on translated fields:

```
$this->Articles->setLocale('es');
$query = $this->Articles->find()->where([
    $this->Articles->translationField('title') => 'Otro Título'
]);
```

Saving in Another Language

The philosophy behind the TranslateBehavior is that you have an entity representing the default language, and multiple translations that can override certain fields in such entity. Keeping this in mind, you can intuitively save translations for any given entity. For example, given the following setup:

```
// in src/Model/Table/ArticlesTable.php
class ArticlesTable extends Table
{
    public function initialize(array $config): void
    {
        $this->addBehavior('Translate', ['fields' => ['title', 'body']]);
    }
}

// in src/Model/Entity/Article.php
class Article extends Entity
{
    use TranslateTrait;
}

// In the Articles Controller
$article = new Article([
    'title' => 'My First Article',
    'body' => 'This is the content',
    'footnote' => 'Some afterwords'
]);

$this->Articles->save($article);
```

So, after you save your first article, you can now save a translation for it, there are a couple ways to do it. The first one is setting the language directly into the entity:

```
$article->_locale = 'es';
$article->title = 'Mi primer Artículo';

$this->Articles->save($article);
```

After the entity has been saved, the translated field will be persisted as well, one thing to note is that values from the default language that were not overridden will be preserved:

```
// Outputs 'This is the content'
echo $article->body;

// Outputs 'Mi primer Artículo'
echo $article->title;
```

Once you override the value, the translation for that field will be saved and can be retrieved as usual:

```
$article->body = 'El contendio';
$this->Articles->save($article);
```

The second way to use for saving entities in another language is to set the default language directly to the table:

```
$article->title = 'Mi Primer Artículo';

$this->Articles->setLocale('es');
$this->Articles->save($article);
```

Setting the language directly in the table is useful when you need to both retrieve and save entities for the same language or when you need to save multiple entities at once.

Saving Multiple Translations

It is a common requirement to be able to add or edit multiple translations to any database record at the same time. This can be done using the `TranslateTrait`:

```
use Cake\ORM\Behavior\Translate\TranslateTrait;
use Cake\ORM\Entity;

class Article extends Entity
{
    use TranslateTrait;
}
```

Now, You can populate translations before saving them:

```
$translations = [
    'fr' => ['title' => "Un article"],
    'es' => ['title' => 'Un artículo'],
];

foreach ($translations as $lang => $data) {
    $article->translation($lang)->set($data, ['guard' => false]);
}
```

(continues on next page)

(continued from previous page)

```
$this->Articles->save($article);
```

And create form controls for your translated fields:

```
// In a view template.
<?= $this->Form->create($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>
```

In your controller, you can marshal the data as normal:

```
$article = $this->Articles->newEntity($this->request->getData());
$this->Articles->save($article);
```

This will result in your article, the french and spanish translations all being persisted. You'll need to remember to add `_translations` into the `$_accessible` fields of your entity as well.

Validating Translated Entities

When attaching `TranslateBehavior` to a model, you can define the validator that should be used when translation records are created/modified by the behavior during `newEntity()` or `patchEntity()`:

```
class ArticlesTable extends Table
{
    public function initialize(array $config): void
    {
        $this->addBehavior('Translate', [
            'fields' => ['title'],
            'validator' => 'translated',
        ]);
    }
}
```

The above will use the validator created by `validationTranslated` to validated translated entities.

Tree

class Cake\ORM\Behavior**TreeBehavior**

It's fairly common to want to store hierarchical data in a database table. Examples of such data might be categories with unlimited subcategories, data related to a multilevel menu system or a literal representation of hierarchy such as departments in a company.

Relational databases are usually not well suited for storing and retrieving this type of data, but there are a few known techniques that can make them effective for working with multi-level information.

The TreeBehavior helps you maintain a hierarchical data structure in the database that can be queried without much overhead and helps reconstruct the tree data for finding and displaying processes.

Requirements

This behavior requires the following columns in your table:

- `parent_id` (nullable) The column holding the ID of the parent row. This column should be indexed.
- `lft` (integer, signed) Used to maintain the tree structure. This column should be indexed.
- `rght` (integer, signed) Used to maintain the tree structure.

You can configure the name of those fields should you need to customize them. More information on the meaning of the fields and how they are used can be found in this article describing the MPTT logic[129]

> **Warning:** The TreeBehavior does not support composite primary keys at this point in time.

A Quick Tour

You enable the Tree behavior by adding it to the Table you want to store hierarchical data in:

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

Once added, you can let CakePHP build the internal structure if the table is already holding some rows:

```
// In a controller
$categories = $this->getTableLocator()->get('Categories');
$categories->recover();
```

You can verify it works by getting any row from the table and asking for the count of descendants it has:

```
$node = $categories->get(1);
echo $categories->childCount($node);
```

[129] https://www.sitepoint.com/hierarchical-data-database-2/

Getting direct descendents

Getting a flat list of the descendants for a node can be done with:

```
$descendants = $categories->find('children', for: 1);

foreach ($descendants as $category) {
    echo $category->name . "\n";
}
```

If you need to pass conditions you do so as per normal:

```
$descendants = $categories
    ->find('children', for: 1)
    ->where(['name LIKE' => '%Foo%'])
    ->all();

foreach ($descendants as $category) {
    echo $category->name . "\n";
}
```

If you instead need a threaded list, where children for each node are nested in a hierarchy, you can stack the 'threaded' finder:

```
$children = $categories
    ->find('children', for: 1)
    ->find('threaded')
    ->toArray();

foreach ($children as $child) {
    echo "{$child->name} has " . count($child->children) . " direct children";
}
```

While, if you're using custom `parent_id` you need to pass it in the 'threaded' finder option (i.e. `parentField`) .

Note: For more information on 'threaded' finder options see *Finding Threaded Data logic*

Getting formatted tree lists

Traversing threaded results usually requires recursive functions in, but if you only require a result set containing a single field from each level so you can display a list, in an HTML select for example, it is better to use the `treeList` finder:

```
$list = $categories->find('treeList')->toArray();

// In a CakePHP template file:
echo $this->Form->control('categories', ['options' => $list]);

// Or you can output it in plain text, for example in a CLI script
foreach ($list as $categoryName) {
    echo $categoryName . "\n";
}
```

The output will be similar to:

```
My Categories
_Fun
__Sport
___Surfing
___Skating
_Trips
__National
__International
```

The `treeList` finder takes a number of options:

- `keyPath`: A dot separated path to fetch the field to use for the array key, or a closure to return the key out of the provided row.
- `valuePath`: A dot separated path to fetch the field to use for the array value, or a closure to return the value out of the provided row.
- `spacer`: A string to be used as prefix for denoting the depth in the tree for each item

An example of all options in use is:

```
$query = $categories->find('treeList',
    keyPath: 'url',
    valuePath: 'id',
    spacer: ' '
);
```

An example using closure:

```
$query = $categories->find('treeList',
    valuePath: function($entity){
        return $entity->url . ' ' . $entity->id
    }
);
```

Finding a path or branch in the tree

One very common task is to find the tree path from a particular node to the root of the tree. This is useful, for example, for adding the breadcrumbs list for a menu structure:

```
$nodeId = 5;
$crumbs = $categories->find('path', for: $nodeId)->all();

foreach ($crumbs as $crumb) {
    echo $crumb->name . ' > ';
}
```

Trees constructed with the TreeBehavior cannot be sorted by any column other than `lft`, this is because the internal representation of the tree depends on this sorting. Luckily, you can reorder the nodes inside the same level without having to change their parent:

```
$node = $categories->get(5);
```

(continues on next page)

(continued from previous page)

```
// Move the node so it shows up one position up when listing children.
$categories->moveUp($node);

// Move the node to the top of the list inside the same level.
$categories->moveUp($node, true);

// Move the node to the bottom.
$categories->moveDown($node, true);
```

Configuration

If the default column names that are used by this behavior don't match your own schema, you can provide aliases for them:

```
public function initialize(array $config): void
{
    $this->addBehavior('Tree', [
        'parent' => 'ancestor_id', // Use this instead of parent_id
        'left' => 'tree_left', // Use this instead of lft
        'right' => 'tree_right' // Use this instead of rght
    ]);
}
```

Node Level (Depth)

Knowing the depth of tree nodes can be useful when you want to retrieve nodes only up to a certain level, for example, when generating menus. You can use the `level` option to specify the field that will save level of each node:

```
$this->addBehavior('Tree', [
    'level' => 'level', // Defaults to null, i.e. no level saving
]);
```

If you don't want to cache the level using a db field you can use `TreeBehavior::getLevel()` method to get level of a node.

Scoping and Multi Trees

Sometimes you want to persist more than one tree structure inside the same table, you can achieve that by using the 'scope' configuration. For example, in a locations table you may want to create one tree per country:

```
class LocationsTable extends Table
{
    public function initialize(array $config): void
    {
        $this->addBehavior('Tree', [
            'scope' => ['country_name' => 'Brazil']
        ]);
    }
}
```

In the previous example, all tree operations will be scoped to only the rows having the column `country_name` set to 'Brazil'. You can change the scoping on the fly by using the 'config' function:

```
$this->behaviors()->Tree->setConfig('scope', ['country_name' => 'France']);
```

Optionally, you can have a finer grain control of the scope by passing a closure as the scope:

```
$this->behaviors()->Tree->setConfig('scope', function ($query) {
    $country = $this->getConfigureContry(); // A made-up function
    return $query->where(['country_name' => $country]);
});
```

Deletion Behavior

By enabling the `cascadeCallbacks` option, `TreeBehavior` will load all of the entities that are going to be deleted. Once loaded, these entities will be deleted individually using `Table::delete()`. This enables ORM callbacks to be fired when tree nodes are deleted:

```
$this->addBehavior('Tree', [
    'cascadeCallbacks' => true,
]);
```

Recovering with custom sort field

By default, `recover()` sorts the items using the primary key. This works great if this is a numeric (auto increment) column, but can lead to weird results if you use UUIDs.

If you need custom sorting for the recovery, you can set a custom order clause in your config:

```
$this->addBehavior('Tree', [
    'recoverOrder' => ['country_name' => 'DESC'],
]);
```

Saving Hierarchical Data

When using the Tree behavior, you usually don't need to worry about the internal representation of the hierarchical structure. The positions where nodes are placed in the tree are deduced from the `parent_id` column in each of your entities:

```
$aCategory = $categoriesTable->get(10);
$aCategory->parent_id = 5;
$categoriesTable->save($aCategory);
```

Providing inexistent parent ids when saving or attempting to create a loop in the tree (making a node child of itself) will throw an exception.

You can make a node into a root in the tree by setting the `parent_id` column to null:

```
$aCategory = $categoriesTable->get(10);
$aCategory->parent_id = null;
$categoriesTable->save($aCategory);
```

Children for the new root node will be preserved.

Deleting Nodes

Deleting a node and all its sub-tree (any children it may have at any depth in the tree) is trivial:

```
$aCategory = $categoriesTable->get(10);
$categoriesTable->delete($aCategory);
```

The TreeBehavior will take care of all internal deleting operations for you. It is also possible to only delete one node and re-assign all children to the immediately superior parent node in the tree:

```
$aCategory = $categoriesTable->get(10);
$categoriesTable->removeFromTree($aCategory);
$categoriesTable->delete($aCategory);
```

All children nodes will be kept and a new parent will be assigned to them.

The deletion of a node is based off of the `lft` and `rght` values of the entity. This is important to note when looping through the various children of a node for conditional deletes:

```
$descendants = $teams->find('children', for: 1)->all();

foreach ($descendants as $descendant) {
    $team = $teams->get($descendant->id); // search for the up-to-date entity object
    if ($team->expired) {
        $teams->delete($team); // deletion reorders the lft and rght of database entries
    }
}
```

TreeBehavior will reorder the `lft` and `rght` values of records in the table when a node is deleted.

In our example above, the `lft` and `rght` values of the entities inside `$descendants` will be inaccurate. You will need to reload existing entity objects if you need an accurate shape of the tree.

Creating a Behavior

In the following examples we will create a very simple `SluggableBehavior`. This behavior will allow us to populate a slug field with the results of `Text::slug()` based on another field.

Before we create our behavior we should understand the conventions for behaviors:

- Behavior files are located in **src/Model/Behavior**, or `MyPlugin\Model\Behavior`.
- Behavior classes should be in the `App\Model\Behavior` namespace, or `MyPlugin\Model\Behavior` namespace.
- Behavior class names end in `Behavior`.
- Behaviors extend `Cake\ORM\Behavior`.

To create our sluggable behavior. Put the following into **src/Model/Behavior/SluggableBehavior.php**:

```
namespace App\Model\Behavior;

use Cake\ORM\Behavior;
```

(continues on next page)

(continued from previous page)

```
class SluggableBehavior extends Behavior
{
}
```

Similar to tables, behaviors also have an `initialize()` hook where you can put your behavior's initialization code, if required:

```
public function initialize(array $config): void
{
    // Some initialization code here
}
```

We can now add this behavior to one of our table classes. In this example we'll use an `ArticlesTable`, as articles often have slug properties for creating friendly URLs:

```
namespace App\Model\Table;

use Cake\ORM\Table;

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

Our new behavior doesn't do much of anything right now. Next, we'll add a mixin method and an event listener so that when we save entities we can automatically slug a field.

Defining Mixin Methods

Any public method defined on a behavior will be added as a 'mixin' method on the table object it is attached to. If you attach two behaviors that provide the same methods an exception will be raised. If a behavior provides the same method as a table class, the behavior method will not be callable from the table. Behavior mixin methods will receive the exact same arguments that are provided to the table. For example, if our SluggableBehavior defined the following method:

```
public function slug($value)
{
    return Text::slug($value, $this->_config['replacement']);
}
```

It could be invoked using:

```
$slug = $articles->slug('My article name');
```

Limiting or Renaming Exposed Mixin Methods

When creating behaviors, there may be situations where you don't want to expose public methods as mixin methods. In these cases you can use the `implementedMethods` configuration key to rename or exclude mixin methods. For example if we wanted to prefix our slug() method we could do the following:

```
protected $_defaultConfig = [
    'implementedMethods' => [
        'superSlug' => 'slug',
    ]
];
```

Applying this configuration will make `slug()` not callable, however it will add a `superSlug()` mixin method to the table. Notably if our behavior implemented other public methods they would **not** be available as mixin methods with the above configuration.

Since the exposed methods are decided by configuration you can also rename/remove mixin methods when adding a behavior to a table. For example:

```
// In a table's initialize() method.
$this->addBehavior('Sluggable', [
    'implementedMethods' => [
        'superSlug' => 'slug',
    ]
]);
```

Defining Event Listeners

Now that our behavior has a mixin method to slug fields, we can implement a callback listener to automatically slug a field when entities are saved. We'll also modify our slug method to accept an entity instead of just a plain value. Our behavior should now look like:

```
namespace App\Model\Behavior;

use ArrayObject;
use Cake\Datasource\EntityInterface;
use Cake\Event\EventInterface;
use Cake\ORM\Behavior;
use Cake\ORM\Entity;
use Cake\ORM\Query\SelectQuery;
use Cake\Utility\Text;

class SluggableBehavior extends Behavior
{
    protected array $_defaultConfig = [
        'field' => 'title',
        'slug' => 'slug',
        'replacement' => '-',
    ];

    public function slug(EntityInterface $entity)
    {
```

(continues on next page)

(continued from previous page)

```
        $config = $this->getConfig();
        $value = $entity->get($config['field']);
        $entity->set($config['slug'], Text::slug($value, $config['replacement']));
    }

    public function beforeSave(EventInterface $event, EntityInterface $entity,␣
↪ArrayObject $options)
    {
        $this->slug($entity);
    }

}
```

The above code shows a few interesting features of behaviors:

- Behaviors can define callback methods by defining methods that follow the *Lifecycle Callbacks* conventions.
- Behaviors can define a default configuration property. This property is merged with the overrides when a behavior is attached to the table.

To prevent the save from continuing, simply stop event propagation in your callback:

```
public function beforeSave(EventInterface $event, EntityInterface $entity, ArrayObject
↪$options)
{
    if (...) {
        $event->stopPropagation();
        $event->setResult(false);
        return;
    }
    $this->slug($entity);
}
```

Alternatively, you can return false from the callback. This has the same effect as stopping event propagation.

Defining Finders

Now that we are able to save articles with slug values, we should implement a finder method so we can fetch articles by their slug. Behavior finder methods, use the same conventions as *Custom Finder Methods* do. Our `find('slug')` method would look like:

```
public function findSlug(SelectQuery $query, string $slug): SelectQuery
{
    return $query->where(['slug' => $slug]);
}
```

Once our behavior has the above method we can call it:

```
$article = $articles->find('slug', slug: $value)->first();
```

Limiting or Renaming Exposed Finder Methods

When creating behaviors, there may be situations where you don't want to expose finder methods, or you need to rename finders to avoid duplicated methods. In these cases you can use the `implementedFinders` configuration key to rename or exclude finder methods. For example if we wanted to rename our `find(slug)` method we could do the following:

```
protected array $_defaultConfig = [
    'implementedFinders' => [
        'slugged' => 'findSlug',
    ]
];
```

Applying this configuration will make `find('slug')` trigger an error. However it will make `find('slugged')` available. Notably if our behavior implemented other finder methods they would **not** be available, as they are not included in the configuration.

Since the exposed methods are decided by configuration you can also rename/remove finder methods when adding a behavior to a table. For example:

```
// In a table's initialize() method.
$this->addBehavior('Sluggable', [
    'implementedFinders' => [
        'slugged' => 'findSlug',
    ]
]);
```

Transforming Request Data into Entity Properties

Behaviors can define logic for how the custom fields they provide are marshalled by implementing the `Cake\ORM\PropertyMarshalInterface`. This interface requires a single method to be implemented:

```
public function buildMarshalMap($marshaller, $map, $options)
{
    return [
        'custom_behavior_field' => function ($value, $entity) {
            // Transform the value as necessary
            return $value . '123';
        }
    ];
}
```

The `TranslateBehavior` has a non-trivial implementation of this interface that you might want to refer to.

Removing Loaded Behaviors

To remove a behavior from your table you can call the `removeBehavior()` method:

```
// Remove the loaded behavior
$this->removeBehavior('Sluggable');
```

Accessing Loaded Behaviors

Once you've attached behaviors to your Table instance you can introspect the loaded behaviors, or access specific behaviors using the `BehaviorRegistry`:

```
// See which behaviors are loaded
$table->behaviors()->loaded();

// Check if a specific behavior is loaded.
// Remember to omit plugin prefixes.
$table->behaviors()->has('CounterCache');

// Get a loaded behavior
// Remember to omit plugin prefixes
$table->behaviors()->get('CounterCache');
```

Re-configuring Loaded Behaviors

To modify the configuration of an already loaded behavior you can combine the `BehaviorRegistry::get` command with `config` command provided by the `InstanceConfigTrait` trait.

For example, if a parent class, such as `AppTable`, loaded the `Timestamp` behavior you could do the following to add, modify or remove the configurations for the behavior. In this case, we will add an event we want Timestamp to respond to:

```
namespace App\Model\Table;

use App\Model\Table\AppTable; // similar to AppController

class UsersTable extends AppTable
{
    public function initialize(array $options): void
    {
        parent::initialize($options);

        // For example, if our parent calls $this->addBehavior('Timestamp')
        // and we want to add an additional event
        if ($this->behaviors()->has('Timestamp')) {
            $this->behaviors()->get('Timestamp')->setConfig([
                'events' => [
                    'Users.login' => [
                        'last_login' => 'always'
                    ],
                ],
            ]);
        }
    }
}
```

Schema System

CakePHP features a schema system that is capable of reflecting and generating schema information for tables in SQL datastores. The schema system can generate/reflect a schema for any SQL platform that CakePHP supports.

The main pieces of the schema system are `Cake\Database\Schema\Collection` and `Cake\Database\Schema\TableSchema`. These classes give you access to database-wide and individual Table object features respectively.

The primary use of the schema system is for *Fixtures*. However, it can also be used in your application if required.

Schema\TableSchema Objects

class Cake\Database\Schema**TableSchema**

The schema subsystem provides a simple TableSchema object to hold data about a table in a database. This object is returned by the schema reflection features:

```
use Cake\Database\Schema\TableSchema;

// Create a table one column at a time.
$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']
]);

// Schema\TableSchema classes could also be created with array data
$schema = new TableSchema('posts', $columns);
```

`Schema\TableSchema` objects allow you to build up information about a table's schema. It helps to normalize and validate the data used to describe a table. For example, the following two forms are equivalent:

```
$schema->addColumn('title', 'string');
// and
$schema->addColumn('title', [
  'type' => 'string'
]);
```

While equivalent, the 2nd form allows more detail and control. This emulates the existing features available in Schema files + the fixture schema in 2.x.

Accessing Column Data

Columns are either added as constructor arguments, or via *addColumn()*. Once fields are added information can be fetched using *column()* or *columns()*:

```
// Get the array of data about a column
$c = $schema->column('title');

// Get the list of all columns.
$cols = $schema->columns();
```

Indexes and Constraints

Indexes are added using the `addIndex()`. Constraints are added using `addConstraint()`. Indexes and constraints cannot be added for columns that do not exist, as it would result in an invalid state. Indexes are different from constraints, and exceptions will be raised if you try to mix types between the methods. An example of both methods is:

```
$schema = new TableSchema('posts');
$schema->addColumn('id', 'integer')
  ->addColumn('author_id', 'integer')
  ->addColumn('title', 'string')
  ->addColumn('slug', 'string');

// Add a primary key.
$schema->addConstraint('primary', [
  'type' => 'primary',
  'columns' => ['id']
]);
// Add a unique key
$schema->addConstraint('slug_idx', [
  'columns' => ['slug'],
  'type' => 'unique',
]);
// Add index
$schema->addIndex('slug_title', [
  'columns' => ['slug', 'title'],
  'type' => 'index'
]);
// Add a foreign key
$schema->addConstraint('author_id_idx', [
  'columns' => ['author_id'],
  'type' => 'foreign',
  'references' => ['authors', 'id'],
  'update' => 'cascade',
  'delete' => 'cascade'
]);
```

If you add a primary key constraint to a single integer column it will automatically be converted into a auto-increment/serial column depending on the database platform:

```
$schema = new TableSchema('posts');
$schema->addColumn('id', 'integer')
```

(continues on next page)

(continued from previous page)

```
->addConstraint('primary', [
    'type' => 'primary',
    'columns' => ['id']
]);
```

In the above example the `id` column would generate the following SQL in MySQL:

```
CREATE TABLE `posts` (
    `id` INTEGER AUTO_INCREMENT,
    PRIMARY KEY (`id`)
)
```

If your primary key contains more than one column, none of them will automatically be converted to an auto-increment value. Instead you will need to tell the table object which column in the composite key you want to auto-increment:

```
$schema = new TableSchema('posts');
$schema->addColumn('id', [
    'type' => 'integer',
    'autoIncrement' => true,
])
->addColumn('account_id', 'integer')
->addConstraint('primary', [
    'type' => 'primary',
    'columns' => ['id', 'account_id']
]);
```

The `autoIncrement` option only works with `integer` and `biginteger` columns.

Reading Indexes and Constraints

Indexes and constraints can be read out of a table object using accessor methods. Assuming that `$schema` is a populated TableSchema instance you could do the following:

```
// Get constraints. Will return the
// names of all constraints.
$constraints = $schema->constraints()

// Get data about a single constraint.
$constraint = $schema->constraint('author_id_idx')

// Get indexes. Will return the
// names of all indexes.
$indexes = $schema->indexes()

// Get data about a single index.
$index = $schema->index('author_id_idx')
```

Adding Table Options

Some drivers (primarily MySQL) support and require additional table metadata. In the case of MySQL the `CHARSET`, `COLLATE` and `ENGINE` properties are required for maintaining a table's structure in MySQL. The following could be used to add table options:

```
$schema->options([
  'engine' => 'InnoDB',
  'collate' => 'utf8_unicode_ci',
]);
```

Platform dialects only handle the keys they are interested in and ignore the rest. Not all options are supported on all platforms.

Converting Tables into SQL

Using the `createSql()` or `dropSql()` you can get platform specific SQL for creating or dropping a specific table:

```
$db = ConnectionManager::get('default');
$schema = new TableSchema('posts', $fields, $indexes);

// Create a table
$queries = $schema->createSql($db);
foreach ($queries as $sql) {
  $db->execute($sql);
}

// Drop a table
$sql = $schema->dropSql($db);
$db->execute($sql);
```

By using a connection's driver the schema data can be converted into platform specific SQL. The return of `createSql` and `dropSql` is a list of SQL queries required to create a table and the required indexes. Some platforms may require multiple statements to create tables with comments and/or indexes. An array of queries is always returned.

Schema Collections

class Cake\Database\Schema**Collection**

`Collection` provides access to the various tables available on a connection. You can use it to get the list of tables or reflect tables into *TableSchema* objects. Basic usage of the class looks like:

```
$db = ConnectionManager::get('default');

// Create a schema collection.
$collection = $db->getSchemaCollection();

// Get the table names
$tables = $collection->listTables();

// Get a single table (instance of Schema\TableSchema)
$tableSchema = $collection->describe('posts');
```

Schema Cache Tool

The SchemaCacheShell provides a simple CLI tool for managing your application's metadata caches. In deployment situations it is helpful to rebuild the metadata cache in-place without clearing the existing cache data. You can do this by running:

```
bin/cake schema_cache build --connection default
```

This will rebuild the metadata cache for all tables on the `default` connection. If you only need to rebuild a single table you can do that by providing its name:

```
bin/cake schema_cache build --connection default articles
```

In addition to building cached data, you can use the SchemaCacheShell to remove cached metadata as well:

```
# Clear all metadata
bin/cake schema_cache clear

# Clear a single table
bin/cake schema_cache clear articles
```

CHAPTER 13

Caching

class Cake\Cache**Cache**

Caching can be used to make reading from expensive or slow resources faster, by maintaining a second copy of the required data in a faster or closer storage system. For example, you can store the results of expensive queries, or remote webservice access that doesn't frequently change in a cache. Once in the cache, reading data from the cache is much cheaper than accessing the remote resource.

Caching in CakePHP is facilitated by the `Cache` class. This class provides a static interface and uniform API to interact with various Caching implementations. CakePHP provides several cache engines, and provides a simple interface if you need to build your own backend. The built-in caching engines are:

- `File` File cache is a simple cache that uses local files. It is the slowest cache engine, and doesn't provide as many features for atomic operations. However, since disk storage is often quite cheap, storing large objects, or elements that are infrequently written work well in files.
- `Memcached` Uses the Memcached[130] extension.
- `Redis` Uses the phpredis[131] extension. Redis provides a fast and persistent cache system similar to Memcached, also provides atomic operations.
- `Apcu` APCu cache uses the PHP APCu[132] extension. This extension uses shared memory on the webserver to store objects. This makes it very fast, and able to provide atomic read/write features.
- `Array` Stores all data in an array. This engine does not provide persistent storage and is intended for use in application test suites.
- `Null` The null engine doesn't actually store anything and fails all read operations.

Regardless of the CacheEngine you choose to use, your application interacts with *Cake\Cache\Cache*.

[130] https://php.net/memcached
[131] https://github.com/phpredis/phpredis
[132] https://php.net/apcu

Configuring Cache Engines

static Cake\Cache\Cache::**setConfig**(*$key*, *$config = null*)

Your application can configure any number of 'engines' during its bootstrap process. Cache engine configurations are defined in **config/app.php**.

For optimal performance CakePHP requires two cache engines to be defined.

- `_cake_core_` is used for storing file maps, and parsed results of *Internationalization & Localization* files.
- `_cake_model_`, is used to store schema descriptions for your applications models.

Using multiple engine configurations also lets you incrementally change the storage as needed. For example in your **config/app.php** you could put the following:

```
// ...
'Cache' => [
    'short' => [
        'className' => 'File',
        'duration' => '+1 hours',
        'path' => CACHE,
        'prefix' => 'cake_short_',
    ],
    // Using a fully namespaced name.
    'long' => [
        'className' => 'Cake\Cache\Engine\FileEngine',
        'duration' => '+1 week',
        'probability' => 100,
        'path' => CACHE . 'long' . DS,
    ],
]
// ...
```

Configuration options can also be provided as a *DSN* string. This is useful when working with environment variables or *PaaS* providers:

```
Cache::setConfig('short', [
    'url' => 'memcached://user:password@cache-host/?timeout=3600&prefix=myapp_',
]);
```

When using a DSN string you can define any additional parameters/options as query string arguments.

You can also configure Cache engines at runtime:

```
// Using a short name
Cache::setConfig('short', [
    'className' => 'File',
    'duration' => '+1 hours',
    'path' => CACHE,
    'prefix' => 'cake_short_'
]);

// Using a fully namespaced name.
Cache::setConfig('long', [
    'className' => 'Cake\Cache\Engine\FileEngine',
```

(continues on next page)

(continued from previous page)

```
    'duration' => '+1 week',
    'probability' => 100,
    'path' => CACHE . 'long' . DS,
]);

// Using a constructed object.
$object = new FileEngine($config);
Cache::setConfig('other', $object);
```

The name of these engine configurations ('short' and 'long') are used as the `$config` parameter for *Cake\Cache\Cache::write()* and *Cake\Cache\Cache::read()*. When configuring cache engines you can refer to the class name using the following syntaxes:

```
// Short name (in App\ or Cake namespaces)
Cache::setConfig('long', ['className' => 'File']);

// Plugin short name
Cache::setConfig('long', ['className' => 'MyPlugin.SuperCache']);

// Full namespace
Cache::setConfig('long', ['className' => 'Cake\Cache\Engine\FileEngine']);

// An object implementing CacheEngineInterface
Cache::setConfig('long', ['className' => $myCache]);
```

Note: When using the FileEngine you might need to use the `mask` option to ensure cache files are made with the correct permissions.

Engine Options

Each engine accepts the following options:

- `duration` Specify how long items in this cache configuration last. Specified as a `strtotime()` compatible expression.
- `groups` List of groups or 'tags' associated to every key stored in this config. Useful when you need to delete a subset of data from a cache.
- `prefix` Prepended to all entries. Good for when you need to share a keyspace with either another cache config or another application.
- `probability` Probability of hitting a cache gc cleanup. Setting to 0 will disable `Cache::gc()` from ever being called automatically.

FileEngine Options

FileEngine uses the following engine specific options:

- `isWindows` Automatically populated with whether the host is windows or not
- `lock` Should files be locked before writing to them?
- `mask` The mask used for created files
- `path` Path to where cachefiles should be saved. Defaults to system's temp dir.

RedisEngine Options

RedisEngine uses the following engine specific options:

- `port` The port your Redis server is running on.
- `host` The host your Redis server is running on.
- `database` The database number to use for connection.
- `password` Redis server password.
- `persistent` Should a persistent connection be made to Redis.
- `timeout` Connection timeout for Redis.
- `unix_socket` Path to a unix socket for Redis.

MemcacheEngine Options

- `compress` Whether to compress data.
- `username` Login to access the Memcache server.
- `password` Password to access the Memcache server.
- `persistent` The name of the persistent connection. All configurations using the same persistent value will share a single underlying connection.
- `serialize` The serializer engine used to serialize data. Available engines are php, igbinary and json. Beside php, the memcached extension must be compiled with the appropriate serializer support.
- `servers` String or array of memcached servers. If an array MemcacheEngine will use them as a pool.
- `duration` Be aware that any duration greater than 30 days will be treated as real Unix time value rather than an offset from current time.
- `options` Additional options for the memcached client. Should be an array of option => value. Use the \ `Memcached::OPT_*` constants as keys.

Configuring Cache Fallbacks

In the event that an engine is not available, such as the `FileEngine` trying to write to an unwritable folder or the `RedisEngine` failing to connect to Redis, the engine will fall back to the noop `NullEngine` and trigger a loggable error. This prevents the application from throwing an uncaught exception due to cache failure.

You can configure Cache configurations to fall back to a specified config using the `fallback` configuration key:

```
Cache::setConfig('redis', [
    'className' => 'Redis',
    'duration' => '+1 hours',
    'prefix' => 'cake_redis_',
    'host' => '127.0.0.1',
    'port' => 6379,
    'fallback' => 'default',
]);
```

If initializing the `RedisEngine` instance fails, the `redis` cache configuration would fall back to using the `default` cache configuration. If initializing the engine for the `default` cache configuration *also* fails, in this scenario the engine would fall back once again to the `NullEngine` and prevent the application from throwing an uncaught exception.

You can turn off cache fallbacks with `false`:

```
Cache::setConfig('redis', [
    'className' => 'Redis',
    'duration' => '+1 hours',
    'prefix' => 'cake_redis_',
    'host' => '127.0.0.1',
    'port' => 6379,
    'fallback' => false
]);
```

When there is no fallback cache failures will be raised as exceptions.

Removing Configured Cache Engines

static Cake\Cache\Cache::**drop**(*$key*)

Once a configuration is created you cannot change it. Instead you should drop the configuration and re-create it using *Cake\Cache\Cache::drop()* and *Cake\Cache\Cache::setConfig()*. Dropping a cache engine will remove the config and destroy the adapter if it was constructed.

Writing to a Cache

static Cake\Cache\Cache::**write**(*$key*, *$value*, *$config = 'default'*)

`Cache::write()` will write a $value to the Cache. You can read or delete this value later by referring to it by `$key`. You may specify an optional configuration to store the cache in as well. If no `$config` is specified, default will be used. `Cache::write()` can store any type of object and is ideal for storing results of model finds:

```
$posts = Cache::read('posts');
if ($posts === null) {
```

(continues on next page)

(continued from previous page)

```
    $posts = $someService->getAllPosts();
    Cache::write('posts', $posts);
}
```

Using `Cache::write()` and `Cache::read()` to reduce the number of trips made to the database to fetch posts.

Note: If you plan to cache the result of queries made with the CakePHP ORM, it is better to use the built-in cache capabilities of the Query object as described in the *Caching Loaded Results* section

---

Writing Multiple Keys at Once

static Cake\Cache\Cache::**writeMany**(*$data*, *$config = 'default'*)

You may find yourself needing to write multiple cache keys at once. While you can use multiple calls to `write()`, `writeMany()` allows CakePHP to use more efficient storage APIs where available. For example using `writeMany()` save multiple network connections when using Memcached:

```
$result = Cache::writeMany([
    'article-' . $slug => $article,
    'article-' . $slug . '-comments' => $comments
]);

// $result will contain
['article-first-post' => true, 'article-first-post-comments' => true]
```

Atomic writes

static Cake\Cache\Cache::**add**(*$key*, *$value $config = 'default'*)

Using `Cache::add()` will let you atomically set a key to a value if the key does not already exist in the cache. If the key already exists in the cache backend or the write fails, `add()` will return `false`:

```
// Set a key to act as a lock
$result = Cache::add($lockKey, true);
if (!$result) {
    return;
}
// Do an action where there can only be one process active at a time.

// Remove the lock key.
Cache::delete($lockKey);
```

Warning: File based caching does not support atomic writes.

Read Through Caching

static Cake\Cache\Cache::**remember**(*$key*, *$callable*, *$config = 'default'*)

Cache helps with read-through caching. If the named cache key exists, it will be returned. If the key does not exist, the callable will be invoked and the results stored in the cache at the provided key.

For example, you often want to cache remote service call results. You could use `remember()` to make this simple:

```
class IssueService
{
    public function allIssues($repo)
    {
        return Cache::remember($repo . '-issues', function () use ($repo) {
            return $this->fetchAll($repo);
        });
    }
}
```

Reading From a Cache

static Cake\Cache\Cache::**read**(*$key*, *$config = 'default'*)

`Cache::read()` is used to read the cached value stored under `$key` from the `$config`. If `$config` is null the default config will be used. `Cache::read()` will return the cached value if it is a valid cache or `null` if the cache has expired or doesn't exist. Use strict comparison operators === or !== to check the success of the `Cache::read()` operation.

For example:

```
$cloud = Cache::read('cloud');
if ($cloud !== null) {
    return $cloud;
}

// Generate cloud data
// ...

// Store data in cache
Cache::write('cloud', $cloud);

return $cloud;
```

Or if you are using another cache configuration called `short`, you can specify it in `Cache::read()` and `Cache::write()` calls as below:

```
// Read key "cloud", but from short configuration instead of default
$cloud = Cache::read('cloud', 'short');
if ($cloud === null) {
    // Generate cloud data
    // ...

    // Store data in cache, using short cache configuration instead of default
    Cache::write('cloud', $cloud, 'short');
```

(continues on next page)

(continued from previous page)

```
}

return $cloud;
```

Reading Multiple Keys at Once

static Cake\Cache\Cache::**readMany**(*$keys*, *$config = 'default'*)

After you've written multiple keys at once, you'll probably want to read them as well. While you could use multiple calls to `read()`, `readMany()` allows CakePHP to use more efficient storage APIs where available. For example using `readMany()` save multiple network connections when using Memcached:

```
$result = Cache::readMany([
    'article-' . $slug,
    'article-' . $slug . '-comments'
]);
// $result will contain
['article-first-post' => '...', 'article-first-post-comments' => '...']
```

Deleting From a Cache

static Cake\Cache\Cache::**delete**(*$key*, *$config = 'default'*)

`Cache::delete()` will allow you to completely remove a cached object from the store:

```
// Remove a key
Cache::delete('my_key');
```

As of 4.4.0, the `RedisEngine` also provides a `deleteAsync()` method which uses the `UNLINK` operation to remove cache keys:

```
Cache::pool('redis')->deleteAsync('my_key');
```

Deleting Multiple Keys at Once

static Cake\Cache\Cache::**deleteMany**(*$keys*, *$config = 'default'*)

After you've written multiple keys at once, you may want to delete them. While you could use multiple calls to `delete()`, `deleteMany()` allows CakePHP to use more efficient storage APIs where available. For example using `deleteMany()` save multiple network connections when using Memcached:

```
$result = Cache::deleteMany([
    'article-' . $slug,
    'article-' . $slug . '-comments'
]);
// $result will contain
['article-first-post' => true, 'article-first-post-comments' => true]
```

Clearing Cached Data

static Cake\Cache\Cache::**clear**(*$config = 'default'*)

Destroy all cached values for a cache configuration. In engines like: Apcu, Memcached, the cache configuration's prefix is used to remove cache entries. Make sure that different cache configurations have different prefixes:

```
// Will clear all keys.
Cache::clear();
```

As of 4.4.0, the `RedisEngine` also provides a `clearBlocking()` method which uses the `UNLINK` operation to remove cache keys:

```
Cache::pool('redis')->clearBlocking();
```

Note: Because APCu uses isolated caches for webserver and CLI they have to be cleared separately (CLI cannot clear webserver and vice versa).

Using Cache to Store Counters

static Cake\Cache\Cache::**increment**(*$key*, *$offset = 1*, *$config = 'default'*)

static Cake\Cache\Cache::**decrement**(*$key*, *$offset = 1*, *$config = 'default'*)

Counters in your application are good candidates for storage in a cache. As an example, a simple countdown for remaining 'slots' in a contest could be stored in Cache. The Cache class exposes atomic ways to increment/decrement counter values. Atomic operations are important for these values as it reduces the risk of contention, and ability for two users to simultaneously lower the value by one, resulting in an incorrect value.

After setting an integer value you can manipulate it using `increment()` and `decrement()`:

```
Cache::write('initial_count', 10);

// Later on
Cache::decrement('initial_count');

// Or
Cache::increment('initial_count');
```

Note: Incrementing and decrementing do not work with FileEngine. You should use APCu, Redis or Memcached instead.

Using Cache to Store Common Query Results

You can greatly improve the performance of your application by putting results that infrequently change, or that are subject to heavy reads into the cache. A perfect example of this are the results from *Cake\ORM\Table::find()*. The Query object allows you to cache results using the `cache()` method. See the *Caching Loaded Results* section for more information.

Using Groups

Sometimes you will want to mark multiple cache entries to belong to certain group or namespace. This is a common requirement for mass-invalidating keys whenever some information changes that is shared among all entries in the same group. This is possible by declaring the groups in cache configuration:

```
Cache::setConfig('site_home', [
    'className' => 'Redis',
    'duration' => '+999 days',
    'groups' => ['comment', 'article'],
]);
```

Cake\Cache\Cache::**clearGroup**(*$group*, *$config = 'default'*)

Let's say you want to store the HTML generated for your homepage in cache, but would also want to automatically invalidate this cache every time a comment or post is added to your database. By adding the groups `comment` and `article`, we have effectively tagged any key stored into this cache configuration with both group names.

For instance, whenever a new post is added, we could tell the Cache engine to remove all entries associated to the `article` group:

```
// 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()` can be used to retrieve mapping between group and configurations, i.e.: having the same group:

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

/**
 * A variation of previous example that clears all Cache configurations
 * having the same group
 */
public function afterSave($event, $entity, $options = [])
{
    if ($entity->isNew()) {
        $configs = Cache::groupConfigs('article');
        foreach ($configs['article'] as $config) {
            Cache::clearGroup('article', $config);
```

(continues on next page)

(continued from previous page)

```
        }
    }
}
```

Groups are shared across all cache configs using the same engine and same prefix. If you are using groups and want to take advantage of group deletion, choose a common prefix for all your configs.

Globally Enable or Disable Cache

static Cake\Cache\Cache::**disable**

You may need to disable all Cache read & writes when trying to figure out cache expiration related issues. You can do this using `enable()` and `disable()`:

```
// Disable all cache reads, and cache writes.
Cache::disable();
```

Once disabled, all reads and writes will return `null`.

static Cake\Cache\Cache::**enable**

Once disabled, you can use `enable()` to re-enable caching:

```
// Re-enable all cache reads, and cache writes.
Cache::enable();
```

static Cake\Cache\Cache::**enabled**

If you need to check on the state of Cache, you can use `enabled()`.

Creating a Cache Engine

You can provide custom `Cache` engines in `App\Cache\Engine` as well as in plugins using `$plugin\Cache\Engine`. Cache engines must be in a cache directory. If you had a cache engine named `MyCustomCacheEngine` it would be placed in either **src/Cache/Engine/MyCustomCacheEngine.php**. Or in **plugins/MyPlugin/src/Cache/Engine/MyCustomCacheEngine.php** as part of a plugin. Cache configs from plugins need to use the plugin dot syntax:

```
Cache::setConfig('custom', [
    'className' => 'MyPlugin.MyCustomCache',
    // ...
]);
```

Custom Cache engines must extend *Cake\Cache\CacheEngine* which defines a number of abstract methods as well as provides a few initialization methods.

The required API for a CacheEngine is

class Cake\Cache**CacheEngine**

The base class for all cache engines used with Cache.

Cake\Cache\CacheEngine::**write**(*$key*, *$value*)

Returns
boolean for success.

Write value for a key into cache, Return `true` if the data was successfully cached, `false` on failure.

Cake\Cache\CacheEngine::**read**(*$key*)

Returns
The cached value or `null` for failure.

Read a key from the cache. Return `null` to indicate the entry has expired or does not exist.

Cake\Cache\CacheEngine::**delete**(*$key*)

Returns
Boolean `true` on success.

Delete a key from the cache. Return `false` to indicate that the entry did not exist or could not be deleted.

Cake\Cache\CacheEngine::**clear**(*$check*)

Returns
Boolean `true` on success.

Delete all keys from the cache. If $check is `true`, you should validate that each value is actually expired.

Cake\Cache\CacheEngine::**clearGroup**(*$group*)

Returns
Boolean `true` on success.

Delete all keys from the cache belonging to the same group.

Cake\Cache\CacheEngine::**decrement**(*$key*, *$offset = 1*)

Returns
Boolean `true` on success.

Decrement a number under the key and return decremented value

Cake\Cache\CacheEngine::**increment**(*$key*, *$offset = 1*)

Returns
Boolean `true` on success.

Increment a number under the key and return incremented value

CHAPTER 14

Bake Console

This page has moved[133].

[133] https://book.cakephp.org/bake/2.x/en/index.html

CHAPTER 15

Console Commands

In addition to a web framework, CakePHP also provides a console framework for creating command line tools & applications. Console applications are ideal for handling a variety of background & maintenance tasks that leverage your existing application configuration, models, plugins and domain logic.

CakePHP provides several console tools for interacting with CakePHP features like i18n and routing that enable you to introspect your application and generate related files.

The CakePHP Console

The CakePHP Console uses a dispatcher-type system to load commands, parse their arguments and invoke the correct command. While the examples below use bash the CakePHP console is compatible with any *nix shell and windows.

A CakePHP application contains **src/Command** directory that contain its commands. It also comes with an executable in the **bin** directory:

```
$ cd /path/to/app
$ bin/cake
```

Note: For Windows, the command needs to be `bin\cake` (note the backslash).

Running the Console with no arguments will list out available commands. You could then run the any of the listed commands by using its name:

```
# run server command
bin/cake server

# run migrations command
bin/cake migrations -h
```

(continues on next page)

(continued from previous page)

```
# run bake (with plugin prefix)
bin/cake bake.bake -h
```

Plugin commands can be invoked without a plugin prefix if the commands's name does not overlap with an application or framework command. In the case that two plugins provide a command with the same name, the first loaded plugin will get the short alias. You can always use the `plugin.command` format to unambiguously reference a command.

Console Applications

By default CakePHP will automatically discover all the commands in your application and its plugins. You may want to reduce the number of exposed commands, when building standalone console applications. You can use your `Application`'s `console()` hook to limit which commands are exposed and rename commands that are exposed:

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

use App\Command\UserCommand;
use App\Command\VersionCommand;
use Cake\Console\CommandCollection;
use Cake\Http\BaseApplication;

class Application extends BaseApplication
{
    public function console(CommandCollection $commands): CommandCollection
    {
        // Add by classname
        $commands->add('user', UserCommand::class);

        // Add instance
        $commands->add('version', new VersionCommand());

        return $commands;
    }
}
```

In the above example, the only commands available would be `help`, `version` and `user`. See the *Commands* section for how to add commands in your plugins.

Note: When adding multiple commands that use the same Command class, the `help` command will display the shortest option.

Renaming Commands

There are cases where you will want to rename commands, to create nested commands or subcommands. While the default auto-discovery of commands will not do this, you can register your commands to create any desired naming.

You can customize the command names by defining each command in your plugin:

```
public function console(CommandCollection $commands): CommandCollection
{
    // Add commands with nested naming
    $commands->add('user dump', UserDumpCommand::class);
    $commands->add('user:show', UserShowCommand::class);

    // Rename a command entirely
    $commands->add('lazer', UserDeleteCommand::class);

    return $commands;
}
```

When overriding the `console()` hook in your application, remember to call `$commands->autoDiscover()` to add commands from CakePHP, your application, and plugins.

If you need to rename/remove any attached commands, you can use the `Console.buildCommands` event on your application event manager to modify the available commands.

Commands

See the *Command Objects* chapter on how to create your first command. Then learn more about commands:

Command Objects

class Cake\Console**Command**

CakePHP comes with a number of built-in commands for speeding up your development, and automating routine tasks. You can use these same libraries to create commands for your application and plugins.

Creating a Command

Let's create our first Command. For this example, we'll create a simple Hello world command. In your application's **src/Command** directory create **HelloCommand.php**. Put the following code inside it:

```
<?php
namespace App\Command;

use Cake\Command\Command;
use Cake\Console\Arguments;
use Cake\Console\ConsoleIo;

class HelloCommand extends Command
{
    public function execute(Arguments $args, ConsoleIo $io): int
```

(continues on next page)

(continued from previous page)

```
    {
        $io->out('Hello world.');

        return static::CODE_SUCCESS;
    }
}
```

Command classes must implement an `execute()` method that does the bulk of their work. This method is called when a command is invoked. Lets call our first command application directory, run:

```
bin/cake hello
```

You should see the following output:

```
Hello world.
```

Our `execute()` method isn't very interesting let's read some input from the command line:

```
<?php
namespace App\Command;

use Cake\Command\Command;
use Cake\Console\Arguments;
use Cake\Console\ConsoleIo;
use Cake\Console\ConsoleOptionParser;

class HelloCommand extends Command
{
    protected function buildOptionParser(ConsoleOptionParser $parser):␣
↪ConsoleOptionParser
    {
        $parser->addArgument('name', [
            'help' => 'What is your name',
        ]);
        return $parser;
    }

    public function execute(Arguments $args, ConsoleIo $io): int
    {
        $name = $args->getArgument('name');
        $io->out("Hello {$name}.");

        return static::CODE_SUCCESS;
    }
}
```

After saving this file, you should be able to run the following command:

```
bin/cake hello jillian

# Outputs
Hello jillian
```

Changing the Default Command Name

CakePHP will use conventions to generate the name your commands use on the command line. If you want to overwrite the generated name implement the `defaultName()` method in your command:

```
public static function defaultName(): string
{
    return 'oh_hi';
}
```

The above would make our `HelloCommand` accessible by `cake oh_hi` instead of `cake hello`.

Defining Arguments and Options

As we saw in the last example, we can use the `buildOptionParser()` hook method to define arguments. We can also define options. For example, we could add a `yell` option to our `HelloCommand`:

```
// ...
protected function buildOptionParser(ConsoleOptionParser $parser): ConsoleOptionParser
{
    $parser
        ->addArgument('name', [
            'help' => 'What is your name',
        ])
        ->addOption('yell', [
            'help' => 'Shout the name',
            'boolean' => true,
        ]);

    return $parser;
}

public function execute(Arguments $args, ConsoleIo $io): int
{
    $name = $args->getArgument('name');
    if ($args->getOption('yell')) {
        $name = mb_strtoupper($name);
    }
    $io->out("Hello {$name}.");

    return static::CODE_SUCCESS;
}
```

See the *Option Parsers* section for more information.

Creating Output

Commands are provided a `ConsoleIo` instance when executed. This object allows you to interact with `stdout`, `stderr` and create files. See the *Command Input/Output* section for more information.

Using Models in Commands

You'll often need access to your application's business logic in console commands. You can load models in commands, just as you would in a controller using `$this->fetchTable()` since command use the `LocatorAwareTrait`:

```
<?php
declare(strict_types=1);

namespace App\Command;

use Cake\Command\Command;
use Cake\Console\Arguments;
use Cake\Console\ConsoleIo;
use Cake\Console\ConsoleOptionParser;

class UserCommand extends Command
{
    // Define the default table. This allows you to use `fetchTable()` without any␣
↪argument.
    protected $defaultTable = 'Users';

    protected function buildOptionParser(ConsoleOptionParser $parser):␣
↪ConsoleOptionParser
    {
        $parser
            ->addArgument('name', [
                'help' => 'What is your name'
            ]);

        return $parser;
    }

    public function execute(Arguments $args, ConsoleIo $io): int
    {
        $name = $args->getArgument('name');
        $user = $this->fetchTable()->findByUsername($name)->first();

        $io->out(print_r($user, true));

        return static::CODE_SUCCESS;
    }
}
```

The above command, will fetch a user by username and display the information stored in the database.

Exit Codes and Stopping Execution

When your commands hit an unrecoverable error you can use the `abort()` method to terminate execution:

```
// ...
public function execute(Arguments $args, ConsoleIo $io): int
{
    $name = $args->getArgument('name');
    if (strlen($name) < 5) {
        // Halt execution, output to stderr, and set exit code to 1
        $io->error('Name must be at least 4 characters long.');
        $this->abort();
    }

    return static::CODE_SUCCESS;
}
```

You can also use `abort()` on the `$io` object to emit a message and code:

```
public function execute(Arguments $args, ConsoleIo $io): int
{
    $name = $args->getArgument('name');
    if (strlen($name) < 5) {
        // Halt execution, output to stderr, and set exit code to 99
        $io->abort('Name must be at least 4 characters long.', 99);
    }

    return static::CODE_SUCCESS;
}
```

You can pass any desired exit code into `abort()`.

Tip: Avoid exit codes 64 - 78, as they have specific meanings described by `sysexits.h`. Avoid exit codes above 127, as these are used to indicate process exit by signal, such as SIGKILL or SIGSEGV.

You can read more about conventional exit codes in the sysexit manual page on most Unix systems (`man sysexits`), or the `System Error Codes` help page in Windows.

Calling other Commands

You may need to call other commands from your command. You can use `executeCommand` to do that:

```
// You can pass an array of CLI options and arguments.
$this->executeCommand(OtherCommand::class, ['--verbose', 'deploy']);

// Can pass an instance of the command if it has constructor args
$command = new OtherCommand($otherArgs);
$this->executeCommand($command, ['--verbose', 'deploy']);
```

Note: When calling `executeCommand()` in a loop, it is recommended to pass in the parent command's `ConsoleIo` instance as the optional 3rd argument to avoid a potential "open files" limit that could occur in some environments.

Setting Command Description

You may want to set a command description via:

```
class UserCommand extends Command
{
    public static function getDescription(): string
    {
        return 'My custom description';
    }
}
```

This will show your description in the Cake CLI:

```
bin/cake

App:
  - user
  └── My custom description
```

As well as in the help section of your command:

```
cake user --help
My custom description

Usage:
cake user [-h] [-q] [-v]
```

Testing Commands

To make testing console applications easier, CakePHP comes with a `ConsoleIntegrationTestTrait` trait that can be used to test console applications and assert against their results.

To get started testing your console application, create a test case that uses the `Cake\TestSuite\ConsoleIntegrationTestTrait` trait. This trait contains a method `exec()` that is used to execute your command. You can pass the same string you would use in the CLI to this method.

Note: For CakePHP 4.4 onwards the `Cake\Console\TestSuite\ConsoleIntegrationTestTrait` namespace should be used.

Let's start with a very simple command, located in **src/Command/UpdateTableCommand.php**:

```
namespace App\Command;

use Cake\Command\Command;
use Cake\Console\Arguments;
use Cake\Console\ConsoleIo;
use Cake\Console\ConsoleOptionParser;

class UpdateTableCommand extends Command
{
    protected function buildOptionParser(ConsoleOptionParser $parser):␣
```

(continues on next page)

(continued from previous page)

```
↪ConsoleOptionParser
    {
        $parser->setDescription('My cool console app');

        return $parser;
    }
}
```

To write an integration test for this command, we would create a test case in **tests/TestCase/Command/UpdateTableTest.php** that uses the `Cake\TestSuite\ConsoleIntegrationTestTrait` trait. This command doesn't do much at the moment, but let's just test that our command's description is displayed in `stdout`:

```
namespace App\Test\TestCase\Command;

use Cake\TestSuite\ConsoleIntegrationTestTrait;
use Cake\TestSuite\TestCase;

class UpdateTableCommandTest extends TestCase
{
    use ConsoleIntegrationTestTrait;

    public function testDescriptionOutput()
    {
        $this->exec('update_table --help');
        $this->assertOutputContains('My cool console app');
    }
}
```

Our test passes! While this is very trivial example, it shows that creating an integration test case for console applications can follow command line conventions. Let's continue by adding more logic to our command:

```
namespace App\Command;

use Cake\Command\Command;
use Cake\Console\Arguments;
use Cake\Console\ConsoleIo;
use Cake\Console\ConsoleOptionParser;
use Cake\I18n\DateTime;

class UpdateTableCommand extends Command
{
    protected function buildOptionParser(ConsoleOptionParser $parser):␣
↪ConsoleOptionParser
    {
        $parser
            ->setDescription('My cool console app')
            ->addArgument('table', [
                'help' => 'Table to update',
                'required' => true
            ]);
```

(continues on next page)

(continued from previous page)

```
        return $parser;
    }

    public function execute(Arguments $args, ConsoleIo $io): int
    {
        $table = $args->getArgument('table');
        $this->fetchTable($table)->updateQuery()
            ->set([
                'modified' => new DateTime()
            ])
            ->execute();

        return static::CODE_SUCCESS;
    }
}
```

This is a more complete command that has required options and relevant logic. Modify your test case to the following snippet of code:

```
namespace Cake\Test\TestCase\Command;

use Cake\Command\Command;
use Cake\I18n\DateTime;
use Cake\TestSuite\ConsoleIntegrationTestTrait;
use Cake\TestSuite\TestCase;

class UpdateTableCommandTest extends TestCase
{
    use ConsoleIntegrationTestTrait;

    protected $fixtures = [
        // assumes you have a UsersFixture
        'app.Users',
    ];

    public function testDescriptionOutput()
    {
        $this->exec('update_table --help');
        $this->assertOutputContains('My cool console app');
    }

    public function testUpdateModified()
    {
        $now = new DateTime('2017-01-01 00:00:00');
        DateTime::setTestNow($now);

        $this->loadFixtures('Users');

        $this->exec('update_table Users');
        $this->assertExitCode(Command::CODE_SUCCESS);

        $user = $this->getTableLocator()->get('Users')->get(1);
```

(continues on next page)

(continued from previous page)

```
        $this->assertSame($user->modified->timestamp, $now->timestamp);

        DateTime::setTestNow(null);
    }
}
```

As you can see from the `testUpdateModified` method, we are testing that our command updates the table that we are passing as the first argument. First, we assert that the command exited with the proper status code, `0`. Then we check that our command did its work, that is, updated the table we provided and set the `modified` column to the current time.

Remember, `exec()` will take the same string you type into your CLI, so you can include options and arguments in your command string.

Testing Interactive Shells

Consoles are often interactive. Testing interactive commands with the `Cake\TestSuite\ConsoleIntegrationTestTrait` trait only requires passing the inputs you expect as the second parameter of `exec()`. They should be included as an array in the order that you expect them.

Continuing with our example command, let's add an interactive confirmation. Update the command class to the following:

```
namespace App\Command;

use Cake\Command\Command;
use Cake\Console\Arguments;
use Cake\Console\ConsoleIo;
use Cake\Console\ConsoleOptionParser;
use Cake\I18n\DateTime;

class UpdateTableCommand extends Command
{
    protected function buildOptionParser(ConsoleOptionParser $parser):␣
↪ConsoleOptionParser
    {
        $parser
            ->setDescription('My cool console app')
            ->addArgument('table', [
                'help' => 'Table to update',
                'required' => true
            ]);

        return $parser;
    }

    public function execute(Arguments $args, ConsoleIo $io): int
    {
        $table = $args->getArgument('table');
        if ($io->ask('Are you sure?', 'n', ['y', 'n']) !== 'y') {
            $io->error('You need to be sure.');
            $this->abort();
        }
```

(continues on next page)

(continued from previous page)

```
        $this->fetchTable($table)->updateQuery()
            ->set([
                'modified' => new DateTime()
            ])
            ->execute();

        return static::CODE_SUCCESS;
    }
}
```

Now that we have an interactive command, we can add a test case that tests that we receive the proper response, and one that tests that we receive an incorrect response. Remove the `testUpdateModified` method and, add the following methods to **tests/TestCase/Command/UpdateTableCommandTest.php**:

```
public function testUpdateModifiedSure()
{
    $now = new DateTime('2017-01-01 00:00:00');
    DateTime::setTestNow($now);

    $this->loadFixtures('Users');

    $this->exec('update_table Users', ['y']);
    $this->assertExitCode(Command::CODE_SUCCESS);

    $user = $this->getTableLocator()->get('Users')->get(1);
    $this->assertSame($user->modified->timestamp, $now->timestamp);

    DateTime::setTestNow(null);
}

public function testUpdateModifiedUnsure()
{
    $user = $this->getTableLocator()->get('Users')->get(1);
    $original = $user->modified->timestamp;

    $this->exec('my_console best_framework', ['n']);
    $this->assertExitCode(Command::CODE_ERROR);
    $this->assertErrorContains('You need to be sure.');

    $user = $this->getTableLocator()->get('Users')->get(1);
    $this->assertSame($original, $user->timestamp);
}
```

In the first test case, we confirm the question, and records are updated. In the second test we don't confirm and records are not updated, and we can check that our error message was written to `stderr`.

Assertion methods

The `Cake\TestSuite\ConsoleIntegrationTestTrait` trait provides a number of assertion methods that make help assert against console output:

```
// assert that the command exited as success
$this->assertExitSuccess();

// assert that the command exited as an error
$this->assertExitError();

// assert that the command exited with the expected code
$this->assertExitCode($expected);

// assert that stdout contains a string
$this->assertOutputContains($expected);

// assert that stderr contains a string
$this->assertErrorContains($expected);

// assert that stdout matches a regular expression
$this->assertOutputRegExp($expected);

// assert that stderr matches a regular expression
$this->assertErrorRegExp($expected);
```

Lifecycle Callbacks

Like Controllers, Commands offer lifecycle events that allow you to observe the framework calling your application code. Commands have:

- `Command.beforeExecute` Is called before a command's `execute()` method is. The event is passed the `ConsoleArguments` parameter as `args`. This event cannot be stopped or have its result replaced.
- `Command.afterExecute` Is called after a command's `execute()` method is complete. The event contains `ConsoleArguments` as `args` and the command result as `result`. This event cannot be stopped or have its result replaced.

Command Input/Output

class Cake\Console**ConsoleIo**

CakePHP provides the `ConsoleIo` object to commands so that they can interactively read user input and output information to the user.

Command Helpers

Command Helpers can be accessed and used from any command:

```
// Output some data as a table.
$io->helper('Table')->output($data);

// Get a helper from a plugin.
$io->helper('Plugin.HelperName')->output($data);
```

You can also get instances of helpers and call any public methods on them:

```
// Get and use the Progress Helper.
$progress = $io->helper('Progress');
$progress->increment(10);
$progress->draw();
```

Creating Helpers

While CakePHP comes with a few command helpers you can create more in your application or plugins. As an example, we'll create a simple helper to generate fancy headings. First create the **src/Command/Helper/HeadingHelper.php** and put the following in it:

```
<?php
namespace App\Command\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);
    }
}
```

We can then use this new helper in one of our shell commands by calling it:

```
// With ### on either side
$this->helper('Heading')->output(['It works!']);

// With ~~~~ on either side
$this->helper('Heading')->output(['It works!', '~', 4]);
```

Helpers generally implement the `output()` method which takes an array of parameters. However, because Console Helpers are vanilla classes they can implement additional methods that take any form of arguments.

Note: Helpers can also live in `src/Shell/Helper` for backwards compatibility.

Built-In Helpers

Table Helper

The TableHelper assists in making well formatted ASCII art tables. Using it is pretty simple:

```
$data = [
    ['Header 1', 'Header', 'Long Header'],
    ['short', 'Longish thing', 'short'],
    ['Longer thing', 'short', 'Longest Value'],
];
$io->helper('Table')->output($data);

// Outputs
+--------------+---------------+---------------+
| Header 1     | Header        | Long Header   |
+--------------+---------------+---------------+
| short        | Longish thing | short         |
| Longer thing | short         | Longest Value |
+--------------+---------------+---------------+
```

You can use the `<text-right>` formatting tag in tables to right align content:

```
$data = [
    ['Name', 'Total Price'],
    ['Cake Mix', '<text-right>1.50</text-right>'],
];
$io->helper('Table')->output($data);

// Outputs
+----------+-------------+
| Name 1   | Total Price |
+----------+-------------+
| Cake Mix |        1.50 |
+----------+-------------+
```

Progress Helper

The ProgressHelper can be used in two different ways. The simple mode lets you provide a callback that is invoked until the progress is complete:

```
$io->helper('Progress')->output(['callback' => function ($progress) {
    // Do work here.
    $progress->increment(20);
    $progress->draw();
}]);
```

You can control the progress bar more by providing additional options:

- `total` The total number of items in the progress bar. Defaults to 100.
- `width` The width of the progress bar. Defaults to 80.
- `callback` The callback that will be called in a loop to advance the progress bar.

An example of all the options in use would be:

```
$io->helper('Progress')->output([
    'total' => 10,
    'width' => 20,
    'callback' => function ($progress) {
        $progress->increment(2);
        $progress->draw();
    }
]);
```

The progress helper can also be used manually to increment and re-render the progress bar as necessary:

```
$progress = $io->helper('Progress');
$progress->init([
    'total' => 10,
    'width' => 20,
]);

$progress->increment(4);
$progress->draw();
```

Getting User Input

Cake\Console\ConsoleIo::**ask**(*$question*, *$choices = null*, *$default = null*)

When building interactive console applications you'll need to get user input. CakePHP provides a way to do this:

```
// Get arbitrary text from the user.
$color = $io->ask('What color do you like?');

// Get a choice from the user.
$selection = $io->askChoice('Red or Green?', ['R', 'G'], 'R');
```

Selection validation is case-insensitive.

Creating Files

Cake\Console\ConsoleIo::**createFile**(*$path*, *$contents*)

Creating files is often important part of many console commands that help automate development and deployment. The `createFile()` method gives you a simple interface for creating files with interactive confirmation:

```
// Create a file with confirmation on overwrite
$io->createFile('bower.json', $stuff);

// Force overwriting without asking
$io->createFile('bower.json', $stuff, true);
```

Creating Output

Writing to `stdout` and `stderr` is another common operation in CakePHP:

```
// Write to stdout
$io->out('Normal message');

// Write to stderr
$io->err('Error message');
```

In addition to vanilla output methods, CakePHP provides wrapper methods that style output with appropriate ANSI colors:

```
// Green text on stdout
$io->success('Success message');

// Cyan text on stdout
$io->info('Informational text');

// Blue text on stdout
$io->comment('Additional context');

// Red text on stderr
$io->error('Error text');

// Yellow text on stderr
$io->warning('Warning text');
```

Color formatting will automatically be disabled if `posix_isatty` returns true, or if the `NO_COLOR` environment variable is set.

`ConsoleIo` provides two convenience methods regarding the output level:

```
// Would only appear when verbose output is enabled (-v)
$io->verbose('Verbose message');

// Would appear at all levels.
$io->quiet('Quiet message');
```

You can also create blank lines or draw lines of dashes:

```
// Output 2 newlines
$io->out($io->nl(2));

// Draw a horizontal line
$io->hr();
```

Lastly, you can update the current line of text on the screen:

```
$io->out('Counting down');
$io->out('10', 0);
for ($i = 9; $i > 0; $i--) {
    sleep(1);
    $io->overwrite($i, 0, 2);
}
```

Note: It is important to remember, that you cannot overwrite text once a new line has been output.

Output Levels

Console applications often need different levels of verbosity. For example, when running as a cron job, most output is un-necessary. You can use output levels to flag output appropriately. The user of the shell, can then decide what level of detail they are interested in by setting the correct flag when calling the command. There are 3 levels:

- `QUIET` - Only absolutely important information should be marked for quiet output.
- `NORMAL` - The default level, and normal usage.
- `VERBOSE` - Mark messages that may be too noisy for everyday use, but helpful for debugging as `VERBOSE`.

You can mark output as follows:

```
// Would appear at all levels.
$io->out('Quiet message', 1, ConsoleIo::QUIET);
$io->quiet('Quiet message');

// Would not appear when quiet output is toggled.
$io->out('normal message', 1, ConsoleIo::NORMAL);
$io->out('loud message', 1, ConsoleIo::VERBOSE);
$io->verbose('Verbose output');

// Would only appear when verbose output is enabled.
$io->out('extra message', 1, ConsoleIo::VERBOSE);
$io->verbose('Verbose output');
```

You can control the output level of commands, by using the `--quiet` and `--verbose` options. These options are added by default, and allow you to consistently control output levels inside your CakePHP comands.

The `--quiet` and `--verbose` options also control how logging data is output to stdout/stderr. Normally info and higher log messages are output to stdout/stderr. When `--verbose` is used, debug logs will be output to stdout. When `--quiet` is used, only warning and higher log messages will be output to stderr.

Styling Output

Styling output is done by including tags - just like HTML - in your output. These tags will be replaced with the correct ansi code sequence, or stripped if you are on a console that doesn't support ansi codes. There are several built-in styles, and you can create more. The built-in ones are

- `success` Success messages. Green text.
- `error` Error messages. Red text.
- `warning` Warning messages. Yellow text.
- `info` Informational messages. Cyan text.
- `comment` Additional text. Blue text.
- `question` Text that is a question, added automatically by shell.

You can create additional styles using `$io->setStyle()`. To declare a new output style you could do:

```
$io->setStyle('flashy', ['text' => 'magenta', 'blink' => true]);
```

This would then allow you to use a <flashy> tag in your shell output, and if ansi colors are enabled, the following would be rendered as blinking magenta text `$this->out('<flashy>Whoooa</flashy> Something went wrong');`. When defining styles you can use the following colors for the `text` and `background` attributes:

- black
- blue
- cyan
- green
- magenta
- red
- white
- yellow

You can also use the following options as boolean switches, setting them to a truthy value enables them.

- blink
- bold
- reverse
- underline

Adding a style makes it available on all instances of ConsoleOutput as well, so you don't have to redeclare styles for both stdout and stderr objects.

Turning Off Coloring

Although coloring is pretty, there may be times when you want to turn it off, or force it on:

```
$io->outputAs(ConsoleOutput::RAW);
```

The above will put the output object into raw output mode. In raw output mode, no styling is done at all. There are three modes you can use.

- `ConsoleOutput::COLOR` - Output with color escape codes in place.
- `ConsoleOutput::PLAIN` - Plain text output, known style tags will be stripped from the output.
- `ConsoleOutput::RAW` - Raw output, no styling or formatting will be done. This is a good mode to use if you are outputting XML or, want to debug why your styling isn't working.

By default on *nix systems ConsoleOutput objects default to color output. On Windows systems, plain output is the default unless the `ANSICON` environment variable is present.

Option Parsers

class Cake\Console**ConsoleOptionParser**

Console applications typically take options and arguments as the primary way to get information from the terminal into your commands.

Defining an OptionParser

Commands and Shells provide a `buildOptionParser($parser)` hook method that you can use to define the options and arguments for your commands:

```
protected function buildOptionParser(ConsoleOptionParser $parser): ConsoleOptionParser
{
    // Define your options and arguments.

    // Return the completed parser
    return $parser;
}
```

Shell classes use the `getOptionParser()` hook method to define their option parser:

```
public function getOptionParser()
{
    // Get an empty parser from the framework.
    $parser = parent::getOptionParser();

    // Define your options and arguments.

    // Return the completed parser
    return $parser;
}
```

Using Arguments

Cake\Console\ConsoleOptionParser::**addArgument**(*$name*, *$params = []*)

Positional arguments are frequently used in command line tools, and `ConsoleOptionParser` allows you to define positional arguments as well as make them required. You can add arguments one at a time with `$parser->addArgument();` or multiple at once with `$parser->addArguments();`:

```
$parser->addArgument('model', ['help' => 'The model to bake']);
```

You can use the following options when creating an argument:

- `help` The help text to display for this argument.
- `required` Whether this parameter is required.
- `index` The index for the arg, if left undefined the argument will be put onto the end of the arguments. If you define the same index twice the first option will be overwritten.
- `choices` An array of valid choices for this argument. If left empty all values are valid. An exception will be raised when parse() encounters an invalid value.

Arguments that have been marked as required will throw an exception when parsing the command if they have been omitted. So you don't have to handle that in your shell.

Adding Multiple Arguments

Cake\Console\ConsoleOptionParser::**addArguments**(*array $args*)

If you have an array with multiple arguments you can use `$parser->addArguments()` to add multiple arguments at once.

```
$parser->addArguments([
    'node' => ['help' => 'The node to create', 'required' => true],
    'parent' => ['help' => 'The parent node', 'required' => true],
]);
```

As with all the builder methods on ConsoleOptionParser, addArguments can be used as part of a fluent method chain.

Validating Arguments

When creating positional arguments, you can use the `required` flag, to indicate that an argument must be present when a shell is called. Additionally you can use `choices` to force an argument to be from a list of valid choices:

```
$parser->addArgument('type', [
    'help' => 'The type of node to interact with.',
    'required' => true,
    'choices' => ['aro', 'aco'],
]);
```

The above will create an argument that is required and has validation on the input. If the argument is either missing, or has an incorrect value an exception will be raised and the shell will be stopped.

Using Options

Cake\Console\ConsoleOptionParser::**addOption**(*$name*, *array $options = []*)

Options or flags are used in command line tools to provide unordered key/value arguments for your commands. Options can define both verbose and short aliases. They can accept a value (e.g `--connection=default`) or be boolean options (e.g `--verbose`). Options are defined with the `addOption()` method:

```
$parser->addOption('connection', [
    'short' => 'c',
    'help' => 'connection',
    'default' => 'default',
]);
```

The above would allow you to use either `cake myshell --connection=other`, `cake myshell --connection other`, or `cake myshell -c other` when invoking the shell.

Boolean switches do not accept or consume values, and their presence just enables them in the parsed parameters:

```
$parser->addOption('no-commit', ['boolean' => true]);
```

This option when used like `cake mycommand --no-commit something` would have a value of `true`, and 'something' would be a treated as a positional argument.

When creating options you can use the following options to define the behavior of the option:

- `short` - The single letter variant for this option, leave undefined for none.
- `help` - Help text for this option. Used when generating help for the option.
- `default` - The default value for this option. If not defined the default will be `true`.
- `boolean` - The option uses no value, it's just a boolean switch. Defaults to `false`.
- `multiple` - The option can be provided multiple times. The parsed option will be an array of values when this option is enabled.
- `choices` - An array of valid choices for this option. If left empty all values are valid. An exception will be raised when parse() encounters an invalid value.

Adding Multiple Options

`Cake\Console\ConsoleOptionParser::addOptions`(*array $options*)

If you have an array with multiple options you can use `$parser->addOptions()` to add multiple options at once.

```
$parser->addOptions([
    'node' => ['short' => 'n', 'help' => 'The node to create'],
    'parent' => ['short' => 'p', 'help' => 'The parent node'],
]);
```

As with all the builder methods on ConsoleOptionParser, addOptions can be used as part of a fluent method chain.

Validating Options

Options can be provided with a set of choices much like positional arguments can be. When an option has defined choices, those are the only valid choices for an option. All other values will raise an `InvalidArgumentException`:

```
$parser->addOption('accept', [
    'help' => 'What version to accept.',
    'choices' => ['working', 'theirs', 'mine'],
]);
```

Using Boolean Options

Options can be defined as boolean options, which are useful when you need to create some flag options. Like options with defaults, boolean options always include themselves into the parsed parameters. When the flags are present they are set to `true`, when they are absent they are set to `false`:

```
$parser->addOption('verbose', [
    'help' => 'Enable verbose output.',
    'boolean' => true
]);
```

The following option would always have a value in the parsed parameter. When not included its default value would be `false`, and when defined it will be `true`.

Building a ConsoleOptionParser from an Array

Cake\Console\ConsoleOptionParser::**buildFromArray**(*$spec*)

Option parsers can also be defined as arrays. Within the array, you can define keys for `arguments`, `options`, `description` and `epilog`. The values for arguments, and options, should follow the format that *Cake\Console\ConsoleOptionParser::addArguments()* and *Cake\Console\ConsoleOptionParser::addOptions()* use. You can also use `buildFromArray` on its own, to build an option parser:

```
public function getOptionParser()
{
    return ConsoleOptionParser::buildFromArray([
        'description' => [
            __("Use this command to grant ACL permissions. Once executed, the "),
            __("ARO specified (and its children, if any) will have ALLOW access "),
            __("to the specified ACO action (and the ACO's children, if any).")
        ],
        'arguments' => [
            'aro' => ['help' => __('ARO to check.'), 'required' => true],
            'aco' => ['help' => __('ACO to check.'), 'required' => true],
            'action' => ['help' => __('Action to check')],
        ],
    ]);
}
```

Merging Option Parsers

Cake\Console\ConsoleOptionParser::**merge**(*$spec*)

When building a group command, you maybe want to combine several parsers for this:

```
$parser->merge($anotherParser);
```

Note that the order of arguments for each parser must be the same, and that options must also be compatible for it work. So do not use keys for different things.

Getting Help from Shells

By defining your options and arguments with the option parser CakePHP can automatically generate rudimentary help information and add a `--help` and `-h` to each of your commands. Using one of these options will allow you to see the generated help content:

```
bin/cake bake --help
bin/cake bake -h
```

Would both generate the help for bake. You can also get help for nested commands:

```
bin/cake bake model --help
bin/cake bake model -h
```

The above would get you the help specific to bake's model command.

Getting Help as XML

When building automated tools or development tools that need to interact with CakePHP shell commands, it's nice to have help available in a machine parse-able format. By providing the `xml` option when requesting help you can have help content returned as XML:

```
cake bake --help xml
cake bake -h xml
```

The above would return an XML document with the generated help, options, and arguments for the selected shell. A sample XML document would look like:

```
<?xml version="1.0"?>
<shell>
    <command>bake fixture</command>
    <description>Generate fixtures for use with the test suite. You can use
        `bake fixture all` to bake all fixtures.</description>
    <epilog>
        Omitting all arguments and options will enter into an interactive
        mode.
    </epilog>
    <options>
        <option name="--help" short="-h" boolean="1">
            <default/>
            <choices/>
        </option>
        <option name="--verbose" short="-v" boolean="1">
            <default/>
            <choices/>
        </option>
        <option name="--quiet" short="-q" boolean="1">
            <default/>
            <choices/>
        </option>
        <option name="--count" short="-n" boolean="">
            <default>10</default>
            <choices/>
        </option>
        <option name="--connection" short="-c" boolean="">
            <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.
```

(continues on next page)

(continued from previous page)

```
            Can use Plugin.name to bake plugin fixtures." required="">
            <choices/>
        </argument>
    </arguments>
</shell>
```

Customizing Help Output

You can further enrich the generated help content by adding a description, and epilog.

Set the Description

Cake\Console\ConsoleOptionParser::**setDescription**(*$text*)

The description displays above the argument and option information. By passing in either an array or a string, you can set the value of the description:

```
// Set multiple lines at once
$parser->setDescription(['line one', 'line two']);

// Read the current value
$parser->getDescription();
```

Set the Epilog

Cake\Console\ConsoleOptionParser::**setEpilog**(*$text*)

Gets or sets the epilog for the option parser. The epilog is displayed after the argument and option information. By passing in either an array or a string, you can set the value of the epilog:

```
// Set multiple lines at once
$parser->setEpilog(['line one', 'line two']);

// Read the current value
$parser->getEpilog();
```

Running Shells as Cron Jobs

A common thing to do with a shell is making it run as a cronjob to clean up the database once in a while or send newsletters. This is trivial to setup, for example:

```
  */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)
```

(continues on next page)

(continued from previous page)

```
# |     |     \------------------ day of month (1 - 31)
# |     \------------------------ hour (0 - 23)
# \-------------------------------- min (0 - 59)
```

You can see more info here: https://en.wikipedia.org/wiki/Cron

Tip: Use `-q` (or *–quiet*) to silence any output for cronjobs.

Cron Jobs on Shared Hosting

On some shared hostings `cd /full/path/to/root && bin/cake mycommand myparam` might not work. Instead you can use `php /full/path/to/root/bin/cake.php mycommand myparam`.

Note: register_argc_argv has to be turned on by including `register_argc_argv = 1` in your php.ini. If you cannot change register_argc_argv globally, you can tell the cron job to use your own configuration by specifying it with `-d register_argc_argv=1` parameter. Example: `php -d register_argc_argv=1 /full/path/to/root/bin/cake.php myshell myparam`

CakePHP Provided Commands

Cache Tool

To help you better manage cached data from a CLI environment, a console command is available for clearing cached data your application has:

```
// Clear one cache config
bin/cake cache clear <configname>

// Clear all cache configs
bin/cake cache clear_all

// Clear one cache group
bin/cake cache clear_group <groupname>
```

Completion Tool

Working with the console gives the developer a lot of possibilities but having to completely know and write those commands can be tedious. Especially when developing new shells where the commands differ per minute iteration. The Completion Shells aids in this matter by providing an API to write completion scripts for shells like bash, zsh, fish etc.

Sub Commands

The Completion Shell consists of a number of sub commands to assist the developer creating its completion script. Each for a different step in the autocompletion process.

Commands

For the first step commands outputs the available Shell Commands, including plugin name when applicable. (All returned possibilities, for this and the other sub commands, are separated by a space.) For example:

```
bin/cake Completion commands
```

Returns:

```
acl api bake command_list completion console i18n schema server test testsuite upgrade
```

Your completion script can select the relevant commands from that list to continue with. (For this and the following sub commands.)

subCommands

Once the preferred command has been chosen subCommands comes in as the second step and outputs the possible sub command for the given shell command. For example:

```
bin/cake Completion subcommands bake
```

Returns:

```
controller db_config fixture model plugin project test view
```

options

As the third and final options outputs options for the given (sub) command as set in getOptionParser. (Including the default options inherited from Shell.) For example:

```
bin/cake Completion options bake
```

Returns:

```
--help -h --verbose -v --quiet -q --everything --connection -c --force -f --plugin -p --
↪prefix --theme -t
```

You can also pass an additional argument being the shell sub-command : it will output the specific options of this sub-command.

How to enable Bash autocompletion for the CakePHP Console

First, make sure the **bash-completion** library is installed. If not, you do it with the following command:

```
apt-get install bash-completion
```

Create a file named **cake** in **/etc/bash_completion.d/** and put the *Bash Completion file content* inside it.

Save the file, then restart your console.

Note: If you are using MacOS X, you can install the **bash-completion** library using **homebrew** with the command `brew install bash-completion`. The target directory for the **cake** file will be **/usr/local/etc/bash_completion.d/**.

Bash Completion file content

This is the code you need to put inside the **cake** file in the correct location in order to get autocompletion when using the CakePHP console:

```
#
# Bash completion file for CakePHP console
#

_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}) )
```

(continues on next page)

(continued from previous page)

```
        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
```

Using autocompletion

Once enabled, the autocompletion can be used the same way than for other built-in commands, using the **TAB** key. Three type of autocompletion are provided. The following output are from a fresh CakePHP install.

Commands

Sample output for commands autocompletion:

```
$ bin/cake <tab>
bake        i18n        schema_cache   routes
console     migrations  plugin         server
```

Subcommands

Sample output for subcommands autocompletion:

```
$ bin/cake bake <tab>
behavior            helper              command
cell                mailer              command_helper
component           migration           template
controller          migration_snapshot  test
fixture             model
form                plugin
```

Options

Sample output for subcommands options autocompletion:

```
$ bin/cake bake -<tab>
-c            --everything  --force       --help        --plugin      -q            -t  ␣
↪         -v
--connection  -f            -h            -p            --prefix      --quiet       --
↪theme        --verbose
```

I18N Tool

The i18n features of CakePHP use po files[134] as their translation source. PO files integrate with commonly used translation tools like Poedit[135].

The i18n commands provides a quick way to generate po template files. These templates files can then be given to translators so they can translate the strings in your application. Once you have translations done, pot files can be merged with existing translations to help update your translations.

Generating POT Files

POT files can be generated for an existing application using the `extract` command. This command will scan your entire application for `__()` style function calls, and extract the message string. Each unique string in your application will be combined into a single POT file:

```
bin/cake i18n extract
```

The above will run the extraction command. The result of this command will be the file **resources/locales/default.pot**. You use the pot file as a template for creating po files. If you are manually creating po files from the pot file, be sure to correctly set the `Plural-Forms` header line.

Generating POT Files for Plugins

You can generate a POT file for a specific plugin using:

```
bin/cake i18n extract --plugin <Plugin>
```

This will generate the required POT files used in the plugins.

Extracting from multiple folders at once

Sometimes, you might need to extract strings from more than one directory of your application. For instance, if you are defining some strings in the `config` directory of your application, you probably want to extract strings from this directory as well as from the `src` directory. You can do it by using the `--paths` option. It takes a comma-separated list of absolute paths to extract:

```
bin/cake i18n extract --paths /var/www/app/config,/var/www/app/src
```

[134] https://en.wikipedia.org/wiki/GNU_gettext
[135] https://www.poedit.net/

Excluding Folders

You can pass a comma separated list of folders that you wish to be excluded. Any path containing a path segment with the provided values will be ignored:

```
bin/cake i18n extract --exclude vendor,tests
```

Skipping Overwrite Warnings for Existing POT Files

By adding `--overwrite`, the shell script will no longer warn you if a POT file already exists and will overwrite by default:

```
bin/cake i18n extract --overwrite
```

Extracting Messages from the CakePHP Core Libraries

By default, the extract shell script will ask you if you like to extract the messages used in the CakePHP core libraries. Set `--extract-core` to yes or no to set the default behavior:

```
bin/cake i18n extract --extract-core yes

// or

bin/cake i18n extract --extract-core no
```

Plugin Tool

The plugin tool allows you to load and unload plugins via the command prompt. If you need help, run:

```
bin/cake plugin --help
```

Loading Plugins

Via the `Load` task you are able to load plugins in your **config/bootstrap.php**. You can do this by running:

```
bin/cake plugin load MyPlugin
```

This will add the following to your **src/Application.php**:

```
// In the bootstrap method add:
$this->addPlugin('MyPlugin');
```

Unloading Plugins

You can unload a plugin by specifying its name:

```
bin/cake plugin unload MyPlugin
```

This will remove the line `$this->addPlugin('MyPlugin',...)` from **src/Application.php**.

Plugin Assets

CakePHP by default serves plugins assets using the `AssetMiddleware` middleware. While this is a good convenience, it is recommended to symlink / copy the plugin assets under app's webroot so that they can be directly served by the web server without invoking PHP. You can do this by running:

```
bin/cake plugin assets symlink
```

Running the above command will symlink all plugins assets under app's webroot. On Windows, which doesn't support symlinks, the assets will be copied in respective folders instead of being symlinked.

You can symlink assets of one particular plugin by specifying its name:

```
bin/cake plugin assets symlink MyPlugin
```

Routes Tool

The routes tool provides a simple to use CLI interface for testing and debugging routes. You can use it to test how routes are parsed, and what URLs routing parameters will generate.

Getting a List of all Routes

```
bin/cake routes
```

Testing URL parsing

You can quickly see how a URL will be parsed using the `check` method:

```
bin/cake routes check /articles/edit/1
```

If your route contains any query string parameters remember to surround the URL in quotes:

```
bin/cake routes check "/articles/?page=1&sort=title&direction=desc"
```

Testing URL Generation

You can see the URL a *routing array* will generate using the `generate` method:

```
bin/cake routes generate controller:Articles action:edit 1
```

Server Tool

The `ServerCommand` lets you stand up a simple webserver using the built in PHP webserver. While this server is *not* intended for production use it can be handy in development when you want to quickly try an idea out and don't want to spend time configuring Apache or Nginx. You can start the server command with:

```
bin/cake server
```

You should see the server boot up and attach to port 8765. You can visit the CLI server by visiting `http://localhost:8765` in your web-browser. You can close the server by pressing `CTRL-C` in your terminal.

Note: Try `bin/cake server -H 0.0.0.0` if the server is unreachable from other hosts.

Changing the Port and Document Root

You can customize the port and document root using options:

```
bin/cake server --port 8080 --document_root path/to/app
```

Interactive Console (REPL)

CakePHP offers REPL(Read Eval Print Loop) plugin[136] to let you explore some CakePHP and your application in an interactive console.

Note: The plugin was shipped with the CakePHP app skeleton before 4.3.

You can start the interactive console using:

```
bin/cake console
```

This will bootstrap your application and start an interactive console. At this point you can interact with your application code and execute queries using your application's models:

```
bin/cake console

>>> $articles = Cake\Datasource\FactoryLocator::get('Table')->get('Articles');
// object(Cake\ORM\Table)(
//
// )
>>> $articles->find()->all();
```

[136] https://github.com/cakephp/repl

Since your application has been bootstrapped you can also test routing using the REPL:

```
>>> Cake\Routing\Router::parse('/articles/view/1');
// [
//   'controller' => 'Articles',
//   'action' => 'view',
//   'pass' => [
//     0 => '1'
//   ],
//   'plugin' => NULL
// ]
```

You can also test generating URLs:

```
>>> Cake\Routing\Router::url(['controller' => 'Articles', 'action' => 'edit', 99]);
// '/articles/edit/99'
```

To quit the REPL you can use `CTRL-C` or by typing `exit`.

Routing in the Console Environment

In command-line interface (CLI), specifically your console commands, `env('HTTP_HOST')` and other webbrowser specific environment variables are not set.

If you generate reports or send emails that make use of `Router::url()` those will contain the default host `http://localhost/` and thus resulting in invalid URLs. In this case you need to specify the domain manually. You can do that using the Configure value `App.fullBaseUrl` from your bootstrap or config, for example.

For sending emails, you should provide Email class with the host you want to send the email with:

```
use Cake\Mailer\Email;

$email = new Email();
$email->setDomain('www.example.org');
```

This asserts that the generated message IDs are valid and fit to the domain the emails are sent from.

CHAPTER 16

Debugging

Debugging is an inevitable and necessary part of any development cycle. While CakePHP doesn't offer any tools that directly connect with any IDE or editor, CakePHP does provide several tools to assist in debugging and exposing what is running under the hood of your application.

Basic Debugging

debug(*mixed $var*, *boolean $showHtml = null*, *$showFrom = true*)

The `debug()` function is a globally available function that works similarly to the PHP function `print_r()`. The `debug()` function allows you to show the contents of a variable in a number of different ways. First, if you'd like data to be shown in an HTML-friendly way, set the second parameter to `true`. The function also prints out the line and file it is originating from by default.

Output from this function is only shown if the core `$debug` variable has been set to `true`.

Also see `dd()`, `pr()` and `pj()`.

stackTrace()

The `stackTrace()` function is available globally, and allows you to output a stack trace wherever the function is called.

breakpoint()

If you have Psysh[137] installed you can use this function in CLI environments to open an interactive console with the current local scope:

```
// Some code
eval(breakpoint());
```

[137] https://psysh.org/

Will open an interactive console that can be used to check local variables and execute other code. You can exit the interactive debugger and resume the original execution by running `quit` or `q` in the interactive session.

Using the Debugger Class

class Cake\Error**Debugger**

To use the debugger, first ensure that `Configure::read('debug')` is set to `true`. You can use `filter_var(env('DEBUG', true), FILTER_VALIDATE_BOOLEAN)`, in **config/app.php** file to ensure that `debug` is a boolean.

The following configuration options can be set in **config/app.php** to change how `Debugger` behaves:

- `Debugger.editor` Choose the which editor URL format you want to use. By default atom, emacs, macvim, phpstorm, sublime, textmate, and vscode are available. You can add additional editor link formats using `Debugger::addEditor()` during your application bootstrap.
- `Debugger.outputMask` A mapping of `key` to `replacement` values that `Debugger` should replace in dumped data and logs generated by `Debugger`.

Outputting Values

static Cake\Error\Debugger::**dump**(*$var*, *$depth = 3*)

Dump prints out the contents of a variable. It will print out all properties and methods (if any) of the supplied variable:

```
$foo = [1,2,3];

Debugger::dump($foo);

// Outputs
array(
    1,
    2,
    3
)

// Simple object
$car = new Car();

Debugger::dump($car);

// Outputs
object(Car) {
    color => 'red'
    make => 'Toyota'
    model => 'Camry'
    mileage => (int)15000
}
```

Masking Data

When dumping data with `Debugger` or rendering error pages, you may want to hide sensitive keys like passwords or API keys. In your **config/bootstrap.php** you can mask specific keys:

```
Debugger::setOutputMask([
    'password' => 'xxxxx',
    'awsKey' => 'yyyyy',
]);
```

As of 4.1.0 you can use the `Debugger.outputMask` configuration value to set output masks.

Logging With Stack Traces

static Cake\Error\Debugger::**log**(*$var*, *$level = 7*, *$depth = 3*)

Creates a detailed stack trace log at the time of invocation. The `log()` method prints out data similar to that done by `Debugger::dump()`, but to the debug.log instead of the output buffer. Note your **tmp** directory (and its contents) must be writable by the web server for `log()` to work correctly.

Generating Stack Traces

static Cake\Error\Debugger::**trace**(*$options*)

Returns the current stack trace. Each line of the trace includes the calling method, including which file and line the call originated from:

```
// In PostsController::index()
pr(Debugger::trace());

// Outputs
PostsController::index() - APP/Controller/DownloadsController.php, line 48
Dispatcher::_invoke() - CORE/src/Routing/Dispatcher.php, line 265
Dispatcher::dispatch() - CORE/src/Routing/Dispatcher.php, line 237
[main] - APP/webroot/index.php, line 84
```

Above is the stack trace generated by calling `Debugger::trace()` in a controller action. Reading the stack trace bottom to top shows the order of currently running functions (stack frames).

Getting an Excerpt From a File

static Cake\Error\Debugger::**excerpt**(*$file*, *$line*, *$context*)

Grab an excerpt from the file at $path (which is an absolute filepath), highlights line number $line with $context number of lines around it.

```
pr(Debugger::excerpt(ROOT . DS . LIBS . 'debugger.php', 321, 2));

// Will output the following.
```

(continues on next page)

(continued from previous page)

```
Array
(
    [0] => <code><span style="color: #000000"> * @access public</span></code>
    [1] => <code><span style="color: #000000"> */</span></code>
    [2] => <code><span style="color: #000000">    function excerpt($file, $line,
↪$context = 2) {</span></code>

    [3] => <span class="code-highlight"><code><span style="color: #000000">        $data␣
↪= $lines = array();</span></code></span>
    [4] => <code><span style="color: #000000">        $data = @explode("\n", file_get_
↪contents($file));</span></code>
)
```

Although this method is used internally, it can be handy if you're creating your own error messages or log entries for custom situations.

static Cake\Error\Debugger::**getType**(*$var*)

Get the type of a variable. Objects will return their class name

Editor Integration

Exception and error pages can contain URLs that directly open in your editor or IDE. CakePHP ships with URL formats for several popular editors, and you can add additional editor formats if required during application bootstrap:

```
// Generate links for vscode.
Debugger::setEditor('vscode')

// Add a custom format
// Format strings will have the {file} and {line}
// placeholders replaced.
Debugger::addEditor('custom', 'thing://open={file}&line={line}');

// You can also use a closure to generate URLs
Debugger::addEditor('custom', function ($file, $line) {
    return "thing://open={$file}&line={$line}";
});
```

Using Logging to Debug

Logging messages is another good way to debug applications, and you can use *Cake\Log\Log* to do logging in your application. All objects that use `LogTrait` have an instance method `log()` which can be used to log messages:

```
$this->log('Got here', 'debug');
```

The above would write `Got here` into the debug log. You can use log entries to help debug methods that involve redirects or complicated loops. You can also use *Cake\Log\Log::write()* to write log messages. This method can be called statically anywhere in your application one Log has been loaded:

```php
// At the top of the file you want to log in.
use Cake\Log\Log;

// Anywhere that Log has been imported.
Log::debug('Got here');
```

Debug Kit

DebugKit is a plugin that provides a number of good debugging tools. It primarily provides a toolbar in the rendered HTML, that provides a plethora of information about your application and the current request. See the DebugKit Documentation[138] for how to install and use DebugKit.

[138] https://book.cakephp.org/debugkit/

CHAPTER 17

Deployment

Once your app is ready to be deployed there are a few things you should do.

Moving files

You can clone your repository onto your production server and then checkout the commit/tag you want to run. Then, run `composer install`. While this requires some knowledge about git and an existing install of `git` and `composer` this process will take care about library dependencies and file and folder permissions.

Be aware that when deploying via FTP you will have to fix file and folder permissions.

You can also use this deployment technique to setup a staging or demo-server (pre-production) and keep it in sync with your local environment.

Adjusting Configuration

You'll want to make a few adjustments to your application's configuration for a production environment. The value of `debug` is extremely important. Turning debug `= false` disables a number of development features that should never be exposed to the Internet at large. Disabling debug changes the following features:

- Debug messages, created with `pr()`, `debug()` and `dd()` are disabled.
- Core CakePHP caches duration are defaulted to 365 days, instead of 10 seconds as in development.
- Error views are less informative, and generic error pages are displayed instead of detailed error messages with stack traces.
- PHP Warnings and Errors are not displayed.

In addition to the above, many plugins and application extensions use `debug` to modify their behavior.

You can check against an environment variable to set the debug level dynamically between environments. This will avoid deploying an application with debug `true` and also save yourself from having to change the debug level each time before deploying to a production environment.

For example, you can set an environment variable in your Apache configuration:

```
SetEnv CAKEPHP_DEBUG 1
```

And then you can set the debug level dynamically in **app_local.php**:

```
$debug = (bool)getenv('CAKEPHP_DEBUG');

return [
    'debug' => $debug,
    .....
];
```

It is recommended that you put configuration that is shared across all of your application's environments in **config/app.php**. For configuration that varies between environments either use **config/app_local.php** or environment variables.

Check Your Security

If you're throwing your application out into the wild, it's a good idea to make sure it doesn't have any obvious leaks:

- Ensure you are using the *Cross Site Request Forgery (CSRF) Middleware* component or middleware.
- You may want to enable the *FormProtection* component. It can help prevent several types of form tampering and reduce the possibility of mass-assignment issues.
- Ensure your models have the correct *Validation* rules enabled.
- Check that only your `webroot` directory is publicly visible, and that your secrets (such as your app salt, and any security keys) are private and unique as well.

Set Document Root

Setting the document root correctly on your application is an important step to keeping your code secure and your application safer. CakePHP applications should have the document root set to the application's `webroot`. This makes the application and configuration files inaccessible through a URL. Setting the document root is different for different webservers. See the *URL Rewriting* documentation for webserver specific information.

In all cases you will want to set the virtual host/domain's document to be `webroot/`. This removes the possibility of files outside of the webroot directory being executed.

Improve Your Application's Performance

Class loading can take a big share of your application's processing time. In order to avoid this problem, it is recommended that you run this command in your production server once the application is deployed:

```
php composer.phar dumpautoload -o
```

Since handling static assets, such as images, JavaScript and CSS files of plugins, through the `Dispatcher` is incredibly inefficient, it is strongly recommended to symlink them for production. This can be done by using the `plugin` command:

```
bin/cake plugin assets symlink
```

The above command will symlink the `webroot` directory of all loaded plugins to appropriate path in the app's `webroot` directory.

If your filesystem doesn't allow creating symlinks the directories will be copied instead of being symlinked. You can also explicitly copy the directories using:

```
bin/cake plugin assets copy
```

CakePHP uses `assert()` internally to provide runtime type checking and provide better error messages during development. You can have PHP skip these assertions by updating your `php.ini` to include:

```
; Turn off assert() code generation.
zend.assertions = -1
```

Skipping code generation for `assert()` will yield faster runtime performance, and is recommended for applications that have good test coverage or that are using a static analyzer.

Deploying an update

On each deploy you'll likely have a few tasks to co-ordinate on your web server. Some typical ones are:

1. Install dependencies with `composer install`. Avoid using `composer update` when doing deploys as you could get unexpected versions of packages.
2. Run database migrations with either the Migrations plugin or another tool.
3. Clear model schema cache with `bin/cake schema_cache clear`. The *Schema Cache Tool* has more information on this command.

CHAPTER 18

Mailer

class Cake\Mailer**Mailer**(*string|array|null $profile = null*)

`Mailer` is a convenience class for sending email. With this class you can send email from any place inside of your application.

Basic Usage

First of all, you should ensure the class is loaded:

```
use Cake\Mailer\Mailer;
```

After you've loaded `Mailer`, you can send an email with the following:

```
$mailer = new Mailer('default');
$mailer->setFrom(['me@example.com' => 'My Site'])
    ->setTo('you@example.com')
    ->setSubject('About')
    ->deliver('My message');
```

Since `Mailer`'s setter methods return the instance of the class, you are able to set its properties with method chaining.

`Mailer` has several methods for defining recipients - `setTo()`, `setCc()`, `setBcc()`, `addTo()`, `addCc()` and `addBcc()`. The main difference being that the first three will overwrite what was already set and the latter will just add more recipients to their respective field:

```
$mailer = new Mailer();
$mailer->setTo('to@example.com', 'To Example');
$mailer->addTo('to2@example.com', 'To2 Example');
// The email's To recipients are: to@example.com and to2@example.com
```

(continues on next page)

(continued from previous page)

```
$mailer->setTo('test@example.com', 'ToTest Example');
// The email's To recipient is: test@example.com
```

Choosing the Sender

When sending email on behalf of other people, it's often a good idea to define the original sender using the Sender header. You can do so using `setSender()`:

```
$mailer = new Mailer();
$mailer->setSender('app@example.com', 'MyApp emailer');
```

Note: It's also a good idea to set the envelope sender when sending mail on another person's behalf. This prevents them from getting any messages about deliverability.

Configuration

Mailer profiles and email transport settings are defined in your application's configuration files. The `Email` and `EmailTransport` keys define mailer profiles and email transport configurations respectively. During application bootstrap configuration settings are passed from `Configure` into the `Mailer` and `TransportFactory` classes using `setConfig()`. By defining profiles and transports, you can keep your application code free of configuration data, and avoid duplication that makes maintenance and deployment more difficult.

To load a predefined configuration, you can use the `setProfile()` method or pass it to the constructor of `Mailer`:

```
$mailer = new Mailer();
$mailer->setProfile('default');

// Or in constructor
$mailer = new Mailer('default');
```

Instead of passing a string which matches a preset configuration name, you can also just load an array of options:

```
$mailer = new Mailer();
$mailer->setProfile(['from' => 'me@example.org', 'transport' => 'my_custom']);

// Or in constructor
$mailer = new Mailer(['from' => 'me@example.org', 'transport' => 'my_custom']);
```

Configuration Profiles

Defining delivery profiles allows you to consolidate common email settings into re-usable profiles. Your application can have as many profiles as necessary. The following configuration keys are used:

- `'from'`: Mailer or array of sender. See `Mailer::setFrom()`.
- `'sender'`: Mailer or array of real sender. See `Mailer::setSender()`.
- `'to'`: Mailer or array of destination. See `Mailer::setTo()`.

- `'cc'`: Mailer or array of carbon copy. See `Mailer::setCc()`.
- `'bcc'`: Mailer or array of blind carbon copy. See `Mailer::setBcc()`.
- `'replyTo'`: Mailer or array to reply the e-mail. See `Mailer::setReplyTo()`.
- `'readReceipt'`: Mailer address or an array of addresses to receive the receipt of read. See `Mailer::setReadReceipt()`.
- `'returnPath'`: Mailer address or an array of addresses to return if have some error. See `Mailer::setReturnPath()`.
- `'messageId'`: Message ID of e-mail. See `Mailer::setMessageId()`.
- `'subject'`: Subject of the message. See `Mailer::setSubject()`.
- `'message'`: Content of message. Do not set this field if you are using rendered content.
- `'priority'`: Priority of the email as numeric value (usually from 1 to 5 with 1 being the highest).
- `'headers'`: Headers to be included. See `Mailer::setHeaders()`.
- `'viewRenderer'`: If you are using rendered content, set the view classname. See `ViewBuilder::setClassName()`.
- `'template'`: If you are using rendered content, set the template name. See `ViewBuilder::setTemplate()`.
- `'theme'`: Theme used when rendering template. See `ViewBuilder::setTheme()`.
- `'layout'`: If you are using rendered content, set the layout to render. See `ViewBuilder::setTemplate()`.
- `'autoLayout'`: If you want to render a template without layout, set this field to `false`. See `ViewBuilder::disableAutoLayout()`.
- `'viewVars'`: If you are using rendered content, set the array with variables to be used in the view. See `Mailer::setViewVars()`.
- `'attachments'`: List of files to attach. See `Mailer::setAttachments()`.
- `'emailFormat'`: Format of email (html, text or both). See `Mailer::setEmailFormat()`.
- `'transport'`: Transport configuration name. See *Configuring Transports*.
- `'log'`: Log level to log the email headers and message. `true` will use LOG_DEBUG. See *Using Levels*. Note that logs will be emitted under the scope named `email`. See also *Logging Scopes*.
- `'helpers'`: Array of helpers used in the email template. `ViewBuilder::setHelpers()`/`ViewBuilder::addHelpers()`.

Note: The values of above keys using Mailer or array, like from, to, cc, etc will be passed as first parameter of corresponding methods. The equivalent for: `$mailer->setFrom('my@example.com', 'My Site')` would be defined as `'from' => ['my@example.com' => 'My Site']` in your config

Setting Headers

In `Mailer` you are free to set whatever headers you want. Do not forget to put the `X-` prefix for your custom headers.

See `Mailer::setHeaders()` and `Mailer::addHeaders()`

Sending Templated Emails

Emails are often much more than just a simple text message. In order to facilitate that, CakePHP provides a way to send emails using CakePHP's *view layer*.

The templates for emails reside in a special folder `templates/email` of your application. Mailer views can also use layouts and elements just like normal views:

```
$mailer = new Mailer();
$mailer
            ->setEmailFormat('html')
            ->setTo('bob@example.com')
            ->setFrom('app@domain.com')
            ->viewBuilder()
                ->setTemplate('welcome')
                ->setLayout('fancy');

$mailer->deliver();
```

The above would use **templates/email/html/welcome.php** for the view and **templates/layout/email/html/fancy.php** for the layout. You can send multipart templated email messages as well:

```
$mailer = new Mailer();
$mailer
            ->setEmailFormat('both')
            ->setTo('bob@example.com')
            ->setFrom('app@domain.com')
            ->viewBuilder()
                ->setTemplate('welcome')
                ->setLayout('fancy');

$mailer->deliver();
```

This would use the following template files:

- **templates/email/text/welcome.php**
- **templates/layout/email/text/fancy.php**
- **templates/email/html/welcome.php**
- **templates/layout/email/html/fancy.php**

When sending templated emails you have the option of sending either `text`, `html` or `both`.

You can set all view related config using the view builder instance got by `Mailer::viewBuilder()` similar to how you do the same in controller.

You can set view variables with `Mailer::setViewVars()`:

```
$mailer = new Mailer('templated');
$mailer->setViewVars(['value' => 12345]);
```

Or you can use the view builder methods `ViewBuilder::setVar()` and `ViewBuilder::setVars()`.

In your email templates you can use these with:

```
<p>Here is your value: <b><?= $value ?></b></p>
```

You can use helpers in emails as well, much like you can in normal template files. By default only the `HtmlHelper` is loaded. You can load additional helpers using the `ViewBuilder::addHelpers()` method:

```
$mailer->viewBuilder()->addHelpers(['Html', 'Custom', 'Text']);
```

When adding helpers be sure to include 'Html' or it will be removed from the helpers loaded in your email template.

Note: In versions prior to 4.3.0, you will need to use `setHelpers()` instead.

If you want to send email using templates in a plugin you can use the familiar *plugin syntax* to do so:

```
$mailer = new Mailer();
$mailer->viewBuilder()->setTemplate('Blog.new_comment');
```

The above would use template and layout from the Blog plugin as an example.

In some cases, you might need to override the default template provided by plugins. You can do this using themes:

```
$mailer->viewBuilder()
    ->setTemplate('Blog.new_comment')
    ->setLayout('Blog.auto_message')
    ->setTheme('TestTheme');
```

This allows you to override the `new_comment` template in your theme without modifying the Blog plugin. The template file needs to be created in the following path: **templates/plugin/TestTheme/plugin/Blog/email/text/new_comment.php**.

Sending Attachments

`Cake\Mailer\Mailer::setAttachments`(*$attachments*)

You can attach files to email messages as well. There are a few different formats depending on what kind of files you have, and how you want the filenames to appear in the recipient's mail client:

1. Array: `$mailer->setAttachments(['/full/file/path/file.png'])` will attach this file with the name file.png..
2. Array with key: `$mailer->setAttachments(['photo.png' => '/full/some_hash.png'])` will attach some_hash.png with the name photo.png. The recipient will see photo.png, not some_hash.png.
3. Nested arrays:

```
$mailer->setAttachments([
    'photo.png' => [
        'file' => '/full/some_hash.png',
```

(continues on next page)

(continued from previous page)

```
        'mimetype' => 'image/png',
        'contentId' => 'my-unique-id',
    ],
]);
```

The above will attach the file with different mimetype and with custom Content ID (when set the content ID the attachment is transformed to inline). The mimetype and contentId are optional in this form.

3.1. When you are using the `contentId`, you can use the file in the HTML body like `<img src="cid:my-content-id">`.

3.2. You can use the `contentDisposition` option to disable the `Content-Disposition` header for an attachment. This is useful when sending ical invites to clients using outlook.

3.3 Instead of the `file` option you can provide the file contents as a string using the `data` option. This allows you to attach files without needing file paths to them.

Relaxing Address Validation Rules

`Cake\Mailer\Mailer::setEmailPattern(`*$pattern*`)`

If you are having validation issues when sending to non-compliant addresses, you can relax the pattern used to validate email addresses. This is sometimes necessary when dealing with some ISPs:

```
$mailer = new Mailer('default');

// Relax the email pattern, so you can send
// to non-conformant addresses.
$mailer->setEmailPattern($newPattern);
```

Sending Emails from CLI

When sending emails within a CLI script (Shells, Tasks, ...) you should manually set the domain name for Mailer to use. It will serve as the host name for the message id (since there is no host name in a CLI environment):

```
$mailer->setDomain('www.example.org');
// Results in message ids like ``<UUID@www.example.org>`` (valid)
// Instead of `<UUID@>`` (invalid)
```

A valid message id can help to prevent emails ending up in spam folders.

Creating Reusable Emails

Until now we have seen how to directly use the the `Mailer` class to create and send one emails. But main feature of mailer is to allow creating reusable emails throughout your application. They can also be used to contain multiple email configurations in one location. This helps keep your code DRYer and keeps email configuration noise out of other areas in your application.

In this example we will be creating a `Mailer` that contains user-related emails. To create our `UserMailer`, create the file **src/Mailer/UserMailer.php**. The contents of the file should look like the following:

```php
namespace App\Mailer;

use Cake\Mailer\Mailer;

class UserMailer extends Mailer
{
    public function welcome($user)
    {
        $this
            ->setTo($user->email)
            ->setSubject(sprintf('Welcome %s', $user->name))
            ->viewBuilder()
                ->setTemplate('welcome_mail'); // By default template with same name as␣
↪method name is used.
    }

    public function resetPassword($user)
    {
        $this
            ->setTo($user->email)
            ->setSubject('Reset password')
            ->setViewVars(['token' => $user->token]);
    }
}
```

In our example we have created two methods, one for sending a welcome email, and another for sending a password reset email. Each of these methods expect a user `Entity` and utilizes its properties for configuring each email.

We are now able to use our `UserMailer` to send out our user-related emails from anywhere in our application. For example, if we wanted to send our welcome email we could do the following:

```php
namespace App\Controller;

use Cake\Mailer\MailerAwareTrait;

class UsersController extends AppController
{
    use MailerAwareTrait;

    public function register()
    {
        $user = $this->Users->newEmptyEntity();
        if ($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);
    }
}
```

If we wanted to completely separate sending a user their welcome email from our application's code, we can have our `UserMailer` subscribe to the `Model.afterSave` event. By subscribing to an event, we can keep our application's

user-related classes completely free of email-related logic and instructions. For example, we could add the following to our `UserMailer`:

```
public function implementedEvents()
{
    return [
        'Model.afterSave' => 'onRegistration',
    ];
}

public function onRegistration(EventInterface $event, EntityInterface $entity,␣
↪ArrayObject $options)
{
    if ($entity->isNew()) {
        $this->send('welcome', [$entity]);
    }
}
```

You can now register the mailer as an event listener and the `onRegistration()` method will be invoked every time the `Model.afterSave` event is fired:

```
// attach to Users event manager
$this->Users->getEventManager()->on($this->getMailer('User'));
```

Note: For information on how to register event listener objects, please refer to the *Registering Listeners* documentation.

Configuring Transports

Email messages are delivered by transports. Different transports allow you to send messages via PHP's `mail()` function, SMTP servers, or not at all which is useful for debugging. Configuring transports allows you to keep configuration data out of your application code and makes deployment simpler as you can simply change the configuration data. An example transport configuration looks like:

```
// In config/app.php
'EmailTransport' => [
    // Sample Mail configuration
    'default' => [
        'className' => 'Mail',
    ],
    // Sample SMTP configuration
    'gmail' => [
        'host' => 'smtp.gmail.com',
        'port' => 587,
        'username' => 'my@gmail.com',
        'password' => 'secret',
        'className' => 'Smtp',
        'tls' => true,
    ],
],
```

Transports can also be configured at runtime using `TransportFactory::setConfig()`:

```
use Cake\Mailer\TransportFactory;

// Define an SMTP transport
TransportFactory::setConfig('gmail', [
    'host' => 'ssl://smtp.gmail.com',
    'port' => 465,
    'username' => 'my@gmail.com',
    'password' => 'secret',
    'className' => 'Smtp'
]);
```

You can configure SSL SMTP servers, like Gmail. To do so, put the `ssl://` prefix in the host and configure the port value accordingly. You can also enable TLS SMTP using the `tls` option:

```
use Cake\Mailer\TransportFactory;

TransportFactory::setConfig('gmail', [
    'host' => 'smtp.gmail.com',
    'port' => 587,
    'username' => 'my@gmail.com',
    'password' => 'secret',
    'className' => 'Smtp',
    'tls' => true
]);
```

The above configuration would enable TLS communication for email messages.

To configure your mailer to use a specific transport you can use `Cake\Mailer\Mailer::setTransport()` method or have the transport in your configuration:

```
// Use a named transport already configured using TransportFactory::setConfig()
$mailer->setTransport('gmail');

// Use a constructed object.
$mailer->setTransport(new \Cake\Mailer\Transport\DebugTransport());
```

Warning: You will need to have access for less secure apps enabled in your Google account for this to work: Allowing less secure apps to access your account[139].

Note: Gmail SMTP settings[140].

Note: To use SSL + SMTP, you will need to have the SSL configured in your PHP install.

Configuration options can also be provided as a *DSN* string. This is useful when working with environment variables or *PaaS* providers:

[139] https://support.google.com/accounts/answer/6010255
[140] https://support.google.com/a/answer/176600?hl=en

```
TransportFactory::setConfig('default', [
    'url' => 'smtp://my@gmail.com:secret@smtp.gmail.com:587?tls=true',
]);
```

When using a DSN string you can define any additional parameters/options as query string arguments.

static Cake\Mailer\Mailer::**drop**(*$key*)

Once configured, transports cannot be modified. In order to modify a transport you must first drop it and then reconfigure it.

Creating Custom Transports

You are able to create your custom transports for situations such as send email using services like SendGrid, MailGun or Postmark. To create your transport, first create the file **src/Mailer/Transport/ExampleTransport.php** (where Example is the name of your transport). To start, your file should look like:

```
namespace App\Mailer\Transport;

use Cake\Mailer\AbstractTransport;
use Cake\Mailer\Message;

class ExampleTransport extends AbstractTransport
{
    public function send(Message $message): array
    {
        // Do something.
    }
}
```

You must implement the method `send(Message $message)` with your custom logic.

Sending emails without using Mailer

The `Mailer` is a higher level abstraction class which acts as a bridge between the `Cake\Mailer\Message`, `Cake\Mailer\Renderer` and `Cake\Mailer\AbstractTransport` classes to configure emails with a fluent interface.

If you want you can use these classes directly with the `Mailer` too.

For example:

```
$render = new \Cake\Mailer\Renderer();
$render->viewBuilder()
    ->setTemplate('custom')
    ->setLayout('sparkly');

$message = new \Cake\Mailer\Message();
$message
    ->setFrom('admin@cakephp.org')
    ->setTo('user@foo.com')
    ->setBody($render->render());
```

(continues on next page)

(continued from previous page)

```
$transport = new \Cake\Mailer\Transport\MailTransport();
$result = $transport->send($message);
```

You can even skip using the `Renderer` and set the message body directly using `Message::setBodyText()` and `Message::setBodyHtml()` methods.

Testing Mailers

To test mailers, add `Cake\TestSuite\EmailTrait` to your test case. The `MailerTrait` uses PHPUnit hooks to replace your application's email transports with a proxy that intercepts email messages and allows you to do assertions on the mail that would be delivered.

Add the trait to your test case to start testing emails, and load routes if your emails need to generate URLs:

```
namespace App\Test\TestCase\Mailer;

use App\Mailer\WelcomeMailer;
use App\Model\Entity\User;

use Cake\TestSuite\EmailTrait;
use Cake\TestSuite\TestCase;

class WelcomeMailerTestCase extends TestCase
{
    use EmailTrait;

    public function setUp(): void
    {
        parent::setUp();
        $this->loadRoutes();
    }
}
```

Let's assume we have a mailer that delivers welcome emails when a new user registers. We want to check that the subject and body contain the user's name:

```
// in our WelcomeMailerTestCase class.
public function testName()
{
    $user = new User([
        'name' => 'Alice Alittea',
        'email' => 'alice@example.org',
    ]);
    $mailer = new WelcomeMailer();
    $mailer->send('welcome', [$user]);

    $this->assertMailSentTo($user->email);
    $this->assertMailContainsText('Hi ' . $user->name);
    $this->assertMailContainsText('Welcome to CakePHP!');
}
```

Assertion methods

The `Cake\TestSuite\EmailTrait` trait provides the following assertions:

```
// Asserts an expected number of emails were sent
$this->assertMailCount($count);

// Asserts that no emails were sent
$this->assertNoMailSent();

// Asserts an email was sent to an address
$this->assertMailSentTo($address);

// Asserts an email was sent from an address
$this->assertMailSentFrom($emailAddress);
$this->assertMailSentFrom([$emailAddress => $displayName]);

// Asserts an email contains expected contents
$this->assertMailContains($contents);

// Asserts an email contains expected html contents
$this->assertMailContainsHtml($contents);

// Asserts an email contains expected text contents
$this->assertMailContainsText($contents);

// Asserts an email contains the expected value within an Message getter (for example,
↪"subject")
$this->assertMailSentWith($expected, $parameter);

// Asserts an email at a specific index was sent to an address
$this->assertMailSentToAt($at, $address);

// Asserts an email at a specific index was sent from an address
$this->assertMailSentFromAt($at, $address);

// Asserts an email at a specific index contains expected contents
$this->assertMailContainsAt($at, $contents);

// Asserts an email at a specific index contains expected html contents
$this->assertMailContainsHtmlAt($at, $contents);

// Asserts an email at a specific index contains expected text contents
$this->assertMailContainsTextAt($at, $contents);

// Asserts an email contains an attachment
$this->assertMailContainsAttachment('test.png');

// Asserts an email at a specific index contains the expected value within an Message␣
↪getter (for example, "cc")
$this->assertMailSentWithAt($at, $expected, $parameter);

// Asserts an email contains a substring in the subject.
```

(continues on next page)

(continued from previous page)

```
$this->assertMailSubjectContains('Free Offer');

// Asserts an email at the specific index contains a substring in the subject.
$this->assertMailSubjectContainsAt(1, 'Free Offer');
```

CHAPTER 19

Error & Exception Handling

CakePHP applications come with error and exception handling setup for you. PHP errors are trapped and displayed or logged. Uncaught exceptions are rendered into error pages automatically.

Configuration

Error configuration is done in your application's **config/app.php** file. By default CakePHP uses `Cake\Error\ErrorTrap` and `Cake\Error\ExceptionTrap` to handle both PHP errors and exceptions respectively. The error configuration allows you to customize error handling for your application. The following options are supported:

- `errorLevel` - int - The level of errors you are interested in capturing. Use the built-in PHP error constants, and bitmasks to select the level of error you are interested in. See *Deprecation Warnings* to disable deprecation warnings.
- `trace` - bool - Include stack traces for errors in log files. Stack traces will be included in the log after each error. This is helpful for finding where/when errors are being raised.
- `exceptionRenderer` - string - The class responsible for rendering uncaught exceptions. If you choose a custom class you should place the file for that class in **src/Error**. This class needs to implement a `render()` method.
- `log` - bool - When `true`, exceptions + their stack traces will be logged to *`Cake\Log\Log`*.
- `skipLog` - array - An array of exception classnames that should not be logged. This is useful to remove NotFoundExceptions or other common, but uninteresting log messages.
- `extraFatalErrorMemory` - int - Set to the number of megabytes to increase the memory limit by when a fatal error is encountered. This allows breathing room to complete logging or error handling.
- `logger` (prior to 4.4.0 use `errorLogger`) - `Cake\Error\ErrorLoggerInterface` - The class responsible for logging errors and unhandled exceptions. Defaults to `Cake\Error\ErrorLogger`.
- `errorRenderer` - `Cake\Error\ErrorRendererInterface` - The class responsible for rendering errors. Default is chosen based on PHP SAPI.

- `ignoredDeprecationPaths` - array - A list of glob compatible paths that deprecation errors should be ignored in. Added in 4.2.0

By default, PHP errors are displayed when `debug` is `true`, and logged when debug is `false`. The fatal error handler will be called independent of `debug` level or `errorLevel` configuration, but the result will be different based on `debug` level. The default behavior for fatal errors is show a page to internal server error (`debug` disabled) or a page with the message, file and line (`debug` enabled).

Note: If you use a custom error handler, the supported options will depend on your handler.

Deprecation Warnings

CakePHP uses deprecation warnings to indicate when features have been deprecated. We also recommend this system for use in your plugins and application code when useful. You can trigger deprecation warnings with `deprecationWarning()`:

```
deprecationWarning('5.0', 'The example() method is deprecated. Use getExample() instead.
↪');
```

When upgrading CakePHP or plugins you may encounter new deprecation warnings. You can temporarily disable deprecation warnings in one of a few ways:

1. Using the `Error.errorLevel` setting to `E_ALL ^ E_USER_DEPRECATED` to ignore *all* deprecation warnings.
2. Using the `Error.ignoredDeprecationPaths` configuration option to ignore deprecations with glob compatible expressions. For example:

   ```
   'Error' => [
       'ignoredDeprecationPaths' => [
           'vendors/company/contacts/*',
           'src/Models/*',
       ],
   ],
   ```

   Would ignore all deprecations from your `Models` directory and the `Contacts` plugin in your application.

Changing Exception Handling

Exception handling in CakePHP offers several ways to tailor how exceptions are handled. Each approach gives you different amounts of control over the exception handling process.

1. *Listen to events* This allows you to be notified through CakePHP events when errors and exceptions have been handled.
2. *Custom templates* This allows you to change the rendered view templates as you would any other template in your application.
3. *Custom Controller* This allows you to control how exception pages are rendered.
4. *Custom ExceptionRenderer* This allows you to control how exception pages and logging are performed.
5. *Create & register your own traps* This gives you complete control over how errors & exceptions are handled, logged and rendered. Use `Cake\Error\ExceptionTrap` and `Cake\Error\ErrorTrap` as reference when implementing your traps.

Listen to Events

The `ErrorTrap` and `ExceptionTrap` handlers will trigger CakePHP events when they handle errors. You can listen to the `Error.beforeRender` event to be notified of PHP errors. The `Exception.beforeRender` event is dispatched when an exception is handled:

```
$errorTrap = new ErrorTrap(Configure::read('Error'));
$errorTrap->getEventManager()->on(
    'Error.beforeRender',
    function (EventInterface $event, PhpError $error) {
        // do your thing
    }
);
```

Within an `Error.beforeRender` handler you have a few options:

- Stop the event to prevent rendering.
- Return a string to skip rendering and use the provided string instead

Within an `Exception.beforeRender` handler you have a few options:

- Stop the event to prevent rendering.
- Set the `exception` data attribute with `setData('exception', $err)` to replace the exception that is being rendered.
- Return a response from the event listener to skip rendering and use the provided response instead.

Custom Templates

The default exception trap renders all uncaught exceptions your application raises with the help of `Cake\Error\Renderer\WebExceptionRenderer`, and your application's `ErrorController`.

The error page views are located at **templates/Error/**. All 4xx errors use the **error400.php** template, and 5xx errors use the **error500.php**. Your error templates will have the following variables available:

- `message` The exception message.
- `code` The exception code.
- `url` The request URL.
- `error` The exception object.

In debug mode if your error extends `Cake\Core\Exception\CakeException` the data returned by `getAttributes()` will be exposed as view variables as well.

Note: You will need to set `debug` to false, to see your **error404** and **error500** templates. In debug mode, you'll see CakePHP's development error page.

Custom Error Page Layout

By default error templates use **templates/layout/error.php** for a layout. You can use the `layout` property to pick a different layout:

```
// inside templates/Error/error400.php
$this->layout = 'my_error';
```

The above would use **templates/layout/my_error.php** as the layout for your error pages.

Many exceptions raised by CakePHP will render specific view templates in debug mode. With debug turned off all exceptions raised by CakePHP will use either **error400.php** or **error500.php** based on their status code.

Custom Controller

The `App\Controller\ErrorController` class is used by CakePHP's exception rendering to render the error page view and receives all the standard request life-cycle events. By modifying this class you can control which components are used and which templates are rendered.

If your application uses *Prefix Routing* you can create custom error controllers for each routing prefix. For example, if you had an `Admin` prefix. You could create the following class:

```
namespace App\Controller\Admin;

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

class ErrorController extends AppController
{
    /**
     * beforeRender callback.
     *
     * @param \Cake\Event\EventInterface $event Event.
     * @return void
     */
    public function beforeRender(EventInterface $event)
    {
        $this->viewBuilder()->setTemplatePath('Error');
    }
}
```

This controller would only be used when an error is encountered in a prefixed controller, and allows you to define prefix specific logic/templates as needed.

Custom ExceptionRenderer

If you want to control the entire exception rendering and logging process you can use the `Error.exceptionRenderer` option in **config/app.php** to choose a class that will render exception pages. Changing the ExceptionRenderer is useful when you want to change the logic used to create an error controller, choose the template, or control the overall rendering process.

Your custom exception renderer class should be placed in **src/Error**. Let's assume our application uses `App\Exception\MissingWidgetException` to indicate a missing widget. We could create an exception renderer that renders specific error pages when this error is handled:

```
// In src/Error/AppExceptionRenderer.php
namespace App\Error;

use Cake\Error\Renderer\WebExceptionRenderer;

class AppExceptionRenderer extends WebExceptionRenderer
{
    public function missingWidget($error)
    {
        $response = $this->controller->getResponse();

        return $response->withStringBody('Oops that widget is missing.');
    }
}

// In config/app.php
'Error' => [
    'exceptionRenderer' => 'App\Error\AppExceptionRenderer',
    // ...
],
// ...
```

The above would handle our `MissingWidgetException`, and allow us to provide custom display/handling logic for those application exceptions.

Exception rendering methods receive the handled exception as an argument, and should return a `Response` object. You can also implement methods to add additional logic when handling CakePHP errors:

```
// In src/Error/AppExceptionRenderer.php
namespace App\Error;

use Cake\Error\Renderer\WebExceptionRenderer;

class AppExceptionRenderer extends WebExceptionRenderer
{
    public function notFound($error)
    {
        // Do something with NotFoundException objects.
    }
}
```

Changing the ErrorController Class

The exception renderer dictates which controller is used for exception rendering. If you want to change which controller is used to render exceptions, override the `_getController()` method in your exception renderer:

```
// in src/Error/AppExceptionRenderer
namespace App\Error;

use App\Controller\SuperCustomErrorController;
use Cake\Controller\Controller;
use Cake\Error\Renderer\WebExceptionRenderer;

class AppExceptionRenderer extends WebExceptionRenderer
{
    protected function _getController(): Controller
    {
        return new SuperCustomErrorController();
    }
}

// in config/app.php
'Error' => [
    'exceptionRenderer' => 'App\Error\AppExceptionRenderer',
    // ...
],
// ...
```

Creating your own Application Exceptions

You can create your own application exceptions using any of the built in SPL exceptions[141], `Exception` itself, or *`Cake\Core\Exception\Exception`*. If your application contained the following exception:

```
use Cake\Core\Exception\CakeException;

class MissingWidgetException extends CakeException
{
}
```

You could provide nice development errors, by creating **templates/Error/missing_widget.php**. When in production mode, the above error would be treated as a 500 error and use the **error500** template.

Exceptions that subclass `Cake\Http\Exception\HttpException`, will have their error code used as an HTTP status code if the error code is between `400` and `506`.

The constructor for `Cake\Core\Exception\CakeException` allows you to pass in additional data. This additional data is interpolated into the the `_messageTemplate`. This allows you to create data rich exceptions, that provide more context around your errors:

```
use Cake\Core\Exception\CakeException;

class MissingWidgetException extends Exception
```

(continues on next page)

[141] https://php.net/manual/en/spl.exceptions.php

(continued from previous page)

```
{
    // Context data is interpolated into this format string.
    protected $_messageTemplate = 'Seems that %s is missing.';

    // You can set a default exception code as well.
    protected $_defaultCode = 404;
}

throw new MissingWidgetException(['widget' => 'Pointy']);
```

When rendered, this your view template would have a `$widget` variable set. If you cast the exception as a string or use its `getMessage()` method you will get `Seems that Pointy is missing.`.

Note: Prior to CakePHP 4.2.0 use class `Cake\Core\Exception\Exception` instead of `Cake\Core\Exception\CakeException`

Logging Exceptions

Using the built-in exception handling, you can log all the exceptions that are dealt with by ErrorTrap by setting the `log` option to `true` in your **config/app.php**. Enabling this will log every exception to *Cake\Log\Log* and the configured loggers.

Note: If you are using a custom exception handler this setting will have no effect. Unless you reference it inside your implementation.

Built in Exceptions for CakePHP

HTTP Exceptions

There are several built-in exceptions inside CakePHP, outside of the internal framework exceptions, there are several exceptions for HTTP methods

exception `Cake\Http\Exception\`**BadRequestException**

Used for doing 400 Bad Request error.

exception `Cake\Http\Exception\`**UnauthorizedException**

Used for doing a 401 Unauthorized error.

exception `Cake\Http\Exception\`**ForbiddenException**

Used for doing a 403 Forbidden error.

exception `Cake\Http\Exception\`**InvalidCsrfTokenException**

Used for doing a 403 error caused by an invalid CSRF token.

exception Cake\Http\Exception**NotFoundException**

Used for doing a 404 Not found error.

exception Cake\Http\Exception**MethodNotAllowedException**

Used for doing a 405 Method Not Allowed error.

exception Cake\Http\Exception**NotAcceptableException**

Used for doing a 406 Not Acceptable error.

exception Cake\Http\Exception**ConflictException**

Used for doing a 409 Conflict error.

exception Cake\Http\Exception**GoneException**

Used for doing a 410 Gone error.

For more details on HTTP 4xx error status codes see **RFC 2616#section-10.4**[142].

exception Cake\Http\Exception**InternalErrorException**

Used for doing a 500 Internal Server Error.

exception Cake\Http\Exception**NotImplementedException**

Used for doing a 501 Not Implemented Errors.

exception Cake\Http\Exception**ServiceUnavailableException**

Used for doing a 503 Service Unavailable error.

For more details on HTTP 5xx error status codes see **RFC 2616#section-10.5**[143].

You can throw these exceptions from your controllers to indicate failure states, or HTTP errors. An example use of the HTTP exceptions could be rendering 404 pages for items that have not been found:

```
use Cake\Http\Exception\NotFoundException;

public function view($id = null)
{
    $article = $this->Articles->findById($id)->first();
    if (empty($article)) {
        throw new NotFoundException(__('Article not found'));
    }
    $this->set('article', $article);
    $this->viewBuilder()->setOption('serialize', ['article']);
}
```

By using exceptions for HTTP errors, you can keep your code both clean, and give RESTful responses to client applications and users.

[142] https://datatracker.ietf.org/doc/html/rfc2616.html#section-10.4
[143] https://datatracker.ietf.org/doc/html/rfc2616.html#section-10.5

Using HTTP Exceptions in your Controllers

You can throw any of the HTTP related exceptions from your controller actions to indicate failure states. For example:

```
use Cake\Network\Exception\NotFoundException;

public function view($id = null)
{
    $article = $this->Articles->findById($id)->first();
    if (empty($article)) {
        throw new NotFoundException(__('Article not found'));
    }
    $this->set('article', 'article');
    $this->viewBuilder()->setOption('serialize', ['article']);
}
```

The above would cause the configured exception handler to catch and process the *NotFoundException*. By default this will create an error page, and log the exception.

Other Built In Exceptions

In addition, CakePHP uses the following exceptions:

exception Cake\View\Exception**MissingViewException**

The chosen view class could not be found.

exception Cake\View\Exception**MissingTemplateException**

The chosen template file could not be found.

exception Cake\View\Exception**MissingLayoutException**

The chosen layout could not be found.

exception Cake\View\Exception**MissingHelperException**

The chosen helper could not be found.

exception Cake\View\Exception**MissingElementException**

The chosen element file could not be found.

exception Cake\View\Exception**MissingCellException**

The chosen cell class could not be found.

exception Cake\View\Exception**MissingCellViewException**

The chosen cell view file could not be found.

exception Cake\Controller\Exception**MissingComponentException**

A configured component could not be found.

exception Cake\Controller\Exception**MissingActionException**

The requested controller action could not be found.

exception Cake\Controller\Exception**PrivateActionException**

Accessing private/protected/_ prefixed actions.

exception Cake\Console\Exception**ConsoleException**

A console library class encounter an error.

exception Cake\Database\Exception**MissingConnectionException**

A model's connection is missing.

exception Cake\Database\Exception**MissingDriverException**

A database driver could not be found.

exception Cake\Database\Exception**MissingExtensionException**

A PHP extension is missing for the database driver.

exception Cake\ORM\Exception**MissingTableException**

A model's table could not be found.

exception Cake\ORM\Exception**MissingEntityException**

A model's entity could not be found.

exception Cake\ORM\Exception**MissingBehaviorException**

A model's behavior could not be found.

exception Cake\ORM\Exception**PersistenceFailedException**

An entity couldn't be saved/deleted while using *Cake\ORM\Table::saveOrFail()* or *Cake\ORM\Table::deleteOrFail()*.

exception Cake\Datasource\Exception**RecordNotFoundException**

The requested record could not be found. This will also set HTTP response headers to 404.

exception Cake\Routing\Exception**MissingControllerException**

The requested controller could not be found.

exception Cake\Routing\Exception**MissingRouteException**

The requested URL cannot be reverse routed or cannot be parsed.

exception Cake\Core\Exception**Exception**

Base exception class in CakePHP. All framework layer exceptions thrown by CakePHP will extend this class.

These exception classes all extend *Exception*. By extending Exception, you can create your own 'framework' errors.

Cake\Core\Exception\Exception::**responseHeader**(*$header = null*, *$value = null*)

See Cake\Network\Request::header()

All Http and Cake exceptions extend the Exception class, which has a method to add headers to the response. For instance when throwing a 405 MethodNotAllowedException the rfc2616 says:

```
"The response MUST include an Allow header containing a list of valid
methods for the requested resource."
```

Customizing PHP Error Handling

By default PHP errors are rendered to console or HTML output, and also logged. If necessary, you can swap out CakePHP's error handling logic with your own.

Custom Error Logging

Error handlers use instances of `Cake\Error\ErrorLoggingInterface` to create log messages and log them to the appropriate place. You can replace the error logger using the `Error.errorLogger` configure value. An example error logger:

```
namespace App\Error;

use Cake\Error\ErrorLoggerInterface;
use Cake\Error\PhpError;
use Psr\Http\Message\ServerRequestInterface;
use Throwable;

/**
 * Log errors and unhandled exceptions to `Cake\Log\Log`
 */
class ErrorLogger implements ErrorLoggerInterface
{
    /**
     * @inheritDoc
     */
    public function logError(
        PhpError $error,
        ?ServerRequestInterface $request,
        bool $includeTrace = false
    ): void {
        // Log PHP Errors
    }

    /**
     * @inheritDoc
     */
    public function logException(
        ?ServerRequestInterface $request,
        bool $includeTrace = false
    ): void {
        // Log exceptions.
    }
}
```

Custom Error Rendering

CakePHP includes error renderers for both web and console environments. If however, you would like to replace the logic that renders errors you can create a class:

```
// src/Error/CustomErrorRenderer.php
namespace App\Error;

use Cake\Error\ErrorRendererInterface;
use Cake\Error\PhpError;

class CustomErrorRenderer implements ErrorRendererInterface
{
    public function write(string $out): void
    {
        // output the rendered error to the appropriate output stream
    }

    public function render(PhpError $error, bool $debug): string
    {
        // Convert the error into the output string.
    }
}
```

The constructor of your renderer will be passed an array of all the Error configuration. You connect your custom error renderer to CakePHP via the `Error.errorRenderer` config value. When replacing error handling you will need to account for both web and command line environments.

CHAPTER 20

Events System

Creating maintainable applications is both a science and an art. It is well-known that a key for having good quality code is making your objects loosely coupled and strongly cohesive at the same time. Cohesion means that all methods and properties for a class are strongly related to the class itself and it is not trying to do the job other objects should be doing, while loosely coupling is the measure of how little a class is "wired" to external objects, and how much that class is depending on them.

There are certain cases where you need to cleanly communicate with other parts of an application, without having to hard code dependencies, thus losing cohesion and increasing class coupling. Using the Observer pattern, which allows objects to notify other objects and anonymous listeners about changes is a useful pattern to achieve this goal.

Listeners in the observer pattern can subscribe to events and choose to act upon them if they are relevant. If you have used JavaScript, there is a good chance that you are already familiar with event driven programming.

CakePHP emulates several aspects of how events are triggered and managed in popular JavaScript libraries such as jQuery. In the CakePHP implementation, an event object is dispatched to all listeners. The event object holds information about the event, and provides the ability to stop event propagation at any point. Listeners can register themselves or can delegate this task to other objects and have the chance to alter the state and the event itself for the rest of the callbacks.

The event subsystem is at the heart of Model, Behavior, Controller, View and Helper callbacks. If you've ever used any of them, you are already somewhat familiar with events in CakePHP.

Example Event Usage

Let's suppose you are building a Cart plugin, and you'd like to focus on just handling order logic. You don't really want to include shipping logic, emailing the user or decrementing the item from the stock, but these are important tasks to the people using your plugin. If you were not using events, you may try to implement this by attaching behaviors to models, or adding components to your controllers. Doing so represents a challenge most of the time, since you would have to come up with the code for externally loading those behaviors or attaching hooks to your plugin controllers.

Instead, you can use events to allow you to cleanly separate the concerns of your code and allow additional concerns to hook into your plugin using events. For example, in your Cart plugin you have an Orders model that deals with creating orders. You'd like to notify the rest of the application that an order has been created. To keep your Orders model clean you could use events:

```
// 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('Order.afterPlace', $this, [
                'order' => $order
            ]);
            $this->getEventManager()->dispatch($event);
            return true;
        }
        return false;
    }
}
```

The above code allows you to notify the other parts of the application that an order has been created. You can then do tasks like send email notifications, update stock, log relevant statistics and other tasks in separate objects that focus on those concerns.

Accessing Event Managers

In CakePHP events are triggered against event managers. Event managers are available in every Table, View and Controller using `getEventManager()`:

```
$events = $this->getEventManager();
```

Each model has a separate event manager, while the View and Controller share one. This allows model events to be self contained, and allow components or controllers to act upon events created in the view if necessary.

Global Event Manager

In addition to instance level event managers, CakePHP provides a global event manager that allows you to listen to any event fired in an application. This is useful when attaching listeners to a specific instance might be cumbersome or difficult. The global manager is a singleton instance of `Cake\Event\EventManager`. Listeners attached to the global dispatcher will be fired before instance listeners at the same priority. You can access the global manager using a static method:

```
// In any configuration file or piece of code that executes before the event
use Cake\Event\EventManager;

EventManager::instance()->on(
    'Order.afterPlace',
    $aCallback
);
```

One important thing you should consider is that there are events that will be triggered having the same name but different subjects, so checking it in the event object is usually required in any function that gets attached globally in order to prevent some bugs. Remember that with the flexibility of using the global manager, some additional complexity is incurred.

`Cake\Event\EventManager::dispatch()` method accepts the event object as an argument and notifies all listener and callbacks passing this object along. The listeners will handle all the extra logic around the `afterPlace` event, you can log the time, send emails, update user statistics possibly in separate objects and even delegating it to offline tasks if you have the need.

Tracking Events

To keep a list of events that are fired on a particular `EventManager`, you can enable event tracking. To do so, simply attach an `Cake\Event\EventList` to the manager:

```
EventManager::instance()->setEventList(new EventList());
```

After firing an event on the manager, you can retrieve it from the event list:

```
$eventsFired = EventManager::instance()->getEventList();
$firstEvent = $eventsFired[0];
```

Tracking can be disabled by removing the event list or calling `Cake\Event\EventList::trackEvents(false)`.

Core Events

There are a number of core events within the framework which your application can listen to. Each layer of CakePHP emits events that you can use in your application.

- *ORM/Model events*
- *Controller events*
- *View events*

Registering Listeners

Listeners are the preferred way to register callbacks for an event. This is done by implementing the `Cake\Event\EventListenerInterface` interface in any class you wish to register some callbacks. Classes implementing it need to provide the `implementedEvents()` method. This method must return an associative array with all event names that the class will handle.

To continue our previous example, let's imagine we have a UserStatistic class responsible for calculating a user's purchasing history, and compiling into global site statistics. This is a great place to use a listener class. Doing so allows you to concentrate the statistics logic in one place and react to events as necessary. Our `UserStatistics` listener might start out like:

```
namespace App\Event;

use Cake\Event\EventListenerInterface;

class UserStatistic implements EventListenerInterface
{
    public function implementedEvents(): array
    {
        return [
            // Custom event names let you design your application events
            // as required.
            'Order.afterPlace' => 'updateBuyStatistic',
        ];
    }

    public function updateBuyStatistic($event)
    {
        // Code to update statistics
    }
}

// From your controller, attach the UserStatistic object to the Order's event manager
$statistics = new UserStatistic();
$this->Orders->getEventManager()->on($statistics);
```

As you can see in the above code, the `on()` function will accept instances of the `EventListener` interface. Internally, the event manager will use `implementedEvents()` to attach the correct callbacks.

Registering Anonymous Listeners

While event listener objects are generally a better way to implement listeners, you can also bind any `callable` as an event listener. For example if we wanted to put any orders into the log files, we could use a simple anonymous function to do so:

```
use Cake\Log\Log;

// From within a controller, or during application bootstrap.
$this->Orders->getEventManager()->on('Order.afterPlace', function ($event) {
    Log::write(
        'info',
        'A new order was placed with id: ' . $event->getSubject()->id
```

(continues on next page)

(continued from previous page)

```
    );
});
```

In addition to anonymous functions you can use any other callable type that PHP supports:

```
$events = [
    'email-sending' => 'EmailSender::sendBuyEmail',
    'inventory' => [$this->InventoryManager, 'decrement'],
];
foreach ($events as $callable) {
    $eventManager->on('Order.afterPlace', $callable);
}
```

When working with plugins that don't trigger specific events, you can leverage event listeners on the default events. Lets take an example 'UserFeedback' plugin which handles feedback forms from users. From your application you would like to know when a Feedback record has been saved and ultimately act on it. You can listen to the global `Model.afterSave` event. However, you can take a more direct approach and only listen to the event you really need:

```
// You can create the following before the
// save operation, ie. config/bootstrap.php
use Cake\Datasource\FactoryLocator;
// If sending emails
use Cake\Mailer\Email;

FactoryLocator::get('Table')->get('ThirdPartyPlugin.Feedbacks')
    ->getEventManager()
    ->on('Model.afterSave', function($event, $entity)
    {
        // For example we can send an email to the admin
        $email = new Email('default');
        $email->setFrom(['info@yoursite.com' => 'Your Site'])
            ->setTo('admin@yoursite.com')
            ->setSubject('New Feedback - Your Site')
            ->send('Body of message');
    });
```

You can use this same approach to bind listener objects.

Interacting with Existing Listeners

Assuming several event listeners have been registered the presence or absence of a particular event pattern can be used as the basis of some action.:

```
// Attach listeners to EventManager.
$this->getEventManager()->on('User.Registration', [$this, 'userRegistration']);
$this->getEventManager()->on('User.Verification', [$this, 'userVerification']);
$this->getEventManager()->on('User.Authorization', [$this, 'userAuthorization']);

// Somewhere else in your application.
$events = $this->getEventManager()->matchingListeners('Verification');
if (!empty($events)) {
    // Perform logic related to presence of 'Verification' event listener.
```

(continues on next page)

(continued from previous page)

```
    // For example removing the listener if present.
    $this->getEventManager()->off('User.Verification');
} else {
    // Perform logic related to absence of 'Verification' event listener
}
```

Note: The pattern passed to the `matchingListeners` method is case sensitive.

Establishing Priorities

In some cases you might want to control the order that listeners are invoked. For instance, if we go back to our user statistics example. It would be ideal if this listener was called at the end of the stack. By calling it at the end of the listener stack, we can ensure that the event was not cancelled, and that no other listeners raised exceptions. We can also get the final state of the objects in the case that other listeners have modified the subject or event object.

Priorities are defined as an integer when adding a listener. The higher the number, the later the method will be fired. The default priority for all listeners is `10`. If you need your method to be run earlier, using any value below this default will work. On the other hand if you desire to run the callback after the others, using a number above `10` will do.

If two callbacks happen to have the same priority value, they will be executed with a the order they were attached. You set priorities using the `on()` method for callbacks, and declaring it in the `implementedEvents()` function for event listeners:

```
// Setting priority for a callback
$callback = [$this, 'doSomething'];
$this->getEventManager()->on(
    'Order.afterPlace',
    ['priority' => 2],
    $callback
);

// Setting priority for a listener
class UserStatistic implements EventListenerInterface
{
    public function implementedEvents()
    {
        return [
            'Order.afterPlace' => [
                'callable' => 'updateBuyStatistic',
                'priority' => 100
            ],
        ];
    }
}
```

As you see, the main difference for `EventListener` objects is that you need to use an array for specifying the callable method and the priority preference. The `callable` key is a special array entry that the manager will read to know what function in the class it should be calling.

Getting Event Data as Function Parameters

When events have data provided in their constructor, the provided data is converted into arguments for the listeners. An example from the View layer is the afterRender callback:

```
$this->getEventManager()
    ->dispatch(new Event('View.afterRender', $this, ['view' => $viewFileName]));
```

The listeners of the `View.afterRender` callback should have the following signature:

```
function (EventInterface $event, $viewFileName)
```

Each value provided to the Event constructor will be converted into function parameters in the order they appear in the data array. If you use an associative array, the result of `array_values` will determine the function argument order.

Note: Unlike in 2.x, converting event data to listener arguments is the default behavior and cannot be disabled.

Dispatching Events

Once you have obtained an instance of an event manager you can dispatch events using `dispatch()`. This method takes an instance of the `Cake\Event\Event` class. Let's look at dispatching an event:

```
// An event listener has to be instantiated before dispatching an event.
// Create a new event and dispatch it.
$event = new Event('Order.afterPlace', $this, [
    'order' => $order
]);
$this->getEventManager()->dispatch($event);
```

`Cake\Event\Event` accepts 3 arguments in its constructor. The first one is the event name, you should try to keep this name as unique as possible, while making it readable. We suggest a convention as follows: `Layer.eventName` for general events happening at a layer level (for example, `Controller.startup`, `View.beforeRender`) and `Layer.Class.eventName` for events happening in specific classes on a layer, for example `Model.User.afterRegister` or `Controller.Courses.invalidAccess`.

The second argument is the `subject`, meaning the object associated to the event, usually when it is the same class triggering events about itself, using `$this` will be the most common case. Although a Component could trigger controller events too. The subject class is important because listeners will get immediate access to the object properties and have the chance to inspect or change them on the fly.

Finally, the third argument is any additional event data. This can be any data you consider useful to pass around so listeners can act upon it. While this can be an argument of any type, we recommend passing an associative array.

The `dispatch()` method accepts an event object as an argument and notifies all subscribed listeners.

Stopping Events

Much like DOM events, you may want to stop an event to prevent additional listeners from being notified. You can see this in action during model callbacks (for example, beforeSave) in which it is possible to stop the saving operation if the code detects it cannot proceed any further.

In order to stop events you can either return `false` in your callbacks or call the `stopPropagation()` method on the event object:

```
public function doSomething($event)
{
    // ...
    return false; // Stops the event
}

public function updateBuyStatistic($event)
{
    // ...
    $event->stopPropagation();
}
```

Stopping an event will prevent any additional callbacks from being called. Additionally the code triggering the event may behave differently based on the event being stopped or not. Generally it does not make sense to stop 'after' events, but stopping 'before' events is often used to prevent the entire operation from occurring.

To check if an event was stopped, you call the `isStopped()` method in the event object:

```
public function place($order)
{
    $event = new Event('Order.beforePlace', $this, ['order' => $order]);
    $this->getEventManager()->dispatch($event);
    if ($event->isStopped()) {
        return false;
    }
    if ($this->Orders->save($order)) {
        // ...
    }
    // ...
}
```

In the previous example the order would not get saved if the event is stopped during the `beforePlace` process.

Getting Event Results

Every time a callback returns a non-null non-false value, it gets stored in the `$result` property of the event object. This is useful when you want to allow callbacks to modify the event execution. Let's take again our `beforePlace` example and let callbacks modify the `$order` data.

Event results can be altered either using the event object result property directly or returning the value in the callback itself:

```
// A listener callback
public function doSomething($event)
{
```

(continues on next page)

(continued from previous page)

```
    // ...
    $alteredData = $event->getData('order') + $moreData;
    return $alteredData;
}

// Another listener callback
public function doSomethingElse($event)
{
    // ...
    $event->setResult(['order' => $alteredData] + $this->result());
}

// Using the event result
public function place($order)
{
    $event = new Event('Order.beforePlace', $this, ['order' => $order]);
    $this->getEventManager()->dispatch($event);
    if (!empty($event->getResult()['order'])) {
        $order = $event->getResult()['order'];
    }
    if ($this->Orders->save($order)) {
        // ...
    }
    // ...
}
```

It is possible to alter any event object property and have the new data passed to the next callback. In most of the cases, providing objects as event data or result and directly altering the object is the best solution as the reference is kept the same and modifications are shared across all callback calls.

Removing Callbacks and Listeners

If for any reason you want to remove any callback from the event manager just call the `Cake\Event\EventManager::off()` method using as arguments the first two parameters you used for attaching it:

```
// Attaching a function
$this->getEventManager()->on('My.event', [$this, 'doSomething']);

// Detaching the function
$this->getEventManager()->off('My.event', [$this, 'doSomething']);

// Attaching an anonymous function.
$myFunction = function ($event) { ... };
$this->getEventManager()->on('My.event', $myFunction);

// Detaching the anonymous function
$this->getEventManager()->off('My.event', $myFunction);

// Adding a EventListener
$listener = new MyEventLister();
$this->getEventManager()->on($listener);
```

(continues on next page)

(continued from previous page)

```
// Detaching a single event key from a listener
$this->getEventManager()->off('My.event', $listener);

// Detaching all callbacks implemented by a listener
$this->getEventManager()->off($listener);
```

Events are a great way of separating concerns in your application and make classes both cohesive and decoupled from each other. Events can be utilized to de-couple application code and make extensible plugins.

Keep in mind that with great power comes great responsibility. Using too many events can make debugging harder and require additional integration testing.

Additional Reading

- *Behaviors*
- *Command Objects*
- *Components*
- *Helpers*
- *Testing Events*

CHAPTER 21

Internationalization & Localization

One of the best ways for an application to reach a larger audience is to cater to multiple languages. This can often prove to be a daunting task, but the internationalization and localization features in CakePHP make it much easier.

First, it's important to understand some terminology. *Internationalization* refers to the ability of an application to be localized. The term *localization* refers to the adaptation of an application to meet specific language (or culture) requirements (i.e. a "locale"). Internationalization and localization are often abbreviated as i18n and l10n respectively; 18 and 10 are the number of characters between the first and last character.

Setting Up Translations

There are only a few steps to go from a single-language application to a multi-lingual application, the first of which is to make use of the `__()` function in your code. Below is an example of some code for a single-language application:

```
<h2>Popular Articles</h2>
```

To internationalize your code, all you need to do is to wrap strings in `__()` like so:

```
<h2><?= __('Popular Articles') ?></h2>
```

Doing nothing else, these two code examples are functionally identical - they will both send the same content to the browser. The `__()` function will translate the passed string if a translation is available, or return it unmodified.

Language Files

Translations can be made available by using language files stored in the application. The default format for CakePHP translation files is the Gettext[144] format. Files need to be placed under **resources/locales/** and within this directory, there should be a subfolder for each language the application needs to support:

```
resources/
    locales/
        en_US/
            default.po
        en_GB/
            default.po
            validation.po
        es/
            default.po
```

The default domain is 'default', therefore the locale folder should at least contain the **default.po** file as shown above. A domain refers to any arbitrary grouping of translation messages. When no group is used, then the default group is selected.

The core strings messages extracted from the CakePHP library can be stored separately in a file named **cake.po** in **resources/locales/**. The CakePHP localized library[145] houses translations for the client-facing translated strings in the core (the cake domain). To use these files, link or copy them into their expected location: **resources/locales/<locale>/cake.po**. If your locale is incomplete or incorrect, please submit a PR in this repository to fix it.

Plugins can also contain translation files, the convention is to use the `under_scored` version of the plugin name as the domain for the translation messages:

```
MyPlugin/
    resources/
        locales/
            fr/
                my_plugin.po
                additional.po
            de/
                my_plugin.po
```

Translation folders can be the two or three letter ISO code of the language or the full ICU locale name such as `fr_FR`, `es_AR`, `da_DK` which contains both the language and the country where it is spoken.

See https://www.localeplanet.com/icu/ for the full list of locales.

Changed in version 4.5.0: As of 4.5.0 plugins can contain multiple translation domains. Use `MyPlugin.additional` to reference plugin domains.

An example translation file could look like this:

```
msgid "My name is {0}"
msgstr "Je m'appelle {0}"

msgid "I'm {0,number} years old"
msgstr "J'ai {0,number} ans"
```

[144] https://en.wikipedia.org/wiki/Gettext
[145] https://github.com/cakephp/localized

Note: Translations are cached - Make sure that you always clear the cache after making changes to translations! You can either use the *cache tool* and run for example `bin/cake cache clear _cake_core_`, or manually clear the `tmp/cache/persistent` folder (if using file based caching).

Extract Pot Files with I18n Shell

To create the pot files from *__()* and other internationalized types of messages that can be found in the application code, you can use the i18n command. Please read the *following chapter* to learn more.

Setting the Default Locale

The default locale can be set in your **config/app.php** file by setting `App.defaultLocale`:

```
'App' => [
    ...
    'defaultLocale' => env('APP_DEFAULT_LOCALE', 'en_US'),
    ...
]
```

This will control several aspects of the application, including the default translations language, the date format, number format and currency whenever any of those is displayed using the localization libraries that CakePHP provides.

Changing the Locale at Runtime

To change the language for translated strings you can call this method:

```
use Cake\I18n\I18n;

I18n::setLocale('de_DE');
```

This will also change how numbers and dates are formatted when using one of the localization tools.

Using Translation Functions

CakePHP provides several functions that will help you internationalize your application. The most frequently used one is `__()`. This function is used to retrieve a single translation message or return the same string if no translation was found:

```
echo __('Popular Articles');
```

If you need to group your messages, for example, translations inside a plugin, you can use the `__d()` function to fetch messages from another domain:

```
echo __d('my_plugin', 'Trending right now');
```

Note: If you want to translate plugins that are vendor namespaced, you must use the domain string `vendor/plugin_name`. But the related language file will become `plugins/<Vendor>/<PluginName>/resources/locales/<locale>/plugin_name.po` inside your plugin folder.

Sometimes translations strings can be ambiguous for people translating them. This can happen if two strings are identical but refer to different things. For example, 'letter' has multiple meanings in English. To solve that problem, you can use the `__x()` function:

```
echo __x('written communication', 'He read the first letter');

echo __x('alphabet learning', 'He read the first letter');
```

The first argument is the context of the message and the second is the message to be translated.

```
msgctxt "written communication"
msgid "He read the first letter"
msgstr "Er las den ersten Brief"
```

Using Variables in Translation Messages

Translation functions allow you to interpolate variables into the messages using special markers defined in the message itself or in the translated string:

```
echo __("Hello, my name is {0}, I'm {1} years old", ['Sara', 12]);
```

Markers are numeric, and correspond to the keys in the passed array. You can also pass variables as independent arguments to the function:

```
echo __("Small step for {0}, Big leap for {1}", 'Man', 'Humanity');
```

All translation functions support placeholder replacements:

```
__d('validation', 'The field {0} cannot be left empty', 'Name');

__x('alphabet', 'He read the letter {0}', 'Z');
```

The `'` (single quote) character acts as an escape code in translation messages. Any variables between single quotes will not be replaced and is treated as literal text. For example:

```
__("This variable '{0}' be replaced.", 'will not');
```

By using two adjacent quotes your variables will be replaced properly:

```
__("This variable ''{0}'' be replaced.", 'will');
```

These functions take advantage of the ICU MessageFormatter[146] so you can translate messages and localize dates, numbers and currency at the same time:

```
echo __(
    'Hi {0}, your balance on the {1,date} is {2,number,currency}',
```

(continues on next page)

[146] https://php.net/manual/en/messageformatter.format.php

(continued from previous page)

```
    ['Charles', new DateTime('2014-01-13 11:12:00'), 1354.37]
);

// Returns
Hi Charles, your balance on the Jan 13, 2014, 11:12 AM is $ 1,354.37
```

Numbers in placeholders can be formatted as well with fine grain control of the output:

```
echo __(
    'You have traveled {0,number} kilometers in {1,number,integer} weeks',
    [5423.344, 5.1]
);

// Returns
You have traveled 5,423.34 kilometers in 5 weeks

echo __('There are {0,number,#,###} people on earth', 6.1 * pow(10, 8));

// Returns
There are 6,100,000,000 people on earth
```

This is the list of formatter specifiers you can put after the word `number`:

- `integer`: Removes the decimal part
- `currency`: Puts the locale currency symbol and rounds decimals
- `percent`: Formats the number as a percentage

Dates can also be formatted by using the word `date` after the placeholder number. A list of extra options follows:

- `short`
- `medium`
- `long`
- `full`

The word `time` after the placeholder number is also accepted and it understands the same options as `date`.

You can also use named placeholders like `{name}` in the message strings. When using named placeholders, pass the placeholder and replacement in an array using key/value pairs, for example:

```
// echos:  Hi. My name is Sara. I'm 12 years old.
echo __("Hi. My name is {name}. I'm {age} years old.", ['name' => 'Sara', 'age' => 12]);
```

Plurals

One crucial part of internationalizing your application is getting your messages pluralized correctly depending on the language they are shown. CakePHP provides a couple ways to correctly select plurals in your messages.

Using ICU Plural Selection

The first one is taking advantage of the ICU message format that comes by default in the translation functions. In the translations file you could have the following strings

```
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}}"
```

And in the application use the following code to output either of the translations for such string:

```
__('{0,plural,=0{No records found }=1{Found 1 record} other{Found # records}}', [0]);

// Returns "Ningún resultado" as the argument {0} is 0

__('{0,plural,=0{No records found} =1{Found 1 record} other{Found # records}}', [1]);

// Returns "1 resultado" because the argument {0} is 1

__('{placeholder,plural,=0{No records found} =1{Found 1 record} other{Found {1} records}}
↪', [0, 'many', 'placeholder' => 2])

// Returns "many resultados" because the argument {placeholder} is 2 and
// argument {1} is 'many'
```

A closer look to the format we just used will make it evident how messages are built:

```
{ [count placeholder],plural, case1{message} case2{message} case3{...} ... }
```

The `[count placeholder]` can be the array key number of any of the variables you pass to the translation function. It will be used for selecting the correct plural form.

Note that to reference `[count placeholder]` within `{message}` you have to use `#`.

You can of course use simpler message ids if you don't want to type the full plural selection sequence in your code

```
msgid "search.results"
msgstr "{0,plural,=0{Ningún resultado} =1{1 resultado} other{{1} resultados}}"
```

Then use the new string in your code:

```
__('search.results', [2, 2]);

// Returns: "2 resultados"
```

The latter version has the downside that there is a need to have a translation messages file even for the default language, but has the advantage that it makes the code more readable and leaves the complicated plural selection strings in the translation files.

Sometimes using direct number matching in plurals is impractical. For example, languages like Arabic require a different plural when you refer to few things and other plural form for many things. In those cases you can use the ICU matching aliases. Instead of writing:

```
=0{No results} =1{...} other{...}
```

You can do:

```
zero{No Results} one{One result} few{...} many{...} other{...}
```

Make sure you read the Language Plural Rules Guide[147] to get a complete overview of the aliases you can use for each language.

Using Gettext Plural Selection

The second plural selection format accepted is using the built-in capabilities of Gettext. In this case, plurals will be stored in the `.po` file by creating a separate message translation line per plural form:

```
# One message identifier for singular
msgid "One file removed"
# Another one for plural
msgid_plural "{0} files removed"
# Translation in singular
msgstr[0] "Un fichero eliminado"
# Translation in plural
msgstr[1] "{0} ficheros eliminados"
```

When using this other format, you are required to use another translation function:

```
// Returns: "10 ficheros eliminados"
$count = 10;
__n('One file removed', '{0} files removed', $count, $count);

// It is also possible to use it inside a domain
__dn('my_plugin', 'One file removed', '{0} files removed', $count, $count);
```

The number inside `msgstr[]` is the number assigned by Gettext for the plural form of the language. Some languages have more than two plural forms, for example Croatian:

```
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"
```

Please visit the Launchpad languages page[148] for a detailed explanation of the plural form numbers for each language.

[147] https://unicode-org.github.io/cldr-staging/charts/37/supplemental/language_plural_rules.html
[148] https://translations.launchpad.net/+languages

Creating Your Own Translators

If you need to diverge from CakePHP conventions regarding where and how translation messages are stored, you can create your own translation message loader. The easiest way to create your own translator is by defining a loader for a single domain and locale:

```
use Cake\I18n\Package;
// Prior to 4.2 you need to 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'
        ...
    ]);

    return $package;
}, 'fr_FR');
```

The above code can be added to your **config/bootstrap.php** so that translations can be found before any translation function is used. The absolute minimum that is required for creating a translator is that the loader function should return a `Cake\I18n\Package` object (prior to 4.2 it should be an `Aura\Intl\Package` object). Once the code is in place you can use the translation functions as usual:

```
I18n::setLocale('fr_FR');
__d('animals', 'Dog'); // Returns "Chien"
```

As you see, `Package` objects take translation messages as an array. You can pass the `setMessages()` method however you like: with inline code, including another file, calling another function, etc. CakePHP provides a few loader functions you can reuse if you just need to change where messages are loaded. For example, you can still use **.po** files, but loaded from another location:

```
use Cake\I18n\MessagesFileLoader as Loader;

// Load messages from resources/locales/folder/sub_folder/filename.po
I18n::setTranslator(
    'animals',
    new Loader('filename', 'folder/sub_folder', 'po'),
    'fr_FR'
);
```

Creating Message Parsers

It is possible to continue using the same conventions CakePHP uses, but use a message parser other than `PoFileParser`. For example, if you wanted to load translation messages using `YAML`, you will first need to created the parser class:

```
namespace App\I18n\Parser;

class YamlFileParser
{
    public function parse($file)
    {
        return yaml_parse_file($file);
    }
}
```

The file should be created in the **src/I18n/Parser** directory of your application. Next, create the translations file under **resources/locales/fr_FR/animals.yaml**

```
Dog: Chien
Cat: Chat
Bird: Oiseau
```

And finally, configure the translation loader for the domain and locale:

```
use Cake\I18n\MessagesFileLoader as Loader;

I18n::setTranslator(
    'animals',
    new Loader('animals', 'fr_FR', 'yaml'),
    'fr_FR'
);
```

Creating Generic Translators

Configuring translators by calling `I18n::setTranslator()` for each domain and locale you need to support can be tedious, specially if you need to support more than a few different locales. To avoid this problem, CakePHP lets you define generic translator loaders for each domain.

Imagine that you wanted to load all translations for the default domain and for any language from an external service:

```
use Cake\I18n\Package;
// Prior to 4.2 you need to use Aura\Intl\Package

I18n::config('default', function ($domain, $locale) {
    $locale = Locale::parseLocale($locale);
    $lang = $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)
```

(continues on next page)

(continued from previous page)

```
    )
});
```

The above example calls an example external service to load a JSON file with the translations and then just build a `Package` object for any locale that is requested in the 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.
});
```

Plurals and Context in Custom Translators

The arrays used for `setMessages()` can be crafted to instruct the translator to store messages under different domains or to trigger Gettext-style plural selection. The following is an example of storing translations for the same key in different contexts:

```
[
    'He reads the letter {0}' => [
        'alphabet' => 'Él lee la letra {0}',
        'written communication' => 'Él lee la carta {0}',
    ],
]
```

Similarly, you can express Gettext-style plurals using the messages array by having a nested array key per plural form:

```
[
    'I have read one book' => 'He leído un libro',
    'I have read {0} books' => [
        'He leído un libro',
        'He leído {0} libros',
    ],
]
```

Using Different Formatters

In previous examples we have seen that Packages are built using `default` as first argument, and it was indicated with a comment that it corresponded to the formatter to be used. Formatters are classes responsible for interpolating variables in translation messages and selecting the correct plural form.

If you're dealing with a legacy application, or you don't need the power offered by the ICU message formatting, CakePHP also provides the `sprintf` formatter:

```
return Package('sprintf', 'fallback_domain', $messages);
```

The messages to be translated will be passed to the `sprintf()` function for interpolating the variables:

```
__('Hello, my name is %s and I am %d years old', 'José', 29);
```

It is possible to set the default formatter for all translators created by CakePHP before they are used for the first time. This does not include manually created translators using the `setTranslator()` and `config()` methods:

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

Localizing Dates and Numbers

When outputting Dates and Numbers in your application, you will often need that they are formatted according to the preferred format for the country or region that you wish your page to be displayed.

In order to change how dates and numbers are displayed you just need to change the current locale setting and use the right classes:

```
use Cake\I18n\I18n;
use Cake\I18n\DateTime;
use Cake\I18n\Number;

I18n::setLocale('fr-FR');

$date = new DateTime('2015-04-05 23:00:00');

echo $date; // Displays 05/04/2015 23:00

echo Number::format(524.23); // Displays 524,23
```

Make sure you read the *Date & Time* and *Number* sections to learn more about formatting options.

By default dates returned for the ORM results use the `Cake\I18n\DateTime` class, so displaying them directly in you application will be affected by changing the current locale.

Parsing Localized Datetime Data

When accepting localized data from the request, it is nice to accept datetime information in a user's localized format. In a controller, or /controllers/middleware you can configure the Date, Time, and DateTime types to parse localized formats:

```
use Cake\Database\TypeFactory;

// Enable default locale format parsing.
TypeFactory::build('datetime')->useLocaleParser();

// Configure a custom datetime format parser format.
TypeFactory::build('datetime')->useLocaleParser()->setLocaleFormat('dd-M-y');

// You can also use IntlDateFormatter constants.
TypeFactory::build('datetime')->useLocaleParser()
    ->setLocaleFormat([IntlDateFormatter::SHORT, -1]);
```

The default parsing format is the same as the default string format.

Converting Request Data from the User's Timezone

When handling data from users in different timezones you will need to convert the datetimes in request data into your application's timezone. You can use `setUserTimezone()` from a controller or /controllers/middleware to make this process simpler:

```
// Set the user's timezone
TypeFactory::build('datetime')->setUserTimezone($user->timezone);
```

Once set, when your application creates or updates entities from request data, the ORM will automatically convert datetime values from the user's timezone into your application's timezone. This ensures that your application is always working in the timezone defined in `App.defaultTimezone`.

If your application handles datetime information in a number of actions you can use a middleware to define both timezone conversion and locale parsing:

```
namespace App\Middleware;

use Cake\Database\TypeFactory;
use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;
use Psr\Http\Server\MiddlewareInterface;
use Psr\Http\Server\RequestHandlerInterface;

class DatetimeMiddleware implements MiddlewareInterface
{
    public function process(
        ServerRequestInterface $request,
        RequestHandlerInterface $handler
    ): ResponseInterface {
        // Get the user from the request.
        // This example assumes your user entity has a timezone attribute.
        $user = $request->getAttribute('identity');
        if ($user) {
            TypeFactory::build('datetime')
                ->useLocaleParser()
                ->setUserTimezone($user->timezone);
        }

        return $handler->handle($request);
    }
}
```

Automatically Choosing the Locale Based on Request Data

By using the `LocaleSelectorMiddleware` in your application, CakePHP will automatically set the locale based on the current user:

```
// in src/Application.php
use Cake\I18n\Middleware\LocaleSelectorMiddleware;

// Update the middleware function, adding the new middleware
```

(continues on next page)

(continued from previous page)

```
public function middleware(MiddlewareQueue $middlewareQueue): MiddlewareQueue
{
    // Add middleware and set the valid locales
    $middlewareQueue->add(new LocaleSelectorMiddleware(['en_US', 'fr_FR']));
    // To accept any locale header value
    $middlewareQueue->add(new LocaleSelectorMiddleware(['*']));
}
```

The `LocaleSelectorMiddleware` will use the `Accept-Language` header to automatically set the user's preferred locale. You can use the locale list option to restrict which locales will automatically be used.

Translate Content/Entities

If you want to translate content/entities then you should look at the *Translate Behavior*.

CHAPTER 22

Logging

While CakePHP core Configure Class settings can really help you see what's happening under the hood, there are certain times that you'll need to log data to the disk in order to find out what's going on. With technologies like SOAP, AJAX, and REST APIs, debugging can be rather difficult.

Logging can also be a way to find out what's been going on in your application over time. What search terms are being used? What sorts of errors are my users being shown? How often is a particular query being executed?

Logging data in CakePHP is done with the `log()` function. It is provided by the `LogTrait`, which is the common ancestor for many CakePHP classes. If the context is a CakePHP class (Controller, Component, View,...), you can log your data. You can also use `Log::write()` directly. See *Writing to Logs*.

Logging Configuration

Configuring `Log` should be done during your application's bootstrap phase. The **config/app.php** file is intended for just this. You can define as many or as few loggers as your application needs. Loggers should be configured using *Cake\Log\Log*. An example would be:

```
use Cake\Log\Engine\FileLog;
use Cake\Log\Log;

// Classname using logger 'class' constant
Log::setConfig('info', [
    'className' => FileLog::class,
    'path' => LOGS,
    'levels' => ['info'],
    'file' => 'info',
]);

// Short classname
```

(continues on next page)

(continued from previous page)

```
Log::setConfig('debug', [
    'className' => 'File',
    'path' => LOGS,
    'levels' => ['notice', 'debug'],
    'file' => 'debug',
]);

// Fully namespaced name.
Log::setConfig('error', [
    'className' => 'Cake\Log\Engine\FileLog',
    'path' => LOGS,
    'levels' => ['warning', 'error', 'critical', 'alert', 'emergency'],
    'file' => 'error',
]);
```

The above creates three loggers, named `info`, `debug` and `error`. Each is configured to handle different levels of messages. They also store their log messages in separate files, so we can separate debug/notice/info logs from more serious errors. See the section on *Using Levels* for more information on the different levels and what they mean.

Once a configuration is created you cannot change it. Instead you should drop the configuration and re-create it using *Cake\Log\Log::drop()* and *Cake\Log\Log::setConfig()*.

It is also possible to create loggers by providing a closure. This is useful when you need full control over how the logger object is built. The closure has to return the constructed logger instance. For example:

```
Log::setConfig('special', function () {
    return new \Cake\Log\Engine\FileLog(['path' => LOGS, 'file' => 'log']);
});
```

Configuration options can also be provided as a *DSN* string. This is useful when working with environment variables or *PaaS* providers:

```
Log::setConfig('error', [
    'url' => 'file:///full/path/to/logs/?levels[]=warning&levels[]=error&file=error',
]);
```

Warning: If you do not configure logging engines, log messages will not be stored.

Error and Exception Logging

Errors and Exceptions can also be logged. By configuring the corresponding values in your **config/app.php** file. Errors will be displayed when debug is `true` and logged when debug is `false`. To log uncaught exceptions, set the `log` option to `true`. See *Configuration* for more information.

Writing to Logs

Writing to the log files can be done in two different ways. The first is to use the static *Cake\Log\Log::write()* method:

```
Log::write('debug', 'Something did not work');
```

The second is to use the `log()` shortcut function available on any class using the `LogTrait`. Calling `log()` will internally call `Log::write()`:

```
// Executing this inside a class using LogTrait
$this->log('Something did not work!', 'debug');
```

All configured log streams are written to sequentially each time *Cake\Log\Log::write()* is called. If you have not configured any logging engines `log()` will return `false` and no log messages will be written.

Using Placeholders in Messages

If you need to log dynamically defined data, you can use placeholders in your log messages and provide an array of key/value pairs in the `$context` parameter:

```
// Will log `Could not process for userid=1`
Log::write('error', 'Could not process for userid={user}', ['user' => $user->id]);
```

Placeholders that do not have keys defined will not be replaced. If you need to use a literal braced word, you must escape the placeholder:

```
// Will log `No {replace}`
Log::write('error', 'No \\{replace}', ['replace' => 'no']);
```

If you include objects in your logging placeholders those objects must implement one of the following methods:

- `__toString()`
- `toArray()`
- `__debugInfo()`

Using Levels

CakePHP supports the standard POSIX set of logging levels. Each level represents an increasing level of severity:

- Emergency: system is unusable
- Alert: action must be taken immediately
- Critical: critical conditions
- Error: error conditions
- Warning: warning conditions
- Notice: normal but significant condition
- Info: informational messages
- Debug: debug-level messages

You can refer to these levels by name when configuring loggers, and when writing log messages. Alternatively, you can use convenience methods like *Cake\Log\Log::error()* to clearly indicate the logging level. Using a level that is not in the above levels will result in an exception.

Note: When `levels` is set to an empty value in a logger's configuration, it will take messages of any level.

Logging Scopes

Often times you'll want to configure different logging behavior for different subsystems or parts of your application. Take for example an e-commerce shop. You'll probably want to handle logging for orders and payments differently than you do other less critical logs.

CakePHP exposes this concept as logging scopes. When log messages are written you can include a scope name. If there is a configured logger for that scope, the log messages will be directed to those loggers. For example:

```
use Cake\Log\Engine\FileLog;

// Configure logs/shops.log to receive all levels, but only
// those with `orders` and `payments` scope.
Log::setConfig('shops', [
    'className' => FileLog::class,
    'path' => LOGS,
    'levels' => [],
    'scopes' => ['orders', 'payments'],
    'file' => 'shops.log',
]);

// Configure logs/payments.log to receive all levels, but only
// those with `payments` scope.
Log::setConfig('payments', [
    'className' => FileLog::class,
    'path' => LOGS,
    'levels' => [],
    'scopes' => ['payments'],
    'file' => 'payments.log',
]);
```

(continues on next page)

(continued from previous page)

```
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']]);
```

Scopes can also be passed as a single string or a numerically indexed array. Note that using this form will limit the ability to pass more data as context:

```
Log::warning('This is a warning', ['orders']);
Log::warning('This is a warning', 'payments');
```

Note: When `scopes` is set to an empty array or `null` in a logger's configuration, it will take messages of any scope. Setting it to `false` will only match messages without scope.

Logging to Files

As its name implies `FileLog` writes log messages to files. The level of log message being written determines the name of the file the message is stored in. If a level is not supplied, `LOG_ERR` is used which writes to the error log. The default log location is **logs/$level.log**:

```
// Executing this inside a CakePHP class
$this->log("Something didn't work!");

// Results in this being appended to logs/error.log
// 2007-11-02 10:22:02 Error: Something didn't work!
```

The configured directory must be writable by the web server user in order for logging to work correctly.

You can configure additional/alternate FileLog locations when configuring a logger. FileLog accepts a `path` which allows for custom paths to be used:

```
Log::setConfig('custom_path', [
    'className' => 'File',
    'path' => '/path/to/custom/place/'
]);
```

`FileLog` engine takes the following options:

- `size` Used to implement basic log file rotation. If log file size reaches specified size the existing file is renamed by appending timestamp to filename and new log file is created. Can be integer bytes value or human readable string values like '10MB', '100KB' etc. Defaults to 10MB.
- `rotate` Log files are rotated specified times before being removed. If value is 0, old versions are removed rather then rotated. Defaults to 10.
- `mask` Set the file permissions for created files. If left empty the default permissions are used.

Note: Missing directories will be automatically created to avoid unnecessary errors thrown when using the FileEngine.

Logging to Syslog

In production environments it is highly recommended that you setup your system to use syslog instead of the file logger. This will perform much better as any writes will be done in a (almost) non-blocking fashion and your operating system logger can be configured separately to rotate files, pre-process writes or use a completely different storage for your logs.

Using syslog is pretty much like using the default FileLog engine, you just need to specify `Syslog` as the engine to be used for logging. The following configuration snippet will replace the default logger with syslog, this should be done in the **config/bootstrap.php** file:

```
Log::setConfig('default', [
    'engine' => 'Syslog'
]);
```

The configuration array accepted for the Syslog logging engine understands the following keys:

- `format`: An sprintf template string with two placeholders, the first one for the error level, and the second for the message itself. This key is useful to add additional information about the server or process in the logged message. For example: `%s - Web Server 1 - %s` will look like `error - Web Server 1 - An error occurred in this request` after replacing the placeholders. This option is deprecated. You should use *Logging Formatters* instead.
- `prefix`: An string that will be prefixed to every logged message.
- `flag`: An integer flag to be used for opening the connection to the logger, by default `LOG_ODELAY` will be used. See `openlog` documentation for more options
- `facility`: The logging slot to use in syslog. By default `LOG_USER` is used. See `syslog` documentation for more options

Creating Log Engines

Log engines can be part of your application, or part of plugins. If for example you had a database logger called `DatabaseLog`. As part of your application it would be placed in **src/Log/Engine/DatabaseLog.php**. As part of a plugin it would be placed in **plugins/LoggingPack/src/Log/Engine/DatabaseLog.php**. To configure log engine you should use *`Cake\Log\Log::setConfig()`*. For example configuring our DatabaseLog would look like:

```
// For src/Log
Log::setConfig('otherFile', [
    'className' => 'Database',
    'model' => 'LogEntry',
    // ...
]);

// For plugin called LoggingPack
Log::setConfig('otherFile', [
    'className' => 'LoggingPack.Database',
    'model' => 'LogEntry',
    // ...
]);
```

When configuring a log engine the `className` parameter is used to locate and load the log handler. All of the other configuration properties are passed to the log engine's constructor as an array.

```php
namespace App\Log\Engine;
use Cake\Log\Engine\BaseLog;

class DatabaseLog extends BaseLog
{
    public function __construct(array $config = [])
    {
        parent::__construct($config);
        // ...
    }

    public function log($level, string $message, array $context = [])
    {
        // Write to the database.
    }
}
```

CakePHP requires that all logging engine implement `Psr\Log\LoggerInterface`. The class `Cake\Log\Engine\BaseLog` is an easy way to satisfy the interface as it only requires you to implement the `log()` method.

Logging Formatters

Logging formatters allow you to control how log messages are formatted independent of the storage engine. Each core provided logging engine comes with a formatter configured to maintain backwards compatible output. However, you can adjust the formatters to fit your requirements. Formatters are configured alongside the logging engine:

```php
use Cake\Log\Engine\SyslogLog;
use App\Log\Formatter\CustomFormatter;

// Simple formatting configuration with no options.
Log::setConfig('error', [
    'className' => SyslogLog::class,
    'formatter' => CustomFormatter::class,
]);

// Configure a formatter with additional options.
Log::setConfig('error', [
    'className' => SyslogLog::class,
    'formatter' => [
        'className' => CustomFormatter::class,
        'key' => 'value',
    ],
]);
```

To implement your own logging formatter you need to extend `Cake\Log\Format\AbstractFormatter` or one of its subclasses. The primary method you need to implement is `format($level, $message, $context)` which is responsible for formatting log messages.

Log API

class Cake\Log**Log**

A simple class for writing to logs.

static Cake\Log\Log::**setConfig**(*$key*, *$config*)

Parameters

- **$name** (string) – Name for the logger being connected, used to drop a logger later on.
- **$config** (array) – Array of configuration information and constructor arguments for the logger.

Get or set the configuration for a Logger. See *Logging Configuration* for more information.

static Cake\Log\Log::**configured**

Returns
An array of configured loggers.

Get the names of the configured loggers.

static Cake\Log\Log::**drop**(*$name*)

Parameters

- **$name** (string) – Name of the logger you wish to no longer receive messages.

static Cake\Log\Log::**write**(*$level*, *$message*, *$scope = []*)

Write a message into all the configured loggers. `$level` indicates the level of log message being created. `$message` is the message of the log entry being written to. `$scope` is the scope(s) a log message is being created in.

static Cake\Log\Log::**levels**

Call this method without arguments, eg: *Log::levels()* to obtain current level configuration.

Convenience Methods

The following convenience methods were added to log *$message* with the appropriate log level.

static Cake\Log\Log::**emergency**(*$message*, *$scope = []*)

static Cake\Log\Log::**alert**(*$message*, *$scope = []*)

static Cake\Log\Log::**critical**(*$message*, *$scope = []*)

static Cake\Log\Log::**error**(*$message*, *$scope = []*)

static Cake\Log\Log::**warning**(*$message*, *$scope = []*)

static Cake\Log\Log::**notice**(*$message*, *$scope = []*)

static Cake\Log\Log::**info**(*$message*, *$scope = []*)

static Cake\Log\Log::**debug**(*$message*, *$scope = []*)

Logging Trait

trait Cake\Log**LogTrait**

A trait that provides shortcut methods for logging

Cake\Log\LogTrait::**log**(*$msg*, *$level = LOG_ERR*)

Log a message to the logs. By default messages are logged as ERROR messages.

Using Monolog

Monolog is a popular logger for PHP. Since it implements the same interfaces as the CakePHP loggers, you can use them in your application as the default logger.

After installing Monolog using composer, configure the logger using the `Log::setConfig()` method:

```
// config/bootstrap.php

use Monolog\Logger;
use Monolog\Handler\StreamHandler;

Log::setConfig('default', function () {
    $log = new Logger('app');
    $log->pushHandler(new StreamHandler('path/to/your/combined.log'));
    return $log;
});

// Optionally stop using the now redundant default loggers
Log::drop('debug');
Log::drop('error');
```

Use similar methods if you want to configure a different logger for your console:

```
// config/bootstrap_cli.php

use Monolog\Logger;
use Monolog\Handler\StreamHandler;

Log::setConfig('default', function () {
    $log = new Logger('cli');
    $log->pushHandler(new StreamHandler('path/to/your/combined-cli.log'));
    return $log;
});

// Optionally stop using the now redundant default CLI loggers
Configure::delete('Log.debug');
Configure::delete('Log.error');
```

Note: When using a console specific logger, make sure to conditionally configure your application logger. This will prevent duplicate log entries.

CHAPTER 23

Modelless Forms

class Cake\Form**Form**

Most of the time you will have forms backed by *ORM entities* and *ORM tables* or other persistent stores, but there are times when you'll need to validate user input and then perform an action if the data is valid. The most common example of this is a contact form.

Creating a Form

Generally when using the Form class you'll want to use a subclass to define your form. This makes testing easier, and lets you re-use your form. Forms are put into **src/Form** and usually have `Form` as a class suffix. For example, a simple contact form would look like:

```
// in 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): Schema
    {
        return $schema->addField('name', 'string')
            ->addField('email', ['type' => 'string'])
            ->addField('body', ['type' => 'text']);
    }
```

(continues on next page)

(continued from previous page)

```
    public function validationDefault(Validator $validator): Validator
    {
        $validator->minLength('name', 10)
            ->email('email');

        return $validator;
    }

    protected function _execute(array $data): bool
    {
        // Send an email.
        return true;
    }
}
```

In the above example we see the 3 hook methods that forms provide:

- `_buildSchema` is used to define the schema data that is used by FormHelper to create an HTML form. You can define field type, length, and precision.
- `validationDefault` Gets a *Cake\Validation\Validator* instance that you can attach validators to.
- `_execute` lets you define the behavior you want to happen when `execute()` is called and the data is valid.

You can always define additional public methods as you need as well.

Processing Request Data

Once you've defined your form, you can use it in your controller to process and validate request data:

```
// In a 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('We will get back to you soon.');
            } else {
                $this->Flash->error('There was a problem submitting your form.');
            }
        }
        $this->set('contact', $contact);
    }
}
```

In the above example, we use the `execute()` method to run our form's `_execute()` method only when the data is

valid, and set flash messages accordingly. If we want to use a non-default validation set we can use the `validate` option:

```
if ($contact->execute($this->request->getData(), 'update')) {
    // Handle form success.
}
```

This option can also be set to `false` to disable validation.

We could have also used the `validate()` method to only validate the request data:

```
$isValid = $form->validate($this->request->getData());

// You can also use other validation sets. The following
// would use the rules defined by `validationUpdate()`
$isValid = $form->validate($this->request->getData(), 'update');
```

Setting Form Values

You can set default values for modelless forms using the `setData()` method. Values set with this method will overwrite existing data in the form object:

```
// In a 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('We will get back to you soon.');
            } else {
                $this->Flash->error('There was a problem submitting your form.');
            }
        }

        if ($this->request->is('get')) {
            $contact->setData([
                'name' => 'John Doe',
                'email' => 'john.doe@example.com'
            ]);
        }

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

Values should only be defined if the request method is GET, otherwise you will overwrite your previous POST Data

which might have validation errors that need corrections. You can use `set()` to add or replace individual fields or a subset of fields:

```
// Set one field.
$contact->set('name', 'John Doe');

// Set multiple fields;
$contact->set([
    'name' => 'John Doe',
    'email' => 'john.doe@example.com',
]);
```

Getting Form Errors

Once a form has been validated you can retrieve the errors from it:

```
$errors = $form->getErrors();
/* $errors contains
[
    'name' => ['length' => 'Name must be at least two characters long'],
    'email' => ['format' => 'A valid email address is required'],
]
*/

$error = $form->getError('email');
/* $error contains
[
    'format' => 'A valid email address is required',
]
*/
```

Invalidating Individual Form Fields from Controller

It is possible to invalidate individual fields from the controller without the use of the Validator class. The most common use case for this is when the validation is done on a remote server. In such case, you must manually invalidate the fields accordingly to the feedback from the remote server:

```
// in src/Form/ContactForm.php
public function setErrors($errors)
{
    $this->_errors = $errors;
}
```

According to how the validator class would have returned the errors, `$errors` must be in this format:

```
['fieldName' => ['validatorName' => 'The error message to display']]
```

Now you will be able to invalidate form fields by setting the fieldName, then set the error messages:

```
// In a controller
$contact = new ContactForm();
$contact->setErrors(['email' => ['_required' => 'Your email is required']]);
```

Proceed to Creating HTML with FormHelper to see the results.

Creating HTML with FormHelper

Once you've created a Form class, you'll likely want to create an HTML form for it. FormHelper understands Form objects just like ORM entities:

```
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();
```

The above would create an HTML form for the `ContactForm` we defined earlier. HTML forms created with FormHelper will use the defined schema and validator to determine field types, maxlengths, and validation errors.

CHAPTER 24

Pagination

One of the main obstacles of creating flexible and user-friendly web applications is designing an intuitive user interface. Many applications tend to grow in size and complexity quickly, and designers and programmers alike find they are unable to cope with displaying hundreds or thousands of records. Refactoring takes time, and performance and user satisfaction can suffer.

Displaying a reasonable number of records per page has always been a critical part of every application and used to cause many headaches for developers. CakePHP eases the burden on the developer by providing a terse way to paginate data.

Pagination in CakePHP controllers is done through the `paginate()` method. You then use *PaginatorHelper* in your view templates to generate pagination controls.

Basic Usage

You can call `paginate()` using an ORM table instance or `Query` object:

```
public function index()
{
    // Paginate the ORM table.
    $this->set('articles', $this->paginate($this->Articles));

    // Paginate a select query
    $query = $this->Articles->find('published')->contain('Comments');
    $this->set('articles', $this->paginate($query));
}
```

Advanced Usage

More complex use cases are supported by configuring the `$paginate` controller property or as the `$settings` argument to `paginate()`. These conditions serve as the basis for you pagination queries. They are augmented by the `sort`, `direction`, `limit`, and `page` parameters passed in from the URL:

```
class ArticlesController extends AppController
{
    protected array $paginate = [
        'limit' => 25,
        'order' => [
            'Articles.title' => 'asc',
        ],
    ];
}
```

Tip: Default `order` options must be defined as an array.

You can also use *Custom Finder Methods* in pagination by using the `finder` option:

```
class ArticlesController extends AppController
{
    protected array $paginate = [
        'finder' => 'published',
    ];
}
```

If your finder method requires additional options you can pass those as values for the finder:

```
class ArticlesController extends AppController
{
    // find articles by tag
    public function tags()
    {
        $tags = $this->request->getParam('pass');

        $customFinderOptions = [
            'tags' => $tags
        ];
        // We're using the $settings argument to paginate() here.
        // But the same structure could be used in $this->paginate
        //
        // Our custom finder is called findTagged inside ArticlesTable.php
        // which is why we're using `tagged` as the key.
        // Our finder should look like:
        // public function findTagged(Query $query, array $tagged = [])
        $settings = [
            'finder' => [
                'tagged' => $customFinderOptions
            ]
        ];
        $articles = $this->paginate($this->Articles, $settings);
```

(continues on next page)

(continued from previous page)

```
        $this->set(compact('articles', 'tags'));
    }
}
```

In addition to defining general pagination values, you can define more than one set of pagination defaults in the controller. The name of each model can be used as a key in the `$paginate` property:

```
class ArticlesController extends AppController
{
    protected array $paginate = [
        'Articles' => [],
        'Authors' => [],
    ];
}
```

The values of the `Articles` and `Authors` keys could contain all the keys that a basic `$paginate` array would.

`Controller::paginate()` returns an instance of `Cake\Datasource\Paging\PaginatedResultSet` which implements the `Cake\Datasource\Paging\PaginatedInterface`.

This object contains the paginated records and the paging params.

Simple Pagination

By default `Controller::paginate()` uses the `Cake\Datasource\Paging\NumericPaginator` class which does a `COUNT()` query to calculate the size of the result set so that page number links can be rendered. On very large datasets this count query can be very expensive. In situations where you only want to show 'Next' and 'Previous' links you can use the 'simple' paginator which does not do a count query:

```
class ArticlesController extends AppController
{
    protected array $paginate = [
        'className' => 'Simple', // Or use Cake\Datasource\Paging\SimplePaginator::class␣
↪FQCN
    ];
}
```

When using the `SimplePaginator` you will not be able to generate page numbers, counter data, links to the last page, or total record count controls.

Paginating Multiple Queries

You can paginate multiple models in a single controller action, using the `scope` option both in the controller's `$paginate` property and in the call to the `paginate()` method:

```
// Paginate property
protected array $paginate = [
    'Articles' => ['scope' => 'article'],
    'Tags' => ['scope' => 'tag']
];
```

(continues on next page)

(continued from previous page)

```
// In a controller action
$articles = $this->paginate($this->Articles, ['scope' => 'article']);
$tags = $this->paginate($this->Tags, ['scope' => 'tag']);
$this->set(compact('articles', 'tags'));
```

The `scope` option will result in the paginator looking in scoped query string parameters. For example, the following URL could be used to paginate both tags and articles at the same time:

```
/dashboard?article[page]=1&tag[page]=3
```

See the *Paginating Multiple Results* section for how to generate scoped HTML elements and URLs for pagination.

Paginating the Same Model multiple Times

To paginate the same model multiple times within a single controller action you need to define an alias for the model.:

```
// In a controller action
$this->paginate = [
    'Articles' => [
        'scope' => 'published_articles',
        'limit' => 10,
        'order' => [
            'id' => 'desc',
        ],
    ],
    'UnpublishedArticles' => [
        'scope' => 'unpublished_articles',
        'limit' => 10,
        'order' => [
            'id' => 'desc',
        ],
    ],
];

$publishedArticles = $this->paginate(
    $this->Articles->find('all', scope: 'published_articles')
        ->where(['published' => true])
);

// Load an additional table object to allow differentiating in the paginator
$unpublishedArticlesTable = $this->fetchTable('UnpublishedArticles', [
    'className' => 'App\Model\Table\ArticlesTable',
    'table' => 'articles',
    'entityClass' => 'App\Model\Entity\Article',
]);

$unpublishedArticles = $this->paginate(
    $unpublishedArticlesTable->find('all', scope: 'unpublished_articles')
        ->where(['published' => false])
);
```

Control which Fields Used for Ordering

By default sorting can be done on any non-virtual column a table has. This is sometimes undesirable as it allows users to sort on un-indexed columns that can be expensive to order by. You can set the allowed list of fields that can be sorted using the `sortableFields` option. This option is required when you want to sort on any associated data, or computed fields that may be part of your pagination query:

```
protected array $paginate = [
    'sortableFields' => [
        'id', 'title', 'Users.username', 'created',
    ],
];
```

Any requests that attempt to sort on fields not in the allowed list will be ignored.

Limit the Maximum Number of Rows per Page

The number of results that are fetched per page is exposed to the user as the `limit` parameter. It is generally undesirable to allow users to fetch all rows in a paginated set. The `maxLimit` option asserts that no one can set this limit too high from the outside. By default CakePHP limits the maximum number of rows that can be fetched to 100. If this default is not appropriate for your application, you can adjust it as part of the pagination options, for example reducing it to `10`:

```
protected array $paginate = [
    // Other keys here.
    'maxLimit' => 10
];
```

If the request's limit param is greater than this value, it will be reduced to the `maxLimit` value.

Out of Range Page Requests

`Controller::paginate()` will throw a `NotFoundException` when trying to access a non-existent page, i.e. page number requested is greater than total page count.

So you could either let the normal error page be rendered or use a try catch block and take appropriate action when a `NotFoundException` is caught:

```
use Cake\Http\Exception\NotFoundException;

public function index()
{
    try {
        $this->paginate();
    } catch (NotFoundException $e) {
        // Do something here like redirecting to first or last page.
        // $e->getPrevious()->getAttributes('pagingParams') will give you required info.
    }
}
```

Using a paginator class directly

You can also use a paginator directly.:

```
// Create a paginator
$paginator = new \Cake\Datasource\Paginator\NumericPaginator();

// Paginate the model
$results = $paginator->paginate(
    // Query or table instance which you need to paginate
    $this->fetchTable('Articles'),
    // Request params
    $this->request->getQueryParams(),
    // Config array having the same structure as options as Controller::$paginate
    [
        'finder' => 'latest',
    ]
);
```

Pagination in the View

Check the *PaginatorHelper* documentation for how to create links for pagination navigation.

CHAPTER 25

Plugins

CakePHP allows you to set up a combination of controllers, models, and views and release them as a pre-packaged application plugin that others can use in their CakePHP applications. If you've created great user management, a simple blog, or web service adapters in one of your applications, why not package it as a CakePHP plugin? This way you can reuse it in your other applications, and share with the community!

A CakePHP plugin is separate from the host application itself and generally provides some well-defined functionality that can be packaged up neatly, and reused with little effort in other applications. The application and the plugin operate in their own respective spaces, but share the application's configuration data (for example, database connections, email transports)

Plugin should define their own top-level namespace. For example: `DebugKit`. By convention, plugins use their package name as their namespace. If you'd like to use a different namespace, you can configure the plugin namespace, when plugins are loaded.

Installing a Plugin With Composer

Many plugins are available on Packagist[149] and can be installed with `Composer`. To install DebugKit, you would do the following:

```
php composer.phar require cakephp/debug_kit
```

This would install the latest version of DebugKit and update your **composer.json**, **composer.lock** file, update **vendor/cakephp-plugins.php**, and update your autoloader.

[149] https://packagist.org

Manually Installing a Plugin

If the plugin you want to install is not available on packagist.org, you can clone or copy the plugin code into your **plugins** directory. Assuming you want to install a plugin named 'ContactManager', you should have a folder in **plugins** named 'ContactManager'. In this directory are the plugin's src, tests and any other directories.

Manually Autoloading Plugin Classes

If you install your plugins via `composer` or `bake` you shouldn't need to configure class autoloading for your plugins.

If you create a plugin manually under the `plugins` folder then will need to tell `composer` to refresh its autoloading cache:

```
php composer.phar dumpautoload
```

If you are using vendor namespaces for your plugins, you'll have to add the namespace to path mapping to the `composer.json` resembling the following before running the above composer command:

```
{
    "autoload": {
        "psr-4": {
            "AcmeCorp\\Users\\": "plugins/AcmeCorp/Users/src/",
        }
    },
    "autoload-dev": {
        "psr-4": {
            "AcmeCorp\\Users\\Test\\": "plugins/AcmeCorp/Users/tests/"
        }
    }
}
```

Loading a Plugin

If you want to use a plugin's routes, console commands, middlewares, event listeners, templates or webroot assets you will need to load the plugin.

If you just want to use helpers, behaviors or components from a plugin you do not need to explicitly load a plugin yet it's recommended to always do so.

There is also a handy console command to load the plugin. Execute the following line:

```
bin/cake plugin load ContactManager
```

This would update the array in your application's `config/plugins.php` with an entry similar to `'ContactManager' => []`.

Plugin Hook Configuration

Plugins offer several hooks that allow a plugin to inject itself into the appropriate parts of your application. The hooks are:

- `bootstrap` Used to load plugin default configuration files, define constants and other global functions.
- `routes` Used to load routes for a plugin. Fired after application routes are loaded.
- `middleware` Used to add plugin middleware to an application's middleware queue.
- `console` Used to add console commands to an application's command collection.
- `services` Used to register application container services

By default all plugins hooks are enabled. You can disable hooks by using the related options of the `plugin load` command:

```
bin/cake plugin load ContactManager --no-routes
```

This would update the array in your application's `config/plugins.php` with an entry similar to `'ContactManager' => ['routes' => false]`.

Plugin Loading Options

Apart from the options for plugin hooks the `plugin load` command has the following options to control plugin loading:

- `--only-debug` Load the plugin only when debug mode is enabled.
- `--only-cli` Load the plugin only for CLI.
- `--optional` Do not throw an error if the plugin is not available.

Loading plugins through `Application::bootstrap()`

Apart from the config array in `config/plugins.php`, plugins can also be loaded in your application's `bootstrap()` method:

```
// In src/Application.php
use Cake\Http\BaseApplication;
use ContactManager\ContactManagerPlugin;

class Application extends BaseApplication
{
    public function bootstrap()
    {
        parent::bootstrap();

        // Load the contact manager plugin by class name
        $this->addPlugin(ContactManagerPlugin::class);

        // Load a plugin with a vendor namespace by 'short name' with options
        $this->addPlugin('AcmeCorp/ContactManager', ['console' => false]);
```

(continues on next page)

(continued from previous page)

```
        // Load a dev dependency that will not exist in production builds.
        $this->addOptionalPlugin('AcmeCorp/ContactManager');
    }
}
```

You can configure hooks with array options, or the methods provided by plugin classes:

```
// In Application::bootstrap()
use ContactManager\ContactManagerPlugin;

// Use the disable/enable to configure hooks.
$plugin = new ContactManagerPlugin();

$plugin->disable('bootstrap');
$plugin->enable('routes');
$this->addPlugin($plugin);
```

Plugin classes also know their names and path information:

```
$plugin = new ContactManagerPlugin();

// Get the plugin name.
$name = $plugin->getName();

// Path to the plugin root, and other paths.
$path = $plugin->getPath();
$path = $plugin->getConfigPath();
$path = $plugin->getClassPath();
```

Using Plugin Classes

You can reference a plugin's controllers, models, components, behaviors, and helpers by prefixing the name of the plugin.

For example, say you wanted to use the ContactManager plugin's `ContactInfoHelper` to output formatted contact information in one of your views. In your controller, using `addHelper()` could look like this:

```
$this->viewBuilder()->addHelper('ContactManager.ContactInfo');
```

Note: This dot separated class name is referred to as *plugin syntax*.

You would then be able to access the `ContactInfoHelper` just like any other helper in your view, such as:

```
echo $this->ContactInfo->address($contact);
```

Plugins can use the models, components, behaviors and helpers provided by the application, or other plugins if necessary:

```
// Use an application component
$this->loadComponent('AppFlash');

// Use another plugin's behavior
$this->addBehavior('OtherPlugin.AuditLog');
```

Creating Your Own Plugins

As a working example, let's begin to create the ContactManager plugin referenced above. To start out, we'll set up our plugin's basic directory structure. It should look like this:

```
/src
/plugins
    /ContactManager
        /config
        /src
            /ContactManagerPlugin.php
            /Controller
                /Component
            /Model
                /Table
                /Entity
                /Behavior
            /View
                /Helper
        /templates
            /layout
        /tests
            /TestCase
            /Fixture
        /webroot
```

Note the name of the plugin folder, '**ContactManager**'. It is important that this folder has the same name as the plugin.

Inside the plugin folder, you'll notice it looks a lot like a CakePHP application, and that's basically what it is. Just instead of an `Application.php` you have a `ContactManagerPlugin.php`. You don't have to include any of the folders you are not using. Some plugins might only define a Component and a Behavior, and in that case they can completely omit the 'templates' directory.

A plugin can also have basically any of the other directories that your application can, such as Config, Console, webroot, etc.

Creating a Plugin Using Bake

The process of creating plugins can be greatly simplified by using bake.

In order to bake a plugin, use the following command:

```
bin/cake bake plugin ContactManager
```

Bake can be used to create classes in your plugin. For example to generate a plugin controller you could run:

```
bin/cake bake controller --plugin ContactManager Contacts
```

Please refer to the chapter /bake/usage if you have any problems with using the command line. Be sure to re-generate your autoloader once you've created your plugin:

```
php composer.phar dumpautoload
```

Plugin Classes

Plugin classes allow a plugin author to define set-up logic, define default hooks, load routes, middleware and console commands. Plugin classes live in **src/{PluginName}Plugin.php**. For our ContactManager plugin, our plugin class could look like:

```
namespace ContactManager;

use Cake\Core\BasePlugin;
use Cake\Core\ContainerInterface;
use Cake\Core\PluginApplicationInterface;
use Cake\Console\CommandCollection;
use Cake\Http\MiddlewareQueue;
use Cake\Routing\RouteBuilder;

class ContactManagerPlugin extends BasePlugin
{
    /**
     * @inheritDoc
     */
    public function middleware(MiddlewareQueue $middleware): MiddlewareQueue
    {
        // Add middleware here.
        $middleware = parent::middleware($middleware);

        return $middleware;
    }

    /**
     * @inheritDoc
     */
    public function console(CommandCollection $commands): CommandCollection
    {
        // Add console commands here.
        $commands = parent::console($commands);

        return $commands;
    }

    /**
     * @inheritDoc
     */
    public function bootstrap(PluginApplicationInterface $app): void
    {
```

(continues on next page)

(continued from previous page)

```
        // Add constants, load configuration defaults.
        // By default will load `config/bootstrap.php` in the plugin.
        parent::bootstrap($app);
    }

    /**
     * @inheritDoc
     */
    public function routes(RouteBuilder $routes): void
    {
        // Add routes.
        // By default will load `config/routes.php` in the plugin.
        parent::routes($routes);
    }

    /**
     * Register application container services.
     *
     * @param \Cake\Core\ContainerInterface $container The Container to update.
     * @return void
     * @link https://book.cakephp.org/5/en/development/dependency-injection.html
↪#dependency-injection
     */
    public function services(ContainerInterface $container): void
    {
        // Add your services here
    }
}
```

Plugin Routes

Plugins can provide routes files containing their routes. Each plugin can contain a **config/routes.php** file. This routes file can be loaded when the plugin is added, or in the application's routes file. To create the ContactManager plugin routes, put the following into **plugins/ContactManager/config/routes.php**:

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

$routes->plugin(
    'ContactManager',
    ['path' => '/contact-manager'],
    function ($routes) {
        $routes->setRouteClass(DashedRoute::class);

        $routes->get('/contacts', ['controller' => 'Contacts']);
        $routes->get('/contacts/{id}', ['controller' => 'Contacts', 'action' => 'view']);
        $routes->put('/contacts/{id}', ['controller' => 'Contacts', 'action' => 'update
↪']);
    }
);
```

The above will connect default routes for your plugin. You can customize this file with more specific routes later on.

You can also load plugin routes in your application's routes list. Doing this provides you more control on how plugin routes are loaded and allows you to wrap plugin routes in additional scopes or prefixes:

```
$routes->scope('/', function ($routes) {
    // Connect other routes.
    $routes->scope('/backend', function ($routes) {
        $routes->loadPlugin('ContactManager');
    });
});
```

The above would result in URLs like `/backend/contact-manager/contacts`.

Plugin Controllers

Controllers for our ContactManager plugin will be stored in **plugins/ContactManager/src/Controller/**. Since the main thing we'll be doing is managing contacts, we'll need a ContactsController for this plugin.

So, we place our new ContactsController in **plugins/ContactManager/src/Controller** and it looks like so:

```
// plugins/ContactManager/src/Controller/ContactsController.php
namespace ContactManager\Controller;

use ContactManager\Controller\AppController;

class ContactsController extends AppController
{
    public function index()
    {
        //...
    }
}
```

Also make the `AppController` if you don't have one already:

```
// plugins/ContactManager/src/Controller/AppController.php
namespace ContactManager\Controller;

use App\Controller\AppController as BaseController;

class AppController extends BaseController
{
}
```

A plugin's `AppController` can hold controller logic common to all controllers in a plugin but is not required if you don't want to use one.

If you want to access what we've got going thus far, visit `/contact-manager/contacts`. You should get a "Missing Model" error because we don't have a Contact model defined yet.

If your application includes the default routing CakePHP provides you will be able to access your plugin controllers using URLs like:

```
// Access the index route of a plugin controller.
/contact-manager/contacts

// Any action on a plugin controller.
/contact-manager/contacts/view/1
```

If your application defines routing prefixes, CakePHP's default routing will also connect routes that use the following pattern:

```
/{prefix}/{plugin}/{controller}
/{prefix}/{plugin}/{controller}/{action}
```

See the section on *Plugin Hook Configuration* for information on how to load plugin specific route files.

Plugin Models

Models for the plugin are stored in **plugins/ContactManager/src/Model**. We've already defined a ContactsController for this plugin, so let's create the table and entity for that 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
{
}
```

If you need to reference a model within your plugin when building associations or defining entity classes, you need to include the plugin name with the class name, separated with a dot. For example:

```
// plugins/ContactManager/src/Model/Table/ContactsTable.php:
namespace ContactManager\Model\Table;

use Cake\ORM\Table;

class ContactsTable extends Table
{
    public function initialize(array $config): void
    {
        $this->hasMany('ContactManager.AltName');
    }
}
```

If you would prefer that the array keys for the association not have the plugin prefix on them, use the alternative syntax:

```
// plugins/ContactManager/src/Model/Table/ContactsTable.php:
namespace ContactManager\Model\Table;

use Cake\ORM\Table;

class ContactsTable extends Table
{
    public function initialize(array $config): void
    {
        $this->hasMany('AltName', [
            'className' => 'ContactManager.AltName',
        ]);
    }
}
```

You can use `Cake\ORM\Locator\LocatorAwareTrait` to load your plugin tables using the familiar *plugin syntax*:

```
// Controllers already use LocatorAwareTrait, so you don't need this.
use Cake\ORM\Locator\LocatorAwareTrait;

$contacts = $this->fetchTable('ContactManager.Contacts');
```

Plugin Templates

Views behave exactly as they do in normal applications. Just place them in the right folder inside of the `plugins/[PluginName]/templates/` folder. For our ContactManager plugin, we'll need a view for our `ContactsController::index()` action, so let's include that as well:

```
// plugins/ContactManager/templates/Contacts/index.php:
<h1>Contacts</h1>
<p>Following is a sortable list of your contacts</p>
<!-- A sortable list of contacts would go here....-->
```

Plugins can provide their own layouts. To add plugin layouts, place your template files inside `plugins/[PluginName]/templates/layout`. To use a plugin layout in your controller you can do the following:

```
$this->viewBuilder()->setLayout('ContactManager.admin');
```

If the plugin prefix is omitted, the layout/view file will be located normally.

Note: For information on how to use elements from a plugin, look up *Elements*

Overriding Plugin Templates from Inside Your Application

You can override any plugin views from inside your app using special paths. If you have a plugin called 'ContactManager' you can override the template files of the plugin with application specific view logic by creating files using the following template **templates/plugin/[Plugin]/[Controller]/[view].php**. For the Contacts controller you could make the following file:

```
templates/plugin/ContactManager/Contacts/index.php
```

Creating this file would allow you to override **plugins/ContactManager/templates/Contacts/index.php**.

If your plugin is in a composer dependency (i.e. 'Company/ContactManager'), the path to the 'index' view of the Contacts controller will be:

```
templates/plugin/TheVendor/ThePlugin/Custom/index.php
```

Creating this file would allow you to override **vendor/thevendor/theplugin/templates/Custom/index.php**.

If the plugin implements a routing prefix, you must include the routing prefix in your application template overrides. For example, if the 'ContactManager' plugin implemented an 'Admin' prefix the overriding path would be:

```
templates/plugin/ContactManager/Admin/ContactManager/index.php
```

Plugin Assets

A plugin's web assets (but not PHP files) can be served through the plugin's `webroot` directory, just like the main application's assets:

```
/plugins/ContactManager/webroot/
                              css/
                              js/
                              img/
                              flash/
                              pdf/
```

You may put any type of file in any directory, just like a regular webroot.

> **Warning:** Handling static assets (such as images, JavaScript and CSS files) through the Dispatcher is very inefficient. See *Improve Your Application's Performance* for more information.

Linking to Assets in Plugins

You can use the *plugin syntax* when linking to plugin assets using the `HtmlHelper`'s script, image, or css methods:

```
// Generates a URL of /contact_manager/css/styles.css
echo $this->Html->css('ContactManager.styles');

// Generates a URL of /contact_manager/js/widget.js
echo $this->Html->script('ContactManager.widget');
```

(continues on next page)

(continued from previous page)

```
// Generates a URL of /contact_manager/img/logo.jpg
echo $this->Html->image('ContactManager.logo');
```

Plugin assets are served using the `AssetMiddleware` middleware by default. This is only recommended for development. In production you should *symlink plugin assets* to improve performance.

If you are not using the helpers, you can prepend /plugin-name/ to the beginning of the URL for an asset within that plugin to serve it. Linking to '/contact_manager/js/some_file.js' would serve the asset **plugins/ContactManager/webroot/js/some_file.js**.

Components, Helpers and Behaviors

A plugin can have Components, Helpers and Behaviors just like a CakePHP application. You can even create plugins that consist only of Components, Helpers or Behaviors which can be a great way to build reusable components that can be dropped into any project.

Building these components is exactly the same as building it within a regular application, with no special naming convention.

Referring to your component from inside or outside of your plugin requires only that you prefix the plugin name before the name of the component. For example:

```
// Component defined in 'ContactManager' plugin
namespace ContactManager\Controller\Component;

use Cake\Controller\Component;

class ExampleComponent extends Component
{
}

// Within your controllers
public function initialize(): void
{
    parent::initialize();
    $this->loadComponent('ContactManager.Example');
}
```

The same technique applies to Helpers and Behaviors.

Commands

Plugins can register their commands inside the `console()` hook. By default all console commands in the plugin are auto-discovered and added to the application's command list. Plugin commands are prefixed with the plugin name. For example, the `UserCommand` provided by the `ContactManager` plugin would be registered as both `contact_manager.user` and `user`. The un-prefixed name will only be taken by a plugin if it is not used by the application, or another plugin.

You can customize the command names by defining each command in your plugin:

```
public function console($commands)
{
    // Create nested commands
    $commands->add('bake model', ModelCommand::class);
    $commands->add('bake controller', ControllerCommand::class);

    return $commands;
}
```

Testing your Plugin

If you are testing controllers or generating URLs, make sure your plugin connects routes `tests/bootstrap.php`.

For more information see *testing plugins* page.

Publishing your Plugin

CakePHP plugins should be published to the packagist[150]. This way other people can use it as composer dependency. You can also propose your plugin to the awesome-cakephp list[151].

Choose a semantically meaningful name for the package name. This should ideally be prefixed with the dependency, in this case "cakephp" as the framework. The vendor name will usually be your GitHub username. Do **not** use the CakePHP namespace (cakephp) as this is reserved to CakePHP owned plugins. The convention is to use lowercase letters and dashes as separator.

So if you created a plugin "Logging" with your GitHub account "FooBar", a good name would be *foo-bar/cakephp-logging*. And the CakePHP owned "Localized" plugin can be found under *cakephp/localized* respectively.

Plugin Map File

When installing plugins via Composer, you may notice that **vendor/cakephp-plugins.php** is created. This configuration file contains a map of plugin names and their paths on the filesystem. It makes it possible for plugins to be installed into the standard vendor directory which is outside of the normal search paths. The `Plugin` class will use this file to locate plugins when they are loaded with `addPlugin()`. You generally won't need to edit this file by hand, as Composer and the `plugin-installer` package will manage it for you.

Manage Your Plugins using Mixer

Another way to discover and manage plugins into your CakePHP application is Mixer[152]. It is a CakePHP plugin which helps you to install plugins from Packagist. It also helps you to manage your existing plugins.

Note: IMPORTANT: Do not use this in production environment.

[150] https://packagist.org
[151] https://github.com/FriendsOfCake/awesome-cakephp
[152] https://github.com/CakeDC/mixer

CHAPTER 26

REST

REST is a foundational concept to the open web. CakePHP provides functionality to build applications that expose REST APIs with low complexity abstractions and interfaces.

CakePHP provides methods for exposing your controller actions via HTTP methods, and serializing view variables based on content-type negotiation. Content-Type negotiation allows clients of your application to send requests with serialize data and receive responses with serialized data via the `Accept` and `Content-Type` headers, or URL extensions.

Getting Started

To get started with adding a REST API to your application, we'll first need a controller containing actions that we want to expose as an API. A basic controller might look something like this:

```
// src/Controller/RecipesController.php
use Cake\View\JsonView;

class RecipesController extends AppController
{
    public function viewClasses(): array
    {
        return [JsonView::class];
    }

    public function index()
    {
        $recipes = $this->Recipes->find('all')->all();
        $this->set('recipes', $recipes);
        $this->viewBuilder()->setOption('serialize', ['recipes']);
```

(continues on next page)

(continued from previous page)

```
    }

    public function view($id)
    {
        $recipe = $this->Recipes->get($id);
        $this->set('recipe', $recipe);
        $this->viewBuilder()->setOption('serialize', ['recipe']);
    }

    public function add()
    {
        $this->request->allowMethod(['post', 'put']);
        $recipe = $this->Recipes->newEntity($this->request->getData());
        if ($this->Recipes->save($recipe)) {
            $message = 'Saved';
        } else {
            $message = 'Error';
        }
        $this->set([
            'message' => $message,
            'recipe' => $recipe,
        ]);
        $this->viewBuilder()->setOption('serialize', ['recipe', 'message']);
    }

    public function edit($id)
    {
        $this->request->allowMethod(['patch', 'post', 'put']);
        $recipe = $this->Recipes->get($id);
        $recipe = $this->Recipes->patchEntity($recipe, $this->request->getData());
        if ($this->Recipes->save($recipe)) {
            $message = 'Saved';
        } else {
            $message = 'Error';
        }
        $this->set([
            'message' => $message,
            'recipe' => $recipe,
        ]);
        $this->viewBuilder()->setOption('serialize', ['recipe', 'message']);
    }

    public function delete($id)
    {
        $this->request->allowMethod(['delete']);
        $recipe = $this->Recipes->get($id);
        $message = 'Deleted';
        if (!$this->Recipes->delete($recipe)) {
            $message = 'Error';
        }
        $this->set('message', $message);
        $this->viewBuilder()->setOption('serialize', ['message']);
```

(continues on next page)

(continued from previous page)

```
    }
}
```

In our `RecipesController`, we have several actions that define the logic to create, edit, view and delete recipes. In each of our actions we're using the `serialize` option to tell CakePHP which view variables should be serialized when making API responses. We'll connect our controller to the application URLs with *RESTful Routing*:

```
// in config/routes.php
$routes->scope('/', function (RouteBuilder $routes): void {
    $routes->setExtensions(['json']);
    $routes->resources('Recipes');
});
```

These routes will enable URLs like `/recipes.json` to return a JSON encoded response. Clients could also make a request to `/recipes` with the `Content-Type: application/json` header as well.

Encoding Response Data

In the above controller, we're defining a `viewClasses()` method. This method defines which views your controller has available for content-negotitation. We're including CakePHP's `JsonView` which enables JSON based responses. To learn more about it and Xml based views see *JSON and XML views*. is used by CakePHP to select a view class to render a REST response with.

Next, we have several methods that expose basic logic to create, edit, view and delete recipes. In each of our actions we're using the `serialize` option to tell CakePHP which view variables should be serialized when making API responses.

If we wanted to modify the data before it is converted into JSON we should not define the `serialize` option, and instead use template files. We would place the REST templates for our RecipesController inside **templates/Recipes/json**.

See the *Content Type Negotiation* for more information on how CakePHP's response negotiation functionality.

Parsing Request Bodies

Creating the logic for the edit action requires another step. Because our resources are serialized as JSON it would be ergonomic if our requests also contained the JSON representation.

In our `Application` class ensure the following is present:

```
$middlewareQueue->add(new BodyParserMiddleware());
```

This middleware will use the `content-type` header to detect the format of request data and parse enabled formats. By default only `JSON` parsing is enabled by default. You can enable XML support by enabling the `xml` constructor option. When a request is made with a `Content-Type` of `application/json`, CakePHP will decode the request data and update the request so that `$request->getData()` contains the parsed body.

You can also wire in additional deserializers for alternate formats if you need them, using `BodyParserMiddleware::addParser()`.

CHAPTER 27

Security

CakePHP provides you some tools to secure your application. The following sections cover those tools:

Security Utility

class Cake\Utility**Security**

The security library[153] handles basic security measures such as providing methods for hashing and encrypting data.

Encrypting and Decrypting Data

static Cake\Utility\Security::**encrypt**(*$text*, *$key*, *$hmacSalt = null*)

static Cake\Utility\Security::**decrypt**(*$cipher*, *$key*, *$hmacSalt = null*)

Encrypt `$text` using AES-256. The `$key` should be a value with a lots of variance in the data much like a good password. The returned result will be the encrypted value with an HMAC checksum.

The openssl[154] extension is required for encrypting/decrypting.

An example use would be:

```
// Assuming key is stored somewhere it can be re-used for
// decryption later.
$key = 'wt1U5MACWJFTXGenFoZoiLwQGrLgdbHA';
$result = Security::encrypt($value, $key);
```

[153] https://api.cakephp.org/5.x/class-Cake.Utility.Security.html
[154] https://php.net/openssl

If you do not supply an HMAC salt, the value of `Security::getSalt()` will be used. Encrypted values can be decrypted using *Cake\Utility\Security::decrypt()*.

This method should **never** be used to store passwords.

Decrypt a previously encrypted value. The `$key` and `$hmacSalt` parameters must match the values used to encrypt or decryption will fail. An example use would be:

```
// Assuming the key is stored somewhere it can be re-used for
// Decryption later.
$key = 'wt1U5MACWJFTXGenFoZoiLwQGrLgdbHA';

$cipher = $user->secrets;
$result = Security::decrypt($cipher, $key);
```

If the value cannot be decrypted due to changes in the key or HMAC salt `false` will be returned.

Hashing Data

static Cake\Utility\Security::**hash**(*$string*, *$type = NULL*, *$salt = false*)

Create a hash from string using given method. Fallback on next available method. If `$salt` is set to `true`, the application's salt value will be used:

```
// Using the application's salt value
$sha1 = Security::hash('CakePHP Framework', 'sha1', true);

// Using a custom salt value
$sha1 = Security::hash('CakePHP Framework', 'sha1', 'my-salt');

// Using the default hash algorithm
$hash = Security::hash('CakePHP Framework');
```

The `hash()` method supports the following hashing strategies:

- md5
- sha1
- sha256

And any other hash algorithm that PHP's `hash()` function supports.

Warning: You should not be using `hash()` for passwords in new applications. Instead you should use the `DefaultPasswordHasher` class which uses bcrypt by default.

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.

static Cake\Utility\Security::**randomString**(*$length*)

Get a random string `$length` long from a secure random source. This method draws from the same random source as `randomBytes()` and will encode the data as a hexadecimal string.

CSRF Protection

Cross-Site Request Forgeries (CSRF) are a class of exploit where unauthorized commands are performed on behalf of an authenticated user without their knowledge or consent.

CakePHP offers two forms of CSRF protection:

- `SessionCsrfProtectionMiddleware` stores CSRF tokens in the session. This requires that your application opens the session on every request with side-effects. The benefits of session based CSRF tokens is that they are scoped to a specific user, and only valid for the duration a session is live.
- `CsrfProtectionMiddleware` stores CSRF tokens in a cookie. Using a cookie allows CSRF checks to be done without any state on the server. Cookie values are verified for authenticity using an HMAC check. However, due to their stateless nature, CSRF tokens are re-usable across users and sessions.

Note: You cannot use both of the following approaches together, you must choose only one. If you use both approaches together, a CSRF token mismatch error will occur on every *PUT* and *POST* request

---

Cross Site Request Forgery (CSRF) Middleware

CSRF protection can be applied to your entire application, or to specific routing scopes. By applying a CSRF middleware to your Application middleware stack you protect all the actions in application:

```
// in src/Application.php
// For Cookie based CSRF tokens.
use Cake\Http\Middleware\CsrfProtectionMiddleware;

// For Session based CSRF tokens.
use Cake\Http\Middleware\SessionCsrfProtectionMiddleware;

public function middleware(MiddlewareQueue $middlewareQueue): MiddlewareQueue
{
    $options = [
        // ...
```

(continues on next page)

(continued from previous page)

```
    ];
    $csrf = new CsrfProtectionMiddleware($options);
    // or
    $csrf = new SessionCsrfProtectionMiddleware($options);

    $middlewareQueue->add($csrf);
    return $middlewareQueue;
}
```

By applying CSRF protection to routing scopes, you can conditionally apply CSRF to specific groups of routes:

```
// in src/Application.php
use Cake\Http\Middleware\CsrfProtectionMiddleware;

public function routes(RouteBuilder $routes) : void
{
    $options = [
        // ...
    ];
    $routes->registerMiddleware('csrf', new CsrfProtectionMiddleware($options));
    parent::routes($routes);
}

// in config/routes.php
$routes->scope('/', function (RouteBuilder $routes) {
    $routes->applyMiddleware('csrf');
});
```

Cookie based CSRF middleware options

The available configuration options are:

- `cookieName` The name of the cookie to send. Defaults to `csrfToken`.
- `expiry` How long the CSRF token should last. Defaults to browser session.
- `secure` Whether or not the cookie will be set with the Secure flag. That is, the cookie will only be set on a HTTPS connection and any attempt over normal HTTP will fail. Defaults to `false`.
- `httponly` Whether or not the cookie will be set with the HttpOnly flag. Defaults to `false`. Prior to 4.1.0 use the `httpOnly` option.
- `samesite` Allows you to declare if your cookie should be restricted to a first-party or same-site context. Possible values are `Lax`, `Strict` and `None`. Defaults to `null`.
- `field` The form field to check. Defaults to `_csrfToken`. Changing this will also require configuring FormHelper.

Session based CSRF middleware options

The available configuration options are:

- `key` The session key to use. Defaults to *csrfToken*
- `field` The form field to check. Changing this will also require configuring FormHelper.

When enabled, you can access the current CSRF token on the request object:

```
$token = $this->request->getAttribute('csrfToken');
```

Should you need to rotate or replace the session CSRF token you can do so with:

```
$this->request = SessionCsrfProtectionMiddleware::replaceToken($this->request);
```

New in version 4.3.0: The `replaceToken` method was added.

Skipping CSRF checks for specific actions

Both CSRF middleware implementations allow you to the skip check callback feature for more fine grained control over URLs for which CSRF token check should be done:

```
// in src/Application.php
use Cake\Http\Middleware\CsrfProtectionMiddleware;

public function middleware(MiddlewareQueue $middlewareQueue): MiddlewareQueue
{
    $csrf = new CsrfProtectionMiddleware();

    // Token check will be skipped when callback returns `true`.
    $csrf->skipCheckCallback(function ($request) {
        // Skip token check for API URLs.
        if ($request->getParam('prefix') === 'Api') {
            return true;
        }
    });

    // Ensure routing middleware is added to the queue before CSRF protection middleware.
    $middlewareQueue->add($csrf);

    return $middlewareQueue;
}
```

Note: You should apply the CSRF protection middleware only for routes which handle stateful requests using cookies/sessions. For example, when developing an API, stateless requests that do not use cookies for authentication are not affected by CSRF so the middleware does not need to be applied for those routes.

Integration with FormHelper

The `CsrfProtectionMiddleware` integrates seamlessly with `FormHelper`. Each time you create a form with `FormHelper`, it will insert a hidden field containing the CSRF token.

Note: When using CSRF protection you should always start your forms with the `FormHelper`. If you do not, you will need to manually create hidden inputs in each of your forms.

CSRF Protection and AJAX Requests

In addition to request data parameters, CSRF tokens can be submitted through a special `X-CSRF-Token` header. Using a header often makes it easier to integrate a CSRF token with JavaScript heavy applications, or XML/JSON based API endpoints.

The CSRF Token can be obtained in JavaScript via the Cookie `csrfToken`, or in PHP via the request object attribute named `csrfToken`. Using the cookie might be easier when your JavaScript code resides in files separate from the CakePHP view templates, and when you already have functionality for parsing cookies via JavaScript.

If you have separate JavaScript files but don't want to deal with handling cookies, you could for example set the token in a global JavaScript variable in your layout, by defining a script block like this:

```
echo $this->Html->scriptBlock(sprintf(
    'var csrfToken = %s;',
    json_encode($this->request->getAttribute('csrfToken'))
));
```

You can then access the token as `csrfToken` or `window.csrfToken` in any script file that is loaded after this script block.

Another alternative would be to put the token in a custom meta tag like this:

```
echo $this->Html->meta('csrfToken', $this->request->getAttribute('csrfToken'));
```

which could then be accessed in your scripts by looking for the `meta` element with the name `csrfToken`, which could be as simple as this when using jQuery:

```
var csrfToken = $('meta[name="csrfToken"]').attr('content');
```

Content Security Policy Middleware

The `CspMiddleware` makes it simpler to add Content-Security-Policy headers in your application. Before using it you should install `paragonie/csp-builder`:

```
composer require paragonie/csp-builder
```

You can then configure the middleware using an array, or passing in a built `CSPBuilder` object:

```
use Cake\Http\Middleware\CspMiddleware;

$csp = new CspMiddleware([
    'script-src' => [
```

(continues on next page)

(continued from previous page)

```
        'allow' => [
            'https://www.google-analytics.com',
        ],
        'self' => true,
        'unsafe-inline' => false,
        'unsafe-eval' => false,
    ],
]);

$middlewareQueue->add($csp);
```

If you want to use a more strict CSP configuration, you can enable nonce based CSP rules with the `scriptNonce` and `styleNonce` options. When enabled these options will modify your CSP policy and set the `cspScriptNonce` and `cspStyleNonce` attributes in the request. These attributes are applied to the `nonce` attribute of all script and CSS link elements created by `HtmlHelper`. This simplifies the adoption of policies that use a nonce-base64[155] and `strict-dynamic` for increased security and easier maintenance:

```
$policy = [
    // Must exist even if empty to set nonce for for script-src
    'script-src' => [],
    'style-src' => [],
];
// Enable automatic nonce addition to script & CSS link tags.
$csp = new CspMiddleware($policy, [
    'scriptNonce' => true,
    'styleNonce' => true,
]);
$middlewareQueue->add($csp);
```

Security Header Middleware

The `SecurityHeaderMiddleware` layer allows you to apply security related headers to your application. Once setup the middleware can apply the following headers to responses:

- `X-Content-Type-Options`
- `X-Download-Options`
- `X-Frame-Options`
- `Referrer-Policy`

This middleware is configured using a fluent interface before it is applied to your application's middleware stack:

```
use Cake\Http\Middleware\SecurityHeadersMiddleware;

$securityHeaders = new SecurityHeadersMiddleware();
$securityHeaders
    ->setReferrerPolicy()
    ->setXFrameOptions()
    ->noOpen()
```

(continues on next page)

[155] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/script-src

(continued from previous page)

```
    ->noSniff();

$middlewareQueue->add($securityHeaders);
```

Here's a list of common HTTP headers[156], and the Mozilla recommended settings[157] for securing web applications.

HTTPS Enforcer Middleware

If you want your application to only be available via HTTPS connections you can use the `HttpsEnforcerMiddleware`:

```
use Cake\Http\Middleware\HttpsEnforcerMiddleware;

// Always raise an exception and never redirect.
$https = new HttpsEnforcerMiddleware([
    'redirect' => false,
]);

// Send a 302 status code when redirecting
$https = new HttpsEnforcerMiddleware([
    'redirect' => true,
    'statusCode' => 302,
]);

// Send additional headers in the redirect response.
$https = new HttpsEnforcerMiddleware([
    'headers' => ['X-Https-Upgrade' => 1],
]);

// Disable HTTPs enforcement when ``debug`` is on.
$https = new HttpsEnforcerMiddleware([
    'disableOnDebug' => true,
]);

// Only trust HTTP_X_ headers from the listed servers.
$https = new HttpsEnforcerMiddleware([
    'trustProxies' => ['192.168.1.1'],
]);
```

If a non-HTTP request is received that does not use GET a `BadRequestException` will be raised.

NOTE: The Strict-Transport-Security header is ignored by the browser when your site has only been accessed using HTTP. Once your site is accessed over HTTPS with no certificate errors, the browser knows your site is HTTPS capable and will honor the Strict-Transport-Security header.

[156] https://en.wikipedia.org/wiki/List_of_HTTP_header_fields
[157] https://infosec.mozilla.org/guidelines/web_security.html

Adding Strict-Transport-Security

When your application requires SSL it is a good idea to set the `Strict-Transport-Security` header. This header value is cached in the browser, and informs browsers that they should always connect with HTTPS connections. You can configure this header with the `hsts` option:

```
$https = new HttpsEnforcerMiddleware([
    'hsts' => [
        // How long the header value should be cached for.
        'maxAge' => 60 * 60 * 24 * 365,
        // should this policy apply to subdomains?
        'includeSubDomains' => true,
        // Should the header value be cacheable in google's HSTS preload
        // service? While not part of the spec it is widely implemented.
        'preload' => true,
    ],
]);
```

CHAPTER 28

Sessions

CakePHP provides a wrapper and suite of utility features on top of PHP's native `session` extension. Sessions allow you to identify unique users across requests and store persistent data for specific users. Unlike Cookies, session data is not available on the client side. Usage of `$_SESSION` is generally avoided in CakePHP, and instead usage of the Session classes is preferred.

Session Configuration

Session configuration is generally defined in **/config/app.php**. The available options are:

- `Session.timeout` - The number of *minutes* a session can remain 'idle'. If no request is received for `timeout` minutes, CakePHP's session handler will expire the session. You can set this option to `0` to disable server side idle timeouts.
- `Session.defaults` - Allows you to use the built-in default session configurations as a base for your session configuration. See below for the built-in defaults.
- `Session.handler` - Allows you to define a custom session handler. The core database and cache session handlers use this. See below for additional information on Session handlers.
- `Session.ini` - Allows you to set additional session ini settings for your config. This combined with `Session.handler` replace the custom session handling features of previous versions
- `Session.cookie` - The name of the cookie to use. Defaults to value set for `session.name` php.ini config.
- `Session.cookiePath` - The url path for which session cookie is set. Maps to the `session.cookie_path` php.ini config. Defaults to base path of app.

CakePHP's defaults `session.cookie_secure` to `true`, when your application is on an SSL protocol. If your application serves from both SSL and non-SSL protocols, then you might have problems with sessions being lost. If you need access to the session on both SSL and non-SSL domains you will want to disable this:

```
Configure::write('Session', [
    'defaults' => 'php',
    'ini' => [
        'session.cookie_secure' => false
    ]
]);
```

CakePHP also sets the SameSite[158] attribute to `Lax` by default for session cookies, which helps protect against CSRF attacks. You can change the default value by setting `session.cookie_samesite` php.ini config:

```
Configure::write('Session', [
    'defaults' => 'php',
    'ini' => [
        'session.cookie_samesite' => 'Strict',
    ],
]);
```

The session cookie path defaults to app's base path. To change this you can use the `session.cookie_path` ini value. For example if you want your session to persist across all subdomains you can do:

```
Configure::write('Session', [
    'defaults' => 'php',
    'ini' => [
        'session.cookie_path' => '/',
        'session.cookie_domain' => '.yourdomain.com',
    ],
]);
```

By default PHP sets the session cookie to expire as soon as the browser is closed, regardless of the configured `Session.timeout` value. The cookie timeout is controlled by the `session.cookie_lifetime` ini value and can be configured using:

```
Configure::write('Session', [
    'defaults' => 'php',
    'ini' => [
        // Invalidate the cookie after 30 minutes
        'session.cookie_lifetime' => 1800
    ]
]);
```

The difference between `Session.timeout` and the `session.cookie_lifetime` value is that the latter relies on the client telling the truth about the cookie. If you require stricter timeout checking, without relying on what the client reports, you should use `Session.timeout`.

Please note that `Session.timeout` corresponds to the total time of inactivity for a user (i.e. the time without visiting any page where the session is used), and does not limit the total amount of minutes a user can stay active on the site.

[158] https://owasp.org/www-community/SameSite

Built-in Session Handlers & Configuration

CakePHP comes with several built-in session configurations. You can either use these as the basis for your session configuration, or you can create a fully custom solution. To use defaults, simply set the 'defaults' key to the name of the default you want to use. You can then override any sub setting by declaring it in your Session config:

```
Configure::write('Session', [
    'defaults' => 'php'
]);
```

The above will use the built-in 'php' session configuration. You could augment part or all of it by doing the following:

```
Configure::write('Session', [
    'defaults' => 'php',
    'cookie' => 'my_app',
    'timeout' => 4320 // 3 days
]);
```

The above overrides the timeout and cookie name for the 'php' session configuration. The built-in configurations are:

- `php` - Saves sessions with the standard settings in your php.ini file.
- `cake` - Saves sessions as files inside `tmp/sessions`. This is a good option when on hosts that don't allow you to write outside your own home dir.
- `database` - Use the built-in database sessions. See below for more information.
- `cache` - Use the built-in cache sessions. See below for more information.

Session Handlers

Session handlers can also be defined in the session config array. By defining the 'handler.engine' config key, you can name the class name, or provide a handler instance. The class/object must implement the native PHP `SessionHandlerInterface`. Implementing this interface will allow `Session` to automatically map the methods for the handler. Both the core Cache and Database session handlers use this method for saving sessions. Additional settings for the handler should be placed inside the handler array. You can then read those values out from inside your handler:

```
'Session' => [
    'handler' => [
        'engine' => 'DatabaseSession',
        'model' => 'CustomSessions',
    ],
]
```

The above shows how you could setup the Database session handler with an application model. When using class names as your handler.engine, CakePHP will expect to find your class in the `Http\Session` namespace. For example, if you had an `AppSessionHandler` class, the file should be **src/Http/Session/AppSessionHandler.php**, and the class name should be `App\Http\Session\AppSessionHandler`. You can also use session handlers from inside plugins. By setting the engine to `MyPlugin.PluginSessionHandler`.

Database Sessions

If you need to use a database to store your session data, configure as follows:

```
'Session' => [
    'defaults' => 'database'
]
```

This configuration requires a database table, having this schema:

```
CREATE TABLE `sessions` (
  `id` char(40) CHARACTER SET ascii COLLATE ascii_bin NOT NULL,
  `created` datetime DEFAULT CURRENT_TIMESTAMP, -- Optional
  `modified` datetime DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP, -- Optional
  `data` blob DEFAULT NULL, -- for PostgreSQL use bytea instead of blob
  `expires` int(10) unsigned DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

You can find a copy of the schema for the sessions table in the application skeleton[159] in **config/schema/sessions.sql**.

You can also use your own `Table` class to handle the saving of the sessions:

```
'Session' => [
    'defaults' => 'database',
    'handler' => [
        'engine' => 'DatabaseSession',
        'model' => 'CustomSessions',
    ],
]
```

The above will tell Session to use the built-in 'database' defaults, and specify that a Table called `CustomSessions` will be the delegate for saving session information to the database.

Cache Sessions

The Cache class can be used to store sessions as well. This allows you to store sessions in a cache like APCu, or Memcached. There are some caveats to using cache sessions, in that if you exhaust the cache space, sessions will start to expire as records are evicted.

To use Cache based sessions you can configure you Session config like:

```
Configure::write('Session', [
    'defaults' => 'cache',
    'handler' => [
        'config' => 'session',
    ],
]);
```

This will configure Session to use the `CacheSession` class as the delegate for saving the sessions. You can use the 'config' key which cache configuration to use. The default cache configuration is `'default'`.

[159] https://github.com/cakephp/app

Session Locking

The app skeleton comes preconfigured with a session config like this:

```
'Session' => [
    'defaults' => 'php',
],
```

This means CakePHP will handle sessions via what is configured in your `php.ini`. In most cases this will be the default configuration so PHP will save any newly created session as a file in e.g. `/var/lib/php/session`

But this also means any computationally heavy task like querying a large dataset combined with an active session will **lock that session file** - therefore blocking users to e.g. open a second tab of your app to do something else in the meantime.

To prevent this behavior you will have to change the way how sessions are being handled in CakePHP by using a different session handler like *Cache Sessions* combined with the *Redis Engine* or another cache engine.

Tip: If you want to read more about Session Locking see here[160]

Setting ini directives

The built-in defaults attempt to provide a common base for session configuration. You may need to tweak specific ini flags as well. CakePHP exposes the ability to customize the ini settings for both default configurations, as well as custom ones. The `ini` key in the session settings, allows you to specify individual configuration values. For example you can use it to control settings like `session.gc_divisor`:

```
Configure::write('Session', [
    'defaults' => 'php',
    'ini' => [
        'session.cookie_name' => 'MyCookie',
        'session.cookie_lifetime' => 1800, // Valid for 30 minutes
        'session.gc_divisor' => 1000,
        'session.cookie_httponly' => true
    ]
]);
```

Creating a Custom Session Handler

Creating a custom session handler is straightforward in CakePHP. In this example we'll create a session handler that stores sessions both in the Cache (APC) and the database. This gives us the best of fast IO of APC, without having to worry about sessions evaporating when the cache fills up.

First we'll need to create our custom class and put it in **src/Http/Session/ComboSession.php**. The class should look something like:

```
namespace App\Http\Session;
```

(continues on next page)

[160] https://ma.ttias.be/php-session-locking-prevent-sessions-blocking-in-requests/

(continued from previous page)

```
use Cake\Cache\Cache;
use Cake\Core\Configure;
use Cake\Http\Session\DatabaseSession;

class ComboSession extends DatabaseSession
{
    protected $cacheKey;

    public function __construct()
    {
        $this->cacheKey = Configure::read('Session.handler.cache');
        parent::__construct();
    }

    // Read data from the session.
    public function read($id): string
    {
        $result = Cache::read($id, $this->cacheKey);
        if ($result) {
            return $result;
        }
        return parent::read($id);
    }

    // Write data into the session.
    public function write($id, $data): bool
    {
        Cache::write($id, $data, $this->cacheKey);
        return parent::write($id, $data);
    }

    // Destroy a session.
    public function destroy($id): bool
    {
        Cache::delete($id, $this->cacheKey);
        return parent::destroy($id);
    }

    // Removes expired sessions.
    public function gc($expires = null): bool
    {
        return parent::gc($expires);
    }
}
```

Our class extends the built-in `DatabaseSession` so we don't have to duplicate all of its logic and behavior. We wrap each operation with a *Cake\Cache\Cache* operation. This lets us fetch sessions from the fast cache, and not have to worry about what happens when we fill the cache. In **config/app.php** make the session block look like:

```
'Session' => [
    'defaults' => 'database',
    'handler' => [
```

(continues on next page)

(continued from previous page)

```
        'engine' => 'ComboSession',
        'model' => 'Session',
        'cache' => 'apc',
    ],
],
// Make sure to add a apc cache config
'Cache' => [
    'apc' => ['engine' => 'Apc']
]
```

Now our application will start using our custom session handler for reading and writing session data.

class Session

Accessing the Session Object

You can access the session data any place you have access to a request object. This means the session is accessible from:

- Controllers
- Views
- Helpers
- Cells
- Components

A basic example of session usage in controllers, views and cells would be:

```
$name = $this->request->getSession()->read('User.name');

// If you are accessing the session multiple times,
// you will probably want a local variable.
$session = $this->request->getSession();
$name = $session->read('User.name');
```

In helpers, use `$this->getView()->getRequest()` to get the request object; In components, use `$this->getController()->getRequest()`.

Reading & Writing Session Data

`Session::`**read**(*$key*, *$default = null*)

You can read values from the session using `Hash::extract()` compatible syntax:

```
$session->read('Config.language', 'en');
```

`Session::`**readOrFail**(*$key*)

The same as convenience wrapper around non-nullable return value:

```
$session->readOrFail('Config.language');
```

This is useful, when you know this key has to be set and you don't want to have to check for the existence in code itself.

`Session::write`(*$key*, *$value*)

`$key` should be the dot separated path you wish to write `$value` to:

```
$session->write('Config.language', 'en');
```

You may also specify one or multiple hashes like so:

```
$session->write([
  'Config.theme' => 'blue',
  'Config.language' => 'en',
]);
```

`Session::delete`(*$key*)

When you need to delete data from the session, you can use `delete()`:

```
$session->delete('Some.value');
```

`static Session::consume`(*$key*)

When you need to read and delete data from the session, you can use `consume()`:

```
$session->consume('Some.value');
```

`Session::check`(*$key*)

If you want to see if data exists in the session, you can use `check()`:

```
if ($session->check('Config.language')) {
    // Config.language exists and is not null.
}
```

Destroying the Session

`Session::destroy`()

Destroying the session is useful when users log out. To destroy a session, use the `destroy()` method:

```
$session->destroy();
```

Destroying a session will remove all serverside data in the session, but will **not** remove the session cookie.

Rotating Session Identifiers

`Session::renew()`

While the `Authentication Plugin` automatically renews the session id when users login and logout, you may need to rotate the session id's manually. To do this use the `renew()` method:

```
$session->renew();
```

Flash Messages

Flash messages are small messages displayed to end users once. They are often used to present error messages, or confirm that actions took place successfully.

To set and display flash messages you should use *FlashComponent* and *FlashHelper*

CHAPTER 29

Testing

CakePHP comes with comprehensive testing support built-in. CakePHP comes with integration for PHPUnit[161]. In addition to the features offered by PHPUnit, CakePHP offers some additional features to make testing easier. This section will cover installing PHPUnit, and getting started with Unit Testing, and how you can use the extensions that CakePHP offers.

Installing PHPUnit

CakePHP uses PHPUnit as its underlying test framework. PHPUnit is the de-facto standard for unit testing in PHP. It offers a deep and powerful set of features for making sure your code does what you think it does. PHPUnit can be installed through using either a PHAR package[162] or Composer[163].

Install PHPUnit with Composer

To install PHPUnit with Composer:

```
$ php composer.phar require --dev phpunit/phpunit:"^10.1"
```

This will add the dependency to the `require-dev` section of your `composer.json`, and then install PHPUnit along with any dependencies.

You can now run PHPUnit using:

```
$ vendor/bin/phpunit
```

[161] https://phpunit.de
[162] https://phpunit.de/#download
[163] https://getcomposer.org

Using the PHAR File

After you have downloaded the **phpunit.phar** file, you can use it to run your tests:

```
php phpunit.phar
```

Tip: As a convenience you can make phpunit.phar available globally on Unix or Linux with the following:

```
chmod +x phpunit.phar
sudo mv phpunit.phar /usr/local/bin/phpunit
phpunit --version
```

Please refer to the PHPUnit documentation for instructions regarding Globally installing the PHPUnit PHAR on Windows[164].

Test Database Setup

Remember to have debug enabled in your **config/app_local.php** file before running any tests. Before running any tests you should be sure to add a `test` datasource configuration to **config/app_local.php**. This configuration is used by CakePHP for fixture tables and data:

```
'Datasources' => [
    'test' => [
        'datasource' => 'Cake\Database\Driver\Mysql',
        'persistent' => false,
        'host' => 'dbhost',
        'username' => 'dblogin',
        'password' => 'dbpassword',
        'database' => 'test_database',
    ],
],
```

Note: It's a good idea to make the test database and your actual database different databases. This will prevent embarrassing mistakes later.

Checking the Test Setup

After installing PHPUnit and setting up your `test` datasource configuration you can make sure you're ready to write and run your own tests by running your application's tests:

```
# For phpunit.phar
$ php phpunit.phar

# For Composer installed phpunit
$ vendor/bin/phpunit
```

[164] https://phpunit.de/manual/current/en/installation.html#installation.phar.windows

The above should run any tests you have, or let you know that no tests were run. To run a specific test you can supply the path to the test as a parameter to PHPUnit. For example, if you had a test case for ArticlesTable class you could run it with:

```
$ vendor/bin/phpunit tests/TestCase/Model/Table/ArticlesTableTest
```

You should see a green bar with some additional information about the tests run, and number passed.

Note: If you are on a Windows system you probably won't see any colors.

Test Case Conventions

Like most things in CakePHP, test cases have some conventions. Concerning tests:

1. PHP files containing tests should be in your `tests/TestCase/[Type]` directories.
2. The filenames of these files should end in **Test.php** instead of just .php.
3. The classes containing tests should extend `Cake\TestSuite\TestCase`, `Cake\TestSuite\IntegrationTestCase` or `\PHPUnit\Framework\TestCase`.
4. Like other classnames, the test case classnames should match the filename. **RouterTest.php** should contain `class RouterTest extends TestCase`.
5. The name of any method containing a test (i.e. containing an assertion) should begin with `test`, as in `testPublished()`. You can also use the `@test` annotation to mark methods as test methods.

Creating Your First Test Case

In the following example, we'll create a test case for a very simple helper method. The helper we're going to test will be formatting progress bar HTML. Our helper looks like:

```
namespace App\View\Helper;

use Cake\View\Helper;

class ProgressHelper extends Helper
{
    public function bar($value)
    {
        $width = round($value / 100, 2) * 100;
        return sprintf(
            '<div class="progress-container">
                <div class="progress-bar" style="width: %s%%"></div>
            </div>', $width);
    }
}
```

This is a very simple example, but it will be useful to show how you can create a simple test case. After creating and saving our helper, we'll create the test case file in **tests/TestCase/View/Helper/ProgressHelperTest.php**. In that file we'll start with the following:

```
namespace App\Test\TestCase\View\Helper;

use App\View\Helper\ProgressHelper;
use Cake\TestSuite\TestCase;
use Cake\View\View;

class ProgressHelperTest extends TestCase
{
    public function setUp(): void
    {
    }

    public function testBar(): void
    {
    }
}
```

We'll flesh out this skeleton in a minute. We've added two methods to start with. First is `setUp()`. This method is called before every *test* method in a test case class. Setup methods should initialize the objects needed for the test, and do any configuration needed. In our setup method we'll add the following:

```
public function setUp(): void
{
    parent::setUp();
    $View = new View();
    $this->Progress = new ProgressHelper($View);
}
```

Calling the parent method is important in test cases, as `TestCase::setUp()` does a number things like backing up the values in *Configure* and, storing the paths in *App*.

Next, we'll fill out the test method. We'll use some assertions to ensure that our code creates the output we expect:

```
public function testBar(): void
{
    $result = $this->Progress->bar(90);
    $this->assertStringContainsString('width: 90%', $result);
    $this->assertStringContainsString('progress-bar', $result);

    $result = $this->Progress->bar(33.3333333);
    $this->assertStringContainsString('width: 33%', $result);
}
```

The above test is a simple one but shows the potential benefit of using test cases. We use `assertStringContainsString()` to ensure that our helper is returning a string that contains the content we expect. If the result did not contain the expected content the test would fail, and we would know that our code is incorrect.

By using test cases you can describe the relationship between a set of known inputs and their expected output. This helps you be more confident of the code you're writing as you can ensure that the code you wrote fulfills the expectations and assertions your tests make. Additionally because tests are code, they can be re-run whenever you make a change. This helps prevent the creation of new bugs.

Note: EventManager is refreshed for each test method. This means that when running multiple tests at once, you will

lose your event listeners that were registered in config/bootstrap.php as the bootstrap is only executed once.

Running Tests

Once you have PHPUnit installed and some test cases written, you'll want to run the test cases very frequently. It's a good idea to run tests before committing any changes to help ensure you haven't broken anything.

By using `phpunit` you can run your application tests. To run your application's tests you can simply run:

```
vendor/bin/phpunit

php phpunit.phar
```

If you have cloned the CakePHP source from GitHub[165] and wish to run CakePHP's unit-tests don't forget to execute the following `Composer` command prior to running `phpunit` so that any dependencies are installed:

```
composer install
```

From your application's root directory. To run tests for a plugin that is part of your application source, first `cd` into the plugin directory, then use `phpunit` command that matches how you installed phpunit:

```
cd plugins

../vendor/bin/phpunit

php ../phpunit.phar
```

To run tests on a standalone plugin, you should first install the project in a separate directory and install its dependencies:

```
git clone git://github.com/cakephp/debug_kit.git
cd debug_kit
php ~/composer.phar install
php ~/phpunit.phar
```

Filtering Test Cases

When you have larger test cases, you will often want to run a subset of the test methods when you are trying to work on a single failing case. With the CLI runner you can use an option to filter test methods:

```
$ phpunit --filter testSave tests/TestCase/Model/Table/ArticlesTableTest
```

The filter parameter is used as a case-sensitive regular expression for filtering which test methods to run.

[165] https://github.com/cakephp/cakephp

Generating Code Coverage

You can generate code coverage reports from the command line using PHPUnit's built-in code coverage tools. PHPUnit will generate a set of static HTML files containing the coverage results. You can generate coverage for a test case by doing the following:

```
$ phpunit --coverage-html webroot/coverage tests/TestCase/Model/Table/ArticlesTableTest
```

This will put the coverage results in your application's webroot directory. You should be able to view the results by going to `http://localhost/your_app/coverage`.

You can also use `phpdbg` to generate coverage instead of xdebug. `phpdbg` is generally faster at generating coverage:

```
$ phpdbg -qrr phpunit --coverage-html webroot/coverage tests/TestCase/Model/Table/
↪ArticlesTableTest
```

Combining Test Suites for Plugins

Often times your application will be composed of several plugins. In these situations it can be pretty tedious to run tests for each plugin. You can make running tests for each of the plugins that compose your application by adding additional `<testsuite>` sections to your application's **phpunit.xml.dist** file:

```
<testsuites>
    <testsuite name="app">
        <directory>tests/TestCase/</directory>
    </testsuite>

    <!-- Add your plugin suites -->
    <testsuite name="forum">
        <directory>plugins/Forum/tests/TestCase/</directory>
    </testsuite>
</testsuites>
```

Any additional test suites added to the `<testsuites>` element will automatically be run when you use `phpunit`.

If you are using `<testsuites>` to use fixtures from plugins that you have installed with composer, the plugin's `composer.json` file should add the fixture namespace to the autoload section. Example:

```
"autoload-dev": {
    "psr-4": {
        "PluginName\\Test\\Fixture\\": "tests/Fixture/"
    }
},
```

Test Case Lifecycle Callbacks

Test cases have a number of lifecycle callbacks you can use when doing testing:

- `setUp` is called before every test method. Should be used to create the objects that are going to be tested, and initialize any data for the test. Always remember to call `parent::setUp()`
- `tearDown` is called after every test method. Should be used to cleanup after the test is complete. Always remember to call `parent::tearDown()`.
- `setupBeforeClass` is called once before test methods in a case are started. This method must be *static*.
- `tearDownAfterClass` is called once after test methods in a case are started. This method must be *static*.

Fixtures

When testing code that depends on models and the database, one can use **fixtures** as a way to create initial state for your application's tests. By using fixture data you can reduce repetitive setup steps in your tests. Fixtures are well suited to data that is common or shared amongst many or all of your tests. Data that is only needed in a subset of tests should be created in tests as needed.

CakePHP uses the connection named `test` in your **config/app.php** configuration file. If this connection is not usable, an exception will be raised and you will not be able to use database fixtures.

CakePHP performs the following during the course of a test run:

1. Creates tables for each of the fixtures needed.
2. Populates tables with data.
3. Runs test methods.
4. Empties the fixture tables.

The schema for fixtures is created at the beginning of a test run via migrations or a SQL dump file.

Test Connections

By default CakePHP will alias each connection in your application. Each connection defined in your application's bootstrap that does not start with `test_` will have a `test_` prefixed alias created. Aliasing connections ensures, you don't accidentally use the wrong connection in test cases. Connection aliasing is transparent to the rest of your application. For example if you use the 'default' connection, instead you will get the `test` connection in test cases. If you use the 'replica' connection, the test suite will attempt to use 'test_replica'.

PHPUnit Configuration

Before you can use fixtures you should double check that your `phpunit.xml` contains the fixture extension:

```
<!-- in phpunit.xml -->
<!-- Setup the extension for fixtures -->
<extensions>
    <bootstrap class="Cake\TestSuite\Fixture\Extension\PHPUnitExtension"/>
</extensions>
```

The extension is included in your application and plugins generated by `bake` by default.

Creating Schema in Tests

You can generate test database schema either via CakePHP's migrations, loading a SQL dump file or using another external schema management tool. You should create your schema in your application's `tests/bootstrap.php` file.

Creating Schema with Migrations

If you use CakePHP's *migrations plugin* to manage your application's schema, you can reuse those migrations to generate your test database schema as well:

```
// in tests/bootstrap.php
use Migrations\TestSuite\Migrator;

$migrator = new Migrator();

// Simple setup for with no plugins
$migrator->run();

// Run migrations for a plugin
$migrator->run(['plugin' => 'Contacts']);

// Run the Documents migrations on the test_docs connection.
$migrator->run(['plugin' => 'Documents', 'connection' => 'test_docs']);
```

If you need to run multiple sets of migrations, those can be run as follows:

```
$migrator->runMany([
    // Run app migrations on test connection.
    ['connection' => 'test'],
    // Run Contacts migrations on test connection.
    ['plugin' => 'Contacts'],
    // Run Documents migrations on test_docs connection.
    ['plugin' => 'Documents', 'connection' => 'test_docs']
]);
```

Using `runMany()` will ensure that plugins that share a database don't drop tables as each set of migrations is run.

The migrations plugin will only run unapplied migrations, and will reset migrations if your current migration head differs from the applied migrations.

You can also configure how migrations should be run in tests in your datasources configuration. See the *migrations docs* for more information.

Creating Schema with Abstract Schema

For plugins that need to define schema in tests, but don't need or want to have dependencies on migrations, you can define schema as a structured array of tables. This format is not recommended for application development as it can be time-consuming to maintain.

Each table can define `columns`, `constraints`, and `indexes`. An example table would be:

```
return [
  'articles' => [
```

(continues on next page)

(continued from previous page)

```
        'columns' => [
            'id' => [
                'type' => 'integer',
            ],
            'author_id' => [
                'type' => 'integer',
                'null' => true,
            ],
            'title' => [
                'type' => 'string',
                'null' => true,
            ],
            'body' => 'text',
            'published' => [
                'type' => 'string',
                'length' => 1,
                'default' => 'N',
            ],
        ],
        'constraints' => [
            'primary' => [
                'type' => 'primary',
                'columns' => [
                    'id',
                ],
            ],
        ],
    ],
    // More tables.
];
```

The options available to `columns`, `indexes` and `constraints` match the attributes that are available in CakePHP's schema reflection APIs. Tables are created incrementally and you must take care to ensure that tables are created before foreign key references are made. Once you have created your schema file you can load it in your `tests/bootstrap.php` with:

```
$loader = new SchemaLoader();
$loader->loadInternalFile($pathToSchemaFile);
```

Creating Schema with SQL Dump Files

To load a SQL dump file you can use the following:

```
// in tests/bootstrap.php
use Cake\TestSuite\Fixture\SchemaLoader;

// Load one or more SQL files.
(new SchemaLoader())->loadSqlFiles('path/to/schema.sql', 'test');
```

At the beginning of each test run `SchemaLoader` will drop all tables in the connection and rebuild tables based on the provided schema file.

Fixture State Managers

By default CakePHP resets fixture state at the end of each test by truncating all the tables in the database. This operation can become expensive as your application grows. By using `TransactionStrategy` each test method will be run inside a transaction that is rolled back at the end of the test. This can yield improved performance but requires your tests not heavily rely on static fixture data, as auto-increment values are not reset before each test.

The fixture state management strategy can be defined within the test case:

```
use Cake\TestSuite\TestCase;
use Cake\TestSuite\Fixture\FixtureStrategyInterface;
use Cake\TestSuite\Fixture\TransactionStrategy;

class ArticlesTableTest extends TestCase
{
    /**
     * Create the fixtures strategy used for this test case.
     * You can use a base class/trait to change multiple classes.
     */
    protected function getFixtureStrategy(): FixtureStrategyInterface
    {
        return new TransactionStrategy();
    }
}
```

Creating Fixtures

Fixtures defines the records that will be inserted into the test database at the beginning of each test. Let's create our first fixture, that will be used to test our own Article model. Create a file named **ArticlesFixture.php** in your **tests/Fixture** directory, with the following content:

```
namespace App\Test\Fixture;

use Cake\TestSuite\Fixture\TestFixture;

class ArticlesFixture extends TestFixture
{
      // Optional. Set this property to load fixtures to a different test datasource
      public $connection = 'test';

      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',
```

(continues on next page)

(continued from previous page)

```
                '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'
            ]
        ];
}
```

Note: It is recommended to not manually add values to auto incremental columns, as it interferes with the sequence generation in PostgreSQL and SQLServer.

The `$connection` property defines the datasource of which the fixture will use. If your application uses multiple datasources, you should make the fixtures match the model's datasources but prefixed with `test_`. For example if your model uses the `mydb` datasource, your fixture should use the `test_mydb` datasource. If the `test_mydb` connection doesn't exist, your models will use the default `test` datasource. Fixture datasources must be prefixed with `test` to reduce the possibility of accidentally truncating all your application's data when running tests.

We can define a set of records that will be populated after the fixture table is created. The format is fairly straight forward, `$records` is an array of records. Each item in `$records` should be a single row. Inside each row, should be an associative array of the columns and values for the row. Just keep in mind that each record in the `$records` array must have the same keys as rows are bulk inserted.

Dynamic Data

To use functions or other dynamic data in your fixture records you can define your records in the fixture's `init()` method:

```
namespace App\Test\Fixture;

use Cake\TestSuite\Fixture\TestFixture;

class ArticlesFixture extends TestFixture
{
    public function init(): void
    {
        $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();
    }
}
```

Note: When overriding `init()` remember to always call `parent::init()`.

Loading Fixtures in your Test Cases

After you've created your fixtures, you'll want to use them in your test cases. In each test case you should load the fixtures you will need. You should load a fixture for every model that will have a query run against it. To load fixtures you define the `$fixtures` property in your model:

```
class ArticlesTest extends TestCase
{
    protected $fixtures = ['app.Articles', 'app.Comments'];
}
```

As of 4.1.0 you can use `getFixtures()` to define your fixture list with a method:

```
public function getFixtures(): array
{
    return [
        'app.Articles',
        'app.Comments',
    ];
}
```

The above will load the Article and Comment fixtures from the application's Fixture directory. You can also load fixtures from CakePHP core, or plugins:

```
class ArticlesTest extends TestCase
{
    protected $fixtures = [
        'plugin.DebugKit.Articles',
        'plugin.MyVendorName/MyPlugin.Messages',
        'core.Comments',
    ];
}
```

Using the `core` prefix will load fixtures from CakePHP, and using a plugin name as the prefix, will load the fixture from the named plugin.

You can load fixtures in subdirectories. Using multiple directories can make it easier to organize your fixtures if you have a larger application. To load fixtures in subdirectories, simply include the subdirectory name in the fixture name:

```
class ArticlesTest extends CakeTestCase
{
    protected $fixtures = ['app.Blog/Articles', 'app.Blog/Comments'];
}
```

In the above example, both fixtures would be loaded from `tests/Fixture/Blog/`.

Fixture Factories

As your application grows, so does the number and the size of your test fixtures. You might find it difficult to maintain them and to keep track of their content. The fixture factories plugin[166] proposes an alternative for large sized applications.

The plugin uses the test suite light plugin[167] in order to truncate all dirty tables before each test.

The following command will help you bake your factories:

```
bin/cake bake fixture_factory -h
```

Once your factories are tuned[168], you are ready to create test fixtures in no time.

Unnecessary interaction with the database will slow down your tests as well as your application. You can create test fixtures without persisting them which can be useful for testing methods without DB interaction:

```
$article = ArticleFactory::make()->getEntity();
```

In order to persist:

```
$article = ArticleFactory::make()->persist();
```

The factories help creating associated fixtures too. Assuming that articles belongs to many authors, we can now, for example, create 5 articles each with 2 authors:

```
$articles = ArticleFactory::make(5)->with('Authors', 2)->getEntities();
```

Note that the fixture factories do not require any fixture creation or declaration. Still, they are fully compatible with the fixtures that come with cakephp. You will find additional insights and documentation here[169].

Loading Routes in Tests

If you are testing mailers, controller components or other classes that require routes and resolving URLs, you will need to load routes. During the `setUp()` of a class or during individual test methods you can use `loadRoutes()` to ensure your application routes are loaded:

```
public function setUp(): void
{
    parent::setUp();
    $this->loadRoutes();
}
```

This method will build an instance of your `Application` and call the `routes()` method on it. If your `Application` class requires specialized constructor parameters you can provide those to `loadRoutes($constructorArgs)`.

[166] https://github.com/vierge-noire/cakephp-fixture-factories
[167] https://github.com/vierge-noire/cakephp-test-suite-light
[168] https://github.com/vierge-noire/cakephp-fixture-factories/blob/main/docs/factories.md
[169] https://github.com/vierge-noire/cakephp-fixture-factories

Creating Routes in Tests

Sometimes it may be be necessary to dynamically add routes in tests, for example when developing plugins, or applications that are extensible.

Just like loading existing application routes, this can be done during `setup()` of a test method, and/or in the individual test methods themselves:

```
use Cake\Routing\Route\DashedRoute;
use Cake\Routing\RouteBuilder;
use Cake\Routing\Router;
use Cake\TestSuite\TestCase;

class PluginHelperTest extends TestCase
{
    protected RouteBuilder $routeBuilder;

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

        $this->routeBuilder = Router::createRouteBuilder('/');
        $this->routeBuilder->scope('/', function (RouteBuilder $routes) {
            $routes->setRouteClass(DashedRoute::class);
            $routes->get(
                '/test/view/{id}',
                ['controller' => 'Tests', 'action' => 'view']
            );
            // ...
        });

        // ...
    }
}
```

This will create a new route builder instance that will merge connected routes into the same route collection used by all other route builder instances that may already exist, or are yet to be created in the environment.

Loading Plugins in Tests

If your application would dynamically load plugins, you can use `loadPlugins()` to load one or more plugins during tests:

```
public function testMethodUsingPluginResources()
{
    $this->loadPlugins(['Company/Cms']);
    // Test logic that requires Company/Cms to be loaded.
}
```

Testing Table Classes

Let's say we already have our Articles Table class defined in **src/Model/Table/ArticlesTable.php**, and it looks like:

```
namespace App\Model\Table;

use Cake\ORM\Table;
use Cake\ORM\Query\SelectQuery;

class ArticlesTable extends Table
{
    public function findPublished(SelectQuery $query): SelectQuery
    {
        $query->where([
            $this->getAlias() . '.published' => 1
        ]);

        return $query;
    }
}
```

We now want to set up a test that will test this table class. Let's now create a file named **ArticlesTableTest.php** in your **tests/TestCase/Model/Table** directory, with the following contents:

```
namespace App\Test\TestCase\Model\Table;

use App\Model\Table\ArticlesTable;
use Cake\TestSuite\TestCase;

class ArticlesTableTest extends TestCase
{
    protected $fixtures = ['app.Articles'];
}
```

In our test cases' variable `$fixtures` we define the set of fixtures that we'll use. You should remember to include all the fixtures that will have queries run against them.

Creating a Test Method

Let's now add a method to test the function `published()` in the Articles table. Edit the file **tests/TestCase/Model/Table/ArticlesTableTest.php** so it now looks like this:

```
namespace App\Test\TestCase\Model\Table;

use App\Model\Table\ArticlesTable;
use Cake\TestSuite\TestCase;

class ArticlesTableTest extends TestCase
{
    protected $fixtures = ['app.Articles'];

    public function setUp(): void
```

(continues on next page)

(continued from previous page)

```
    {
        parent::setUp();
        $this->Articles = $this->getTableLocator()->get('Articles');
    }

    public function testFindPublished(): void
    {
        $query = $this->Articles->find('published')->select(['id', 'title']);
        $this->assertInstanceOf('Cake\ORM\Query\SelectQuery', $query);
        $result = $query->enableHydration(false)->toArray();
        $expected = [
            ['id' => 1, 'title' => 'First Article'],
            ['id' => 2, 'title' => 'Second Article'],
            ['id' => 3, 'title' => 'Third Article']
        ];

        $this->assertEquals($expected, $result);
    }
}
```

You can see we have added a method called `testFindPublished()`. We start by creating an instance of our `ArticlesTable` class, and then run our `find('published')` method. In `$expected` we set what we expect should be the proper result (that we know since we have defined which records are initially populated to the article table.) We test that the result equals our expectation by using the `assertEquals()` method. See the *Running Tests* section for more information on how to run your test case.

Using the fixture factories, the test would now look like this:

```
namespace App\Test\TestCase\Model\Table;

use App\Test\Factory\ArticleFactory;
use Cake\TestSuite\TestCase;

class ArticlesTableTest extends TestCase
{
    public function testFindPublished(): void
    {
        // Persist 3 published articles
        $articles = ArticleFactory::make(['published' => 1], 3)->persist();
        // Persist 2 unpublished articles
        ArticleFactory::make(['published' => 0], 2)->persist();

        $result = ArticleFactory::find('published')->find('list')->toArray();

        $expected = [
            $articles[0]->id => $articles[0]->title,
            $articles[1]->id => $articles[1]->title,
            $articles[2]->id => $articles[2]->title,
        ];

        $this->assertEquals($expected, $result);
    }
}
```

No fixtures need to be loaded. The 5 articles created will exist only in this test. The static method `::find()` will query the database without using the table `ArticlesTable` and it's events.

Mocking Model Methods

There will be times you'll want to mock methods on models when testing them. You should use `getMockForModel` to create testing mocks of table classes. It avoids issues with reflected properties that normal mocks have:

```
public function testSendingEmails(): void
{
    $model = $this->getMockForModel('EmailVerification', ['send']);
    $model->expects($this->once())
        ->method('send')
        ->will($this->returnValue(true));

    $model->verifyEmail('test@example.com');
}
```

In your `tearDown()` method be sure to remove the mock with:

```
$this->getTableLocator()->clear();
```

Controller Integration Testing

While you can test controller classes in a similar fashion to Helpers, Models, and Components, CakePHP offers a specialized `IntegrationTestTrait` trait. Using this trait in your controller test cases allows you to test controllers from a high level.

If you are unfamiliar with integration testing, it is a testing approach that allows you to test multiple units in concert. The integration testing features in CakePHP simulate an HTTP request being handled by your application. For example, testing your controller will also exercise any components, models and helpers that would be involved in handling a given request. This gives you a more high level test of your application and all its working parts.

Say you have a typical ArticlesController, and its corresponding model. The controller code looks like:

```
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)) {
                // Redirect as per PRG pattern
                return $this->redirect(['action' => 'index']);
            }
        }
```

(continues on next page)

(continued from previous page)

```
        if (!empty($short)) {
            $result = $this->Articles->find('all', fields: ['id', 'title'])->all();
        } else {
            $result = $this->Articles->find()->all();
        }

        $this->set([
            'title' => 'Articles',
            'articles' => $result
        ]);
    }
}
```

Create a file named **ArticlesControllerTest.php** in your **tests/TestCase/Controller** directory and put the following inside:

```
namespace App\Test\TestCase\Controller;

use Cake\TestSuite\IntegrationTestTrait;
use Cake\TestSuite\TestCase;

class ArticlesControllerTest extends TestCase
{
    use IntegrationTestTrait;

    protected $fixtures = ['app.Articles'];

    public function testIndex(): void
    {
        $this->get('/articles');

        $this->assertResponseOk();
        // More asserts.
    }

    public function testIndexQueryData(): void
    {
        $this->get('/articles?page=1');

        $this->assertResponseOk();
        // More asserts.
    }

    public function testIndexShort(): void
    {
        $this->get('/articles/index/short');

        $this->assertResponseOk();
        $this->assertResponseContains('Articles');
        // More asserts.
    }
```

(continues on next page)

(continued from previous page)

```
    public function testIndexPostData(): void
    {
        $data = [
            'user_id' => 1,
            'published' => 1,
            'slug' => 'new-article',
            'title' => 'New Article',
            'body' => 'New Body'
        ];
        $this->post('/articles', $data);

        $this->assertResponseSuccess();
        $articles = $this->getTableLocator()->get('Articles');
        $query = $articles->find()->where(['title' => $data['title']]);
        $this->assertEquals(1, $query->count());
    }
}
```

This example shows a few of the request sending methods and a few of the assertions that `IntegrationTestTrait` provides. Before you can do any assertions you'll need to dispatch a request. You can use one of the following methods to send a request:

- `get()` Sends a GET request.
- `post()` Sends a POST request.
- `put()` Sends a PUT request.
- `delete()` Sends a DELETE request.
- `patch()` Sends a PATCH request.
- `options()` Sends an OPTIONS request.
- `head()` Sends a HEAD request.

All of the methods except `get()` and `delete()` accept a second parameter that allows you to send a request body. After dispatching a request you can use the various assertions provided by `IntegrationTestTrait` or PHPUnit to ensure your request had the correct side-effects.

Setting up the Request

The `IntegrationTestTrait` trait comes with a number of helpers to to configure the requests you will send to your application under test:

```
// Set cookies
$this->cookie('name', 'Uncle Bob');

// Set session data
$this->session(['Auth.User.id' => 1]);

// Configure headers
$this->configRequest([
    'headers' => ['Accept' => 'application/json']
]);
```

The state set by these helper methods is reset in the `tearDown()` method.

Testing Actions Protected by CsrfComponent or SecurityComponent

When testing actions protected by either SecurityComponent or CsrfComponent you can enable automatic token generation to ensure your tests won't fail due to token mismatches:

```
public function testAdd(): void
{
    $this->enableCsrfToken();
    $this->enableSecurityToken();
    $this->post('/posts/add', ['title' => 'Exciting news!']);
}
```

It is also important to enable debug in tests that use tokens to prevent the SecurityComponent from thinking the debug token is being used in a non-debug environment. When testing with other methods like `requireSecure()` you can use `configRequest()` to set the correct environment variables:

```
// Fake out SSL connections.
$this->configRequest([
    'environment' => ['HTTPS' => 'on']
]);
```

If your action requires unlocked fields you can declare them with `setUnlockedFields()`:

```
$this->setUnlockedFields(['dynamic_field']);
```

Integration Testing PSR-7 Middleware

Integration testing can also be used to test your entire PSR-7 application and /controllers/middleware. By default `IntegrationTestTrait` will auto-detect the presence of an `App\Application` class and automatically enable integration testing of your Application.

You can customize the application class name used, and the constructor arguments, by using the `configApplication()` method:

```
public function setUp(): void
{
    $this->configApplication('App\App', [CONFIG]);
}
```

You should also take care to try and use application-bootstrap to load any plugins containing events/routes. Doing so will ensure that your events/routes are connected for each test case. Alternatively if you wish to load plugins manually in a test you can use the `loadPlugins()` method.

Testing with Encrypted Cookies

If you use the encrypted-cookie-middleware in your application, there are helper methods for setting encrypted cookies in your test cases:

```
// Set a cookie using AES and the default key.
$this->cookieEncrypted('my_cookie', 'Some secret values');

// Assume this action modifies the cookie.
$this->get('/articles/index');

$this->assertCookieEncrypted('An updated value', 'my_cookie');
```

Testing Flash Messages

If you want to assert the presence of flash messages in the session and not the rendered HTML, you can use `enableRetainFlashMessages()` in your tests to retain flash messages in the session so you can write assertions:

```
// Enable retention of flash messages instead of consuming them.
$this->enableRetainFlashMessages();
$this->get('/articles/delete/9999');

$this->assertSession('That article does not exist', 'Flash.flash.0.message');

// Assert a flash message in the 'flash' key.
$this->assertFlashMessage('Article deleted', 'flash');

// Assert the second flash message, also  in the 'flash' key.
$this->assertFlashMessageAt(1, 'Article really deleted');

// Assert a flash message in the 'auth' key at the first position
$this->assertFlashMessageAt(0, 'You are not allowed to enter this dungeon!', 'auth');

// Assert a flash messages uses the error element
$this->assertFlashElement('Flash/error');

// Assert the second flash message element
$this->assertFlashElementAt(1, 'Flash/error');
```

Testing a JSON Responding Controller

JSON is a friendly and common format to use when building a web service. Testing the endpoints of your web service is very simple with CakePHP. Let us begin with a simple example controller that responds in JSON:

```
use Cake\View\JsonView;

class MarkersController extends AppController
{
    public function viewClasses(): array
    {
        return [JsonView::class];
```

(continues on next page)

(continued from previous page)

```
    }

    public function view($id)
    {
        $marker = $this->Markers->get($id);
        $this->set('marker', $marker);
        $this->viewBuilder()->setOption('serialize', ['marker']);
    }
}
```

Now we create the file **tests/TestCase/Controller/MarkersControllerTest.php** and make sure our web service is returning the proper response:

```
class MarkersControllerTest extends IntegrationTestCase
{
    use IntegrationTestTrait;

    public function testGet(): void
    {
        $this->configRequest([
            'headers' => ['Accept' => 'application/json']
        ]);
        $this->get('/markers/view/1.json');

        // Check that the response was a 200
        $this->assertResponseOk();

        $expected = [
            ['id' => 1, 'lng' => 66, 'lat' => 45],
        ];
        $expected = json_encode($expected, JSON_PRETTY_PRINT);
        $this->assertEquals($expected, (string)$this->_response->getBody());
    }
}
```

We use the `JSON_PRETTY_PRINT` option as CakePHP's built in JsonView will use that option when `debug` is enabled.

Testing with file uploads

Simulating file uploads is straightforward when you use the default "*uploaded files as objects*" mode. You can simply create instances that implement \Psr\Http\Message\UploadedFileInterface[170] (the default implementation currently used by CakePHP is `\Laminas\Diactoros\UploadedFile`), and pass them in your test request data. In the CLI environment such objects will by default pass validation checks that test whether the file was uploaded via HTTP. The same is not true for array style data as found in `$_FILES`, it would fail that check.

In order to simulate exactly how the uploaded file objects would be present on a regular request, you not only need to pass them in the request data, but you also need to pass them to the test request configuration via the `files` option. It's not technically necessary though unless your code accesses uploaded files via the *`Cake\Http\ServerRequest::getUploadedFile()`* or *`Cake\Http\ServerRequest::getUploadedFiles()`* methods.

Let's assume articles have a teaser image, and a `Articles hasMany Attachments` association, the form would look like something like this accordingly, where one image file, and multiple attachments/files would be accepted:

[170] https://www.php-fig.org/psr/psr-7/#16-uploaded-files

```
<?= $this->Form->create($article, ['type' => 'file']) ?>
<?= $this->Form->control('title') ?>
<?= $this->Form->control('teaser_image', ['type' => 'file']) ?>
<?= $this->Form->control('attachments.0.attachment', ['type' => 'file']) ?>
<?= $this->Form->control('attachments.0.description']) ?>
<?= $this->Form->control('attachments.1.attachment', ['type' => 'file']) ?>
<?= $this->Form->control('attachments.1.description']) ?>
<?= $this->Form->button('Submit') ?>
<?= $this->Form->end() ?>
```

The test that would simulate the corresponding request could look like this:

```
public function testAddWithUploads(): void
{
    $teaserImage = new \Laminas\Diactoros\UploadedFile(
        '/path/to/test/file.jpg', // stream or path to file representing the temp file
        12345,                    // the filesize in bytes
        \UPLOAD_ERR_OK,           // the upload/error status
        'teaser.jpg',             // the filename as sent by the client
        'image/jpeg'              // the mimetype as sent by the client
    );

    $textAttachment = new \Laminas\Diactoros\UploadedFile(
        '/path/to/test/file.txt',
        12345,
        \UPLOAD_ERR_OK,
        'attachment.txt',
        'text/plain'
    );

    $pdfAttachment = new \Laminas\Diactoros\UploadedFile(
        '/path/to/test/file.pdf',
        12345,
        \UPLOAD_ERR_OK,
        'attachment.pdf',
        'application/pdf'
    );

    // This is the data accessible via `$this->request->getUploadedFile()`
    // and `$this->request->getUploadedFiles()`.
    $this->configRequest([
        'files' => [
            'teaser_image' => $teaserImage,
            'attachments' => [
                0 => [
                    'attachment' => $textAttachment,
                ],
                1 => [
                    'attachment' => $pdfAttachment,
                ],
            ],
        ],
    ]);
```

(continues on next page)

(continued from previous page)

```
    // This is the data accessible via `$this->request->getData()`.
    $postData = [
        'title' => 'New Article',
        'teaser_image' => $teaserImage,
        'attachments' => [
            0 => [
                'attachment' => $textAttachment,
                'description' => 'Text attachment',
            ],
            1 => [
                'attachment' => $pdfAttachment,
                'description' => 'PDF attachment',
            ],
        ],
    ];
    $this->post('/articles/add', $postData);

    $this->assertResponseOk();
    $this->assertFlashMessage('The article was saved successfully');
    $this->assertFileExists('/path/to/uploads/teaser.jpg');
    $this->assertFileExists('/path/to/uploads/attachment.txt');
    $this->assertFileExists('/path/to/uploads/attachment.pdf');
}
```

Tip: If you configure the test request with files, then it *must* match the structure of your POST data (but only include the uploaded file objects)!

Likewise you can simulate upload errors[171] or otherwise invalid files that do not pass validation:

```
public function testAddWithInvalidUploads(): void
{
    $missingTeaserImageUpload = new \Laminas\Diactoros\UploadedFile(
        '',
        0,
        \UPLOAD_ERR_NO_FILE,
        '',
        ''
    );

    $uploadFailureAttachment = new \Laminas\Diactoros\UploadedFile(
        '/path/to/test/file.txt',
        1234567890,
        \UPLOAD_ERR_INI_SIZE,
        'attachment.txt',
        'text/plain'
    );

    $invalidTypeAttachment = new \Laminas\Diactoros\UploadedFile(
```

(continues on next page)

[171] https://www.php.net/manual/en/features.file-upload.errors.php

(continued from previous page)

```
        '/path/to/test/file.exe',
        12345,
        \UPLOAD_ERR_OK,
        'attachment.exe',
        'application/vnd.microsoft.portable-executable'
    );

    $this->configRequest([
        'files' => [
            'teaser_image' => $missingTeaserImageUpload,
            'attachments' => [
                0 => [
                    'file' => $uploadFailureAttachment,
                ],
                1 => [
                    'file' => $invalidTypeAttachment,
                ],
            ],
        ],
    ]);

    $postData = [
        'title' => 'New Article',
        'teaser_image' => $missingTeaserImageUpload,
        'attachments' => [
            0 => [
                'file' => $uploadFailureAttachment,
                'description' => 'Upload failure attachment',
            ],
            1 => [
                'file' => $invalidTypeAttachment,
                'description' => 'Invalid type attachment',
            ],
        ],
    ];
    $this->post('/articles/add', $postData);

    $this->assertResponseOk();
    $this->assertFlashMessage('The article could not be saved');
    $this->assertResponseContains('A teaser image is required');
    $this->assertResponseContains('Max allowed filesize exceeded');
    $this->assertResponseContains('Unsupported file type');
    $this->assertFileNotExists('/path/to/uploads/teaser.jpg');
    $this->assertFileNotExists('/path/to/uploads/attachment.txt');
    $this->assertFileNotExists('/path/to/uploads/attachment.exe');
}
```

Disabling Error Handling Middleware in Tests

When debugging tests that are failing because your application is encountering errors it can be helpful to temporarily disable the error handling middleware to allow the underlying error to bubble up. You can use `disableErrorHandlerMiddleware()` to do this:

```
public function testGetMissing(): void
{
    $this->disableErrorHandlerMiddleware();
    $this->get('/markers/not-there');
    $this->assertResponseCode(404);
}
```

In the above example, the test would fail and the underlying exception message and stack trace would be displayed instead of the rendered error page being checked.

Assertion methods

The `IntegrationTestTrait` trait provides a number of assertion methods that make testing responses much simpler. Some examples are:

```
// Check for a 2xx response code
$this->assertResponseOk();

// Check for a 2xx/3xx response code
$this->assertResponseSuccess();

// Check for a 4xx response code
$this->assertResponseError();

// Check for a 5xx response code
$this->assertResponseFailure();

// Check for a specific response code, for example, 200
$this->assertResponseCode(200);

// Check the Location header
$this->assertRedirect(['controller' => 'Articles', 'action' => 'index']);

// Check that no Location header has been set
$this->assertNoRedirect();

// Check a part of the Location header
$this->assertRedirectContains('/articles/edit/');

// Assert location header does not contain
$this->assertRedirectNotContains('/articles/edit/');

// Assert not empty response content
$this->assertResponseNotEmpty();

// Assert empty response content
$this->assertResponseEmpty();
```

(continues on next page)

(continued from previous page)

```
// Assert response content
$this->assertResponseEquals('Yeah!');

// Assert response content doesn't equal
$this->assertResponseNotEquals('No!');

// Assert partial response content
$this->assertResponseContains('You won!');
$this->assertResponseNotContains('You lost!');

// Assert file sent back
$this->assertFileResponse('/absolute/path/to/file.ext');

// Assert layout
$this->assertLayout('default');

// Assert which template was rendered (if any)
$this->assertTemplate('index');

// Assert data in the session
$this->assertSession(1, 'Auth.User.id');

// Assert response header.
$this->assertHeader('Content-Type', 'application/json');
$this->assertHeaderContains('Content-Type', 'html');

// Assert content-type header doesn't contain xml
$this->assertHeaderNotContains('Content-Type', 'xml');

// Assert view variables
$user =  $this->viewVariable('user');
$this->assertEquals('jose', $user->username);

// Assert cookie values in the response
$this->assertCookie('1', 'thingid');

// Assert a cookie is or is not present
$this->assertCookieIsSet('remember_me');
$this->assertCookieNotSet('remember_me');

// Check the content type
$this->assertContentType('application/json');
```

In addition to the above assertion methods, you can also use all of the assertions in TestSuite[172] and those found in PHPUnit[173].

[172] https://api.cakephp.org/5.x/class-Cake.TestSuite.TestCase.html
[173] https://phpunit.de/manual/current/en/appendixes.assertions.html

Comparing test results to a file

For some types of test, it may be easier to compare the result of a test to the contents of a file - for example, when testing the rendered output of a view. The `StringCompareTrait` adds a simple assert method for this purpose.

Usage involves using the trait, setting the comparison base path and calling `assertSameAsFile`:

```
use Cake\TestSuite\StringCompareTrait;
use Cake\TestSuite\TestCase;

class SomeTest extends TestCase
{
    use StringCompareTrait;

    public function setUp(): void
    {
        $this->_compareBasePath = APP . 'tests' . DS . 'comparisons' . DS;
        parent::setUp();
    }

    public function testExample(): void
    {
        $result = ...;
        $this->assertSameAsFile('example.php', $result);
    }
}
```

The above example will compare `$result` to the contents of the file `APP/tests/comparisons/example.php`.

A mechanism is provided to write/update test files, by setting the environment variable `UPDATE_TEST_COMPARISON_FILES`, which will create and/or update test comparison files as they are referenced:

```
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
```

Console Integration Testing

See *Testing Commands* for how to test console commands.

Mocking Injected Dependencies

See mocking-services-in-tests for how to replace services injected with the dependency injection container in your integration tests.

Mocking HTTP Client Responses

See *Testing* to know how to create mock responses to external APIs.

Testing Views

Generally most applications will not directly test their HTML code. Doing so is often results in fragile, difficult to maintain test suites that are prone to breaking. When writing functional tests using `IntegrationTestTrait` you can inspect the rendered view content by setting the `return` option to 'view'. While it is possible to test view content using `IntegrationTestTrait`, a more robust and maintainable integration/view testing can be accomplished using tools like Selenium webdriver[174].

Testing Components

Let's pretend we have a component called PagematronComponent in our application. This component helps us set the pagination limit value across all the controllers that use it. Here is our example component located in **src/Controller/Component/PagematronComponent.php**:

```
class PagematronComponent extends Component
{
    public $controller = null;

    public function setController($controller)
    {
        $this->controller = $controller;
        // Make sure the controller is using pagination
        if (!isset($this->controller->paginate)) {
            $this->controller->paginate = [];
        }
    }

    public function startup(EventInterface $event)
    {
        $this->setController($event->getSubject());
    }
```

(continues on next page)

[174] https://www.selenium.dev/

(continued from previous page)

```
    public function adjust($length = 'short'): void
    {
        switch ($length) {
            case 'long':
                $this->controller->paginate['limit'] = 100;
            break;
            case 'medium':
                $this->controller->paginate['limit'] = 50;
            break;
            default:
                $this->controller->paginate['limit'] = 20;
            break;
        }
    }
}
```

Now we can write tests to ensure our paginate `limit` parameter is being set correctly by the `adjust()` method in our component. We create the file **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
{
    protected $component;
    protected $controller;

    public function setUp(): void
    {
        parent::setUp();
        // Setup our component and provide it a basic controller.
        // If your component relies on Application features, use AppController.
        $request = new ServerRequest();
        $response = new Response();
        $this->controller = new Controller($request);
        $registry = new ComponentRegistry($this->controller);

        $this->component = new PagematronComponent($registry);
        $event = new Event('Controller.startup', $this->controller);
        $this->component->startup($event);
    }

    public function testAdjust(): void
    {
```

(continues on next page)

(continued from previous page)

```
        // Test our adjust method with different parameter settings
        $this->component->adjust();
        $this->assertEquals(20, $this->controller->paginate['limit']);

        $this->component->adjust('medium');
        $this->assertEquals(50, $this->controller->paginate['limit']);

        $this->component->adjust('long');
        $this->assertEquals(100, $this->controller->paginate['limit']);
    }

    public function tearDown(): void
    {
        parent::tearDown();
        // Clean up after we're done
        unset($this->component, $this->controller);
    }
}
```

Testing Helpers

Since a decent amount of logic resides in Helper classes, it's important to make sure those classes are covered by test cases.

First we create an example helper to test. The `CurrencyRendererHelper` will help us display currencies in our views and for simplicity only has one method `usd()`:

```
// src/View/Helper/CurrencyRendererHelper.php
namespace App\View\Helper;

use Cake\View\Helper;

class CurrencyRendererHelper extends Helper
{
    public function usd($amount): string
    {
        return 'USD ' . number_format($amount, 2, '.', ',');
    }
}
```

Here we set the decimal places to 2, decimal separator to dot, thousands separator to comma, and prefix the formatted number with 'USD' string.

Now we create our tests:

```
// tests/TestCase/View/Helper/CurrencyRendererHelperTest.php

namespace App\Test\TestCase\View\Helper;

use App\View\Helper\CurrencyRendererHelper;
use Cake\TestSuite\TestCase;
```

(continues on next page)

(continued from previous page)

```
use Cake\View\View;

class CurrencyRendererHelperTest extends TestCase
{
    public $helper = null;

    // Here we instantiate our helper
    public function setUp(): void
    {
        parent::setUp();
        $View = new View();
        $this->helper = new CurrencyRendererHelper($View);
    }

    // Testing the usd() function
    public function testUsd(): void
    {
        $this->assertEquals('USD 5.30', $this->helper->usd(5.30));

        // We should always have 2 decimal digits
        $this->assertEquals('USD 1.00', $this->helper->usd(1));
        $this->assertEquals('USD 2.05', $this->helper->usd(2.05));

        // Testing the thousands separator
        $this->assertEquals(
          'USD 12,000.70',
          $this->helper->usd(12000.70)
        );
    }
}
```

Here, we call `usd()` with different parameters and tell the test suite to check if the returned values are equal to what is expected.

Save this and execute the test. You should see a green bar and messaging indicating 1 pass and 4 assertions.

When you are testing a Helper which uses other helpers, be sure to mock the View clases `loadHelpers` method.

Testing Events

The *Events System* is a great way to decouple your application code, but sometimes when testing, you tend to test the results of events in the test cases that execute those events. This is an additional form of coupling that can be removed by using `assertEventFired` and `assertEventFiredWith` instead.

Expanding on the Orders example, say we have the following tables:

```
class OrdersTable extends Table
{
    public function place($order): bool
    {
        if ($this->save($order)) {
            // moved cart removal to CartsTable
```

(continues on next page)

(continued from previous page)

```
            $event = new Event('Model.Order.afterPlace', $this, [
                'order' => $order
            ]);
            $this->getEventManager()->dispatch($event);
            return true;
        }
        return false;
    }
}

class CartsTable extends Table
{
    public function implementedEvents(): array
    {
        return [
            'Model.Order.afterPlace' => 'removeFromCart'
        ];
    }

    public function removeFromCart(EventInterface $event): void
    {
        $order = $event->getData('order');
        $this->delete($order->cart_id);
    }
}
```

Note: To assert that events are fired, you must first enable *Tracking Events* on the event manager you wish to assert against.

To test the `OrdersTable` above, we enable tracking in `setUp()` then assert that the event was fired, and assert that the `$order` entity was passed in the event data:

```
namespace App\Test\TestCase\Model\Table;

use App\Model\Table\OrdersTable;
use Cake\Event\EventList;
use Cake\TestSuite\TestCase;

class OrdersTableTest extends TestCase
{
    protected $fixtures = ['app.Orders'];

    public function setUp(): void
    {
        parent::setUp();
        $this->Orders = $this->getTableLocator()->get('Orders');
        // enable event tracking
        $this->Orders->getEventManager()->setEventList(new EventList());
    }

    public function testPlace(): void
```

(continues on next page)

(continued from previous page)

```
    {
        $order = new Order([
            'user_id' => 1,
            'item' => 'Cake',
            'quantity' => 42,
        ]);

        $this->assertTrue($this->Orders->place($order));

        $this->assertEventFired('Model.Order.afterPlace', $this->Orders->
↪getEventManager());
        $this->assertEventFiredWith('Model.Order.afterPlace', 'order', $order, $this->
↪Orders->getEventManager());
    }
}
```

By default, the global `EventManager` is used for assertions, so testing global events does not require passing the event manager:

```
$this->assertEventFired('My.Global.Event');
$this->assertEventFiredWith('My.Global.Event', 'user', 1);
```

Testing Email

See *Testing Mailers* for information on testing email.

Creating Test Suites

If you want several of your tests to run at the same time, you can create a test suite. A test suite is composed of several test cases. You can either create test suites in your application's **phpunit.xml** file. A simple example would be:

```
<testsuites>
  <testsuite name="Models">
    <directory>src/Model</directory>
    <file>src/Service/UserServiceTest.php</file>
    <exclude>src/Model/Cloud/ImagesTest.php</exclude>
  </testsuite>
</testsuites>
```

Creating Tests for Plugins

Tests for plugins are created in their own directory inside the plugins folder.

```
/src
/plugins
    /Blog
        /tests
            /TestCase
            /Fixture
```

They work just like normal tests but you have to remember to use the naming conventions for plugins when importing classes. This is an example of a testcase for the `BlogPost` model from the plugins chapter of this manual. A difference from other tests is in the first line where 'Blog.BlogPost' is imported. You also need to prefix your plugin fixtures with `plugin.Blog.BlogPosts`:

```
namespace Blog\Test\TestCase\Model\Table;

use Blog\Model\Table\BlogPostsTable;
use Cake\TestSuite\TestCase;

class BlogPostsTableTest extends TestCase
{
    // Plugin fixtures located in /plugins/Blog/tests/Fixture/
    protected $fixtures = ['plugin.Blog.BlogPosts'];

    public function testSomething(): void
    {
        // Test something.
    }
}
```

If you want to use plugin fixtures in the app tests you can reference them using `plugin.pluginName.fixtureName` syntax in the `$fixtures` array. Additionally if you use vendor plugin name or fixture directories you can use the following: `plugin.vendorName/pluginName.folderName/fixtureName`.

Before you can use fixtures you should ensure you have the *fixture listener* configured in your `phpunit.xml` file. You should also ensure that your fixtures are loadable. Ensure the following is present in your **composer.json** file:

```
"autoload-dev": {
    "psr-4": {
        "MyPlugin\\Test\\": "plugins/MyPlugin/tests/"
    }
}
```

Note: Remember to run `composer.phar dumpautoload` when adding new autoload mappings.

Generating Tests with Bake

If you use bake to generate scaffolding, it will also generate test stubs. If you need to re-generate test case skeletons, or if you want to generate test skeletons for code you wrote, you can use `bake`:

```
bin/cake bake test <type> <name>
```

`<type>` should be one of:

1. Entity
2. Table
3. Controller
4. Component
5. Behavior
6. Helper
7. Shell
8. Task
9. ShellHelper
10. Cell
11. Form
12. Mailer
13. Command

While `<name>` should be the name of the object you want to bake a test skeleton for.

CHAPTER 30

Validation

The validation package in CakePHP provides features to build validators that can validate arbitrary arrays of data with ease. You can find a list of available Validation rules in the API[175].

Creating Validators

class Cake\Validation**Validator**

Validator objects define the rules that apply to a set of fields. Validator objects contain a mapping between fields and validation sets. In turn, the validation sets contain a collection of rules that apply to the field they are attached to. Creating a validator is simple:

```
use Cake\Validation\Validator;

$validator = new Validator();
```

Once created, you can start defining sets of rules for the fields you want to validate:

```
$validator
    ->requirePresence('title')
    ->notEmptyString('title', 'Please fill this field')
    ->add('title', [
        'length' => [
            'rule' => ['minLength', 10],
            'message' => 'Titles need to be at least 10 characters long',
        ]
    ])
    ->allowEmptyDateTime('published')
```

(continues on next page)

[175] https://api.cakephp.org/5.x/class-Cake.Validation.Validation.html

(continued from previous page)

```
    ->add('published', 'boolean', [
        'rule' => 'boolean',
    ])
    ->requirePresence('body')
    ->add('body', 'length', [
        'rule' => ['minLength', 50],
        'message' => 'Articles must have a substantial body.',
    ]);
```

As seen in the example above, validators are built with a fluent interface that allows you to define rules for each field you want to validate.

There were a few methods called in the example above, so let's go over the various features. The `add()` method allows you to add new rules to a validator. You can either add rules individually or in groups as seen above.

Requiring Field Presence

The `requirePresence()` method requires the field to be present in any validated array. If the field is absent, validation will fail. The `requirePresence()` method has 4 modes:

- `true` The field's presence is always required.
- `false` The field's presence is not required.
- `create` The field's presence is required when validating a **create** operation.
- `update` The field's presence is required when validating an **update** operation.

By default, `true` is used. Key presence is checked by using `array_key_exists()` so that null values will count as present. You can set the mode using the second parameter:

```
$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.',
    ],
]);
```

Allowing Empty Fields

Validators offer several methods to control which fields accept empty values and which empty values are accepted and not forwarded to other validation rules for the named field. CakePHP provides empty value support for different shapes of data:

1. `allowEmptyString()` Should be used when you want to only accept an empty string.
2. `allowEmptyArray()` Should be used when you want to accept an array.
3. `allowEmptyDate()` Should be used when you want to accept an empty string, or an array that is marshalled into a date field.
4. `allowEmptyTime()` Should be used when you want to accept an empty string, or an array that is marshalled into a time field.
5. `allowEmptyDateTime()` Should be used when you want to accept an empty string or an array that is marshalled into a datetime or timestamp field.
6. `allowEmptyFile()` Should be used when you want to accept an array that contains an empty uploaded file.

You also can use following specific validators: `notEmptyString()`, `notEmptyArray()`, `notEmptyFile()`, `notEmptyDate()`, `notEmptyTime()`, `notEmptyDateTime()`.

The `allowEmpty*` methods support a `when` parameter that allows you to control when a field can or cannot be empty:

- `false` The field is not allowed to be empty.
- `create` The field can be empty when validating a **create** operation.
- `update` The field can be empty when validating an **update** operation.
- A callback that returns `true` or `false` to indicate whether a field is allowed to be empty. See the *Conditional Validation* section for examples on how to use this parameter.

An example of these methods in action is:

```
$validator->allowEmptyDateTime('published')
    ->allowEmptyString('title', 'Title cannot be empty', false)
    ->allowEmptyString('body', 'Body cannot be empty', 'update')
    ->allowEmptyFile('header_image', 'update');
    ->allowEmptyDateTime('posted', 'update');
```

Adding Validation Rules

The `Validator` class provides methods that make building validators simple and expressive. For example adding validation rules to a username could look like:

```
$validator = new Validator();
$validator
    ->email('username')
    ->ascii('username')
    ->lengthBetween('username', [4, 8]);
```

See the Validator API documentation[176] for the full set of validator methods.

[176] https://api.cakephp.org/5.x/class-Cake.Validation.Validator.html

Using Custom Validation Rules

In addition to using methods on the `Validator`, and coming from providers, you can also use any callable, including anonymous functions, as validation rules:

```
// Use a global function
$validator->add('title', 'custom', [
    'rule' => 'validate_title',
    'message' => 'The title is not valid'
]);

// Use an array callable that is not in a provider
$validator->add('title', 'custom', [
    'rule' => [$this, 'method'],
    'message' => 'The title is not valid'
]);

// Use a closure
$extra = 'Some additional value needed inside the closure';
$validator->add('title', 'custom', [
    'rule' => function ($value, $context) use ($extra) {
        // Custom logic that returns true/false
    },
    'message' => 'The title is not valid'
]);

// Use a rule from a custom provider
$validator->add('title', 'custom', [
    'rule' => 'customRule',
    'provider' => 'custom',
    'message' => 'The title is not unique enough'
]);
```

Closures or callable methods will receive 2 arguments when called. The first will be the value for the field being validated. The second is a context array containing data related to the validation process:

- **data**: The original data passed to the validation method, useful if you plan to create rules comparing values.
- **providers**: The complete list of rule provider objects, useful if you need to create complex rules by calling multiple providers.
- **newRecord**: Whether the validation call is for a new record or a preexisting one.

Closures should return boolean true if the validation passes. If it fails, return boolean false or for a custom error message return a string, see the *Conditional/Dynamic Error Messages* section for further details.

Conditional/Dynamic Error Messages

Validation rule methods, being it *custom callables*, or *methods supplied by providers*, can either return a boolean, indicating whether the validation succeeded, or they can return a string, which means that the validation failed, and that the returned string should be used as the error message.

Possible existing error messages defined via the `message` option will be overwritten by the ones returned from the validation rule method:

```
$validator->add('length', 'custom', [
    'rule' => function ($value, $context) {
        if (!$value) {
            return false;
        }

        if ($value < 10) {
            return 'Error message when value is less than 10';
        }

        if ($value > 20) {
            return 'Error message when value is greater than 20';
        }

        return true;
    },
    'message' => 'Generic error message used when `false` is returned'
]);
```

Conditional Validation

When defining validation rules, you can use the `on` key to define when a validation rule should be applied. If left undefined, the rule will always be applied. Other valid values are `create` and `update`. Using one of these values will make the rule apply to only create or update operations.

Additionally, you can provide a callable function that will determine whether or not a particular rule should be applied:

```
$validator->add('picture', 'file', [
    'rule' => ['mimeType', ['image/jpeg', 'image/png']],
    'on' => function ($context) {
        return !empty($context['data']['show_profile_picture']);
    }
]);
```

You can access the other submitted field values using the `$context['data']` array. The above example will make the rule for 'picture' optional depending on whether the value for `show_profile_picture` is empty. You could also use the `uploadedFile` validation rule to create optional file upload inputs:

```
$validator->add('picture', 'file', [
    'rule' => ['uploadedFile', ['optional' => true]],
]);
```

The `allowEmpty*`, `notEmpty*` and `requirePresence()` methods will also accept a callback function as their last argument. If present, the callback determines whether or not the rule should be applied. For example, a field is sometimes allowed to be empty:

```
$validator->allowEmptyString('tax', 'This field is required', function ($context) {
    return !$context['data']['is_taxable'];
});
```

Likewise, a field can be required to be populated when certain conditions are met:

```
$validator->notEmptyString('email_frequency', 'This field is required', function (
↪$context) {
    return !empty($context['data']['wants_newsletter']);
});
```

In the above example, the `email_frequency` field cannot be left empty if the the user wants to receive the newsletter.

Further it's also possible to require a field to be present under certain conditions only:

```
$validator->requirePresence('full_name', function ($context) {
    if (isset($context['data']['action'])) {
        return $context['data']['action'] === 'subscribe';
    }
    return false;
});
$validator->requirePresence('email');
```

This would require the `full_name` field to be present only in case the user wants to create a subscription, while the `email` field would always be required.

The `$context` parameter passed to custom conditional callbacks contains the following keys:

- `data` The data being validated.
- `newRecord` a boolean indicating whether a new or existing record is being validated.
- `field` The current field being validated.
- `providers` The validation providers attached to the current validator.

Marking Rules as the Last to Run

When fields have multiple rules, each validation rule will be run even if the previous one has failed. This allows you to collect as many validation errors as you can in a single pass. If you want to stop execution after a specific rule has failed, you can set the `last` option to `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.',
        ],
    ]);
```

If the minLength rule fails in the example above, the maxLength rule will not be run.

Make Rules 'last' by default

You can have the `last` option automatically applied to each rule you can use the `setStopOnFailure()` method to enable this behavior:

```
public function validationDefault(Validator $validator): Validator
{
    $validator
        ->setStopOnFailure()
        ->requirePresence('email', 'create')
        ->notBlank('email')
        ->email('email');

    return $validator;
}
```

When enabled all fields will stop validation on the first failing rule instead of checking all possible rules. In this case only a single error message will appear under the form field.

Adding Validation Providers

The `Validator`, `ValidationSet` and `ValidationRule` classes do not provide any validation methods themselves. Validation rules come from 'providers'. You can bind any number of providers to a Validator object. Validator instances come with a 'default' provider setup automatically. The default provider is mapped to the `Validation` class. This makes it simple to use the methods on that class as validation rules. When using Validators and the ORM together, additional providers are configured for the table and entity objects. You can use the `setProvider()` method to add any additional providers your application needs:

```
$validator = new Validator();

// Use an object instance.
$validator->setProvider('custom', $myObject);

// Use a class name. Methods must be static.
$validator->setProvider('custom', 'App\Model\Validation');
```

Validation providers can be objects, or class names. If a class name is used the methods must be static. To use a provider other than 'default', be sure to set the `provider` key in your rule:

```
// Use a rule from the table provider
$validator->add('title', 'custom', [
    'rule' => 'customTableMethod',
    'provider' => 'table'
]);
```

If you wish to add a `provider` to all `Validator` objects that are created in the future, you can use the `addDefaultProvider()` method as follows:

```
use Cake\Validation\Validator;

// Use an object instance.
Validator::addDefaultProvider('custom', $myObject);

// Use a class name. Methods must be static.
Validator::addDefaultProvider('custom', 'App\Model\Validation');
```

Note: DefaultProviders must be added before the `Validator` object is created therefore **config/bootstrap.php** is the best place to set up your default providers.

You can use the Localized plugin[177] to get providers based on countries. With this plugin, you'll be able to validate model fields, depending on a country, ie:

```
namespace App\Model\Table;

use Cake\ORM\Table;
use Cake\Validation\Validator;

class PostsTable extends Table
{
    public function validationDefault(Validator $validator): Validator
    {
        // add the provider to the validator
        $validator->setProvider('fr', 'Cake\Localized\Validation\FrValidation');
        // use the provider in a field validation rule
        $validator->add('phoneField', 'myCustomRuleNameForPhone', [
            'rule' => 'phone',
            'provider' => 'fr'
        ]);

        return $validator;
    }
}
```

The localized plugin uses the two letter ISO code of the countries for validation, like en, fr, de.

There are a few methods that are common to all classes, defined through the ValidationInterface interface[178]:

```
phone() to check a phone number
postal() to check a postal code
personId() to check a country specific person ID
```

[177] https://github.com/cakephp/localized
[178] https://github.com/cakephp/localized/blob/master/src/Validation/ValidationInterface.php

Nesting Validators

When validating *Modelless Forms* with nested data, or when working with models that contain array data types, it is necessary to validate the nested data you have. CakePHP makes it simple to add validators to specific attributes. For example, assume you are working with a non-relational database and need to store an article and its comments:

```
$data = [
    'title' => 'Best article',
    'comments' => [
        ['comment' => ''],
    ],
];
```

To validate the comments you would use a nested validator:

```
$validator = new Validator();
$validator->add('title', 'not-blank', ['rule' => 'notBlank']);

$commentValidator = new Validator();
$commentValidator->add('comment', 'not-blank', ['rule' => 'notBlank']);

// Connect the nested validators.
$validator->addNestedMany('comments', $commentValidator);

// Get all errors including those from nested validators.
$validator->validate($data);
```

You can create 1:1 'relationships' with `addNested()` and 1:N 'relationships' with `addNestedMany()`. With both methods, the nested validator's errors will contribute to the parent validator's errors and influence the final result. Like other validator features, nested validators support error messages and conditional application:

```
$validator->addNestedMany(
    'comments',
    $commentValidator,
    'Invalid comment',
    'create'
);
```

The error message for a nested validator can be found in the `_nested` key.

Creating Reusable Validators

While defining validators inline where they are used makes for good example code, it doesn't lead to maintainable applications. Instead, you should create `Validator` sub-classes for your reusable validation logic:

```
// In src/Model/Validation/ContactValidator.php
namespace App\Model\Validation;

use Cake\Validation\Validator;

class ContactValidator extends Validator
{
    public function __construct()
```

(continues on next page)

(continued from previous page)

```
    {
        parent::__construct();
        // Add validation rules here.
    }
}
```

Validating Data

Now that you've created a validator and added the rules you want to it, you can start using it to validate data. Validators are able to validate array data. For example, if you wanted to validate a contact form before creating and sending an email you could do the following:

```
use Cake\Validation\Validator;

$validator = new Validator();
$validator
    ->requirePresence('email')
    ->add('email', 'validFormat', [
        'rule' => 'email',
        'message' => 'E-mail must be valid',
    ])
    ->requirePresence('name')
    ->notEmptyString('name', 'We need your name.')
    ->requirePresence('comment')
    ->notEmptyString('comment', 'You need to give a comment.');

$errors = $validator->validate($this->request->getData());
if (empty($errors)) {
    // Send an email.
}
```

The `getErrors()` method will return a non-empty array when there are validation failures. The returned array of errors will be structured like:

```
$errors = [
    'email' => ['E-mail must be valid'],
];
```

If you have multiple errors on a single field, an array of error messages will be returned per field. By default the `getErrors()` method applies rules for the 'create' mode. If you'd like to apply 'update' rules you can do the following:

```
$errors = $validator->validate($this->request->getData(), false);
if (!$errors) {
    // Send an email.
}
```

Note: If you need to validate entities you should use methods like `newEntity()`, `newEntities()`, `patchEntity()`, `patchEntities()` as they are designed for that.

Validating Entity Data

Validation is meant for checking request data coming from forms or other user interfaces used to populate the entities.

The request data is validated automatically when using the `newEntity()`, `newEntities()`, `patchEntity()` or `patchEntities()` methods of `Table` class:

```
// In the ArticlesController class
$article = $this->Articles->newEntity($this->request->getData());
if ($article->getErrors()) {
    // Do work to show error messages.
}
```

Similarly, when you need to validate multiple entities at a time, you can use the `newEntities()` method:

```
// In the ArticlesController class
$entities = $this->Articles->newEntities($this->request->getData());
foreach ($entities as $entity) {
    if (!$entity->getErrors()) {
        $this->Articles->save($entity);
    }
}
```

The `newEntity()`, `patchEntity()`, `newEntities()` and `patchEntities()` methods allow you to specify which associations are validated, and which validation sets to apply using the `options` parameter:

```
$valid = $this->Articles->newEntity($article, [
    'associated' => [
        'Comments' => [
            'associated' => ['User'],
            'validate' => 'special',
        ],
    ],
]);
```

Apart from validating user provided data maintaining integrity of data regardless where it came from is important. To solve this problem CakePHP offers a second level of validation which is called "application rules". You can read more about them in the *Applying Application Rules* section.

Core Validation Rules

CakePHP provides a basic suite of validation methods in the `Validation` class. The Validation class contains a variety of static methods that provide validators for several common validation situations.

The API documentation[179] for the `Validation` class provides a good list of the validation rules that are available, and their basic usage.

Some of the validation methods accept additional parameters to define boundary conditions or valid options. You can provide these boundary conditions and options as follows:

[179] https://api.cakephp.org/5.x/class-Cake.Validation.Validation.html

```
$validator = new Validator();
$validator
    ->add('title', 'minLength', [
        'rule' => ['minLength', 10],
    ])
    ->add('rating', 'validValue', [
        'rule' => ['range', 1, 5],
    ]);
```

Core rules that take additional parameters should have an array for the `rule` key that contains the rule as the first element, and the additional parameters as the remaining parameters.

CHAPTER 31

App Class

class Cake\Core**App**

The App class is responsible for resource location and path management.

Finding Classes

static Cake\Core\App::**className**(*$name*, *$type = ''*, *$suffix = ''*)

This method is used to resolve class names throughout CakePHP. It resolves the short form names CakePHP uses and returns the fully resolved class name:

```
// Resolve a short class name with the namespace + suffix.
App::className('Flash', 'Controller/Component', 'Component');
// Returns Cake\Controller\Component\FlashComponent

// Resolve a plugin name.
App::className('DebugKit.Toolbar', 'Controller/Component', 'Component');
// Returns DebugKit\Controller\Component\ToolbarComponent

// Names with \ in them will be returned unaltered.
App::className('App\Cache\ComboCache');
// Returns App\Cache\ComboCache
```

When resolving classes, the `App` namespace will be tried, and if the class does not exist the `Cake` namespace will be attempted. If both class names do not exist, `false` will be returned.

Finding Paths to Resources

static Cake\Core\App::**path**(*string $package*, *?string $plugin = null*)

The method returns paths set using `App.paths` app config:

```
// Get the templates path set using ``App.paths.templates`` app config.
App::path('templates');
```

The same way you can retrieve paths for `locales`, `plugins`.

Finding Paths to Namespaces

static Cake\Core\App::**classPath**(*string $package*, *?string $plugin = null*)

Used to get locations for paths based on conventions:

```
// Get the path to Controller/ in your application
App::classPath('Controller');
```

This can be done for all namespaces that are part of your application.

`App::classPath()` will only return the default path, and will not be able to provide any information about additional paths the autoloader is configured for.

static Cake\Core\App::**core**(*string $package*)

Used for finding the path to a package inside CakePHP:

```
// Get the path to Cache engines.
App::core('Cache/Engine');
```

Locating Themes

Since themes are plugins, you can use the methods above to get the path to a theme.

Loading Vendor Files

Ideally vendor files should be autoloaded with `Composer`, if you have vendor files that cannot be autoloaded or installed with Composer you will need to use `require` to load them.

If you cannot install a library with Composer, it is best to install each library in a directory following Composer's convention of `vendor/$author/$package`. If you had a library called AcmeLib, you could install it into `vendor/Acme/AcmeLib`. Assuming it did not use PSR-0 compatible classnames you could autoload the classes within it using `classmap` in your application's `composer.json`:

```
"autoload": {
    "psr-4": {
        "App\\": "src/",
        "App\\Test\\": "tests/"
```

(continues on next page)

(continued from previous page)

```
    },
    "classmap": [
        "vendor/Acme/AcmeLib"
    ]
}
```

If your vendor library does not use classes, and instead provides functions, you can configure Composer to load these files at the beginning of each request using the `files` autoloading strategy:

```
"autoload": {
    "psr-4": {
        "App\\": "src/",
        "App\\Test\\": "tests/"
    },
    "files": [
        "vendor/Acme/AcmeLib/functions.php"
    ]
}
```

After configuring the vendor libraries you will need to regenerate your application's autoloader using:

```
$ php composer.phar dump-autoload
```

If you happen to not be using Composer in your application, you will need to manually load all vendor libraries yourself.

CHAPTER 32

Collections

class Cake\Collection**Collection**

The collection classes provide a set of tools to manipulate arrays or `Traversable` objects. If you have ever used underscore.js, you have an idea of what you can expect from the collection classes.

Collection instances are immutable; modifying a collection will instead generate a new collection. This makes working with collection objects more predictable as operations are side-effect free.

Quick Example

Collections can be created using an array or `Traversable` object. You'll also interact with collections every time you interact with the ORM in CakePHP. A simple use of a Collection would be:

```
use Cake\Collection\Collection;

$items = ['a' => 1, 'b' => 2, 'c' => 3];
$collection = new Collection($items);

// Create a new collection containing elements
// with a value greater than one.
$overOne = $collection->filter(function ($value, $key, $iterator) {
    return $value > 1;
});
```

You can also use the `collection()` helper function instead of `new Collection()`:

```
$items = ['a' => 1, 'b' => 2, 'c' => 3];

// These both make a Collection instance.
```

(continues on next page)

(continued from previous page)

```
$collectionA = new Collection($items);
$collectionB = collection($items);
```

The benefit of the helper method is that it is easier to chain off of than `(new Collection($items))`.

The `CollectionTrait` allows you to integrate collection-like features into any `Traversable` object you have in your application as well.

List of Methods

append	*appendItem*	*avg*
buffered	*chunk*	*chunkWithKeys*
combine	*compile*	*contains*
countBy	*each*	*every*
extract	*filter*	*first*
firstMatch	*groupBy*	*indexBy*
insert	*isEmpty*	*last*
listNested	*map*	*match*
max	*median*	*min*
nest	*prepend*	*prependItem*
reduce	*reject*	*sample*
shuffle	*skip*	*some*
sortBy	*stopWhen*	*sumOf*
take	*through*	*transpose*
unfold	*zip*	

Iterating

`Cake\Collection\Collection::`**`each`**`(`*`$callback`*`)`

Collections can be iterated and/or transformed into new collections with the `each()` and `map()` methods. The `each()` method will not create a new collection, but will allow you to modify any objects within the collection:

```
$collection = new Collection($items);
$collection = $collection->each(function ($value, $key) {
    echo "Element $key: $value";
});
```

The return of `each()` will be the collection object. Each will iterate the collection immediately applying the callback to each value in the collection.

`Cake\Collection\Collection::`**`map`**`(`*`$callback`*`)`

The `map()` method will create a new collection based on the output of the callback being applied to each object in the original collection:

```
$items = ['a' => 1, 'b' => 2, 'c' => 3];
$collection = new Collection($items);
```

(continues on next page)

(continued from previous page)

```
$new = $collection->map(function ($value, $key) {
    return $value * 2;
});

// $result contains [2, 4, 6];
$result = $new->toList();

// $result contains ['a' => 2, 'b' => 4, 'c' => 6];
$result = $new->toArray();
```

The `map()` method will create a new iterator which lazily creates the resulting items when iterated.

`Cake\Collection\Collection::`**`extract`**`(`*$path*`)`

One of the most common uses for a `map()` function is to extract a single column from a collection. If you are looking to build a list of elements containing the values for a particular property, you can use the `extract()` method:

```
$collection = new Collection($people);
$names = $collection->extract('name');

// $result contains ['mark', 'jose', 'barbara'];
$result = $names->toList();
```

As with many other functions in the collection class, you are allowed to specify a dot-separated path for extracting columns. This example will return a collection containing the author names from a list of articles:

```
$collection = new Collection($articles);
$names = $collection->extract('author.name');

// $result contains ['Maria', 'Stacy', 'Larry'];
$result = $names->toList();
```

Finally, if the property you are looking after cannot be expressed as a path, you can use a callback function to return it:

```
$collection = new Collection($articles);
$names = $collection->extract(function ($article) {
    return $article->author->name . ', ' . $article->author->last_name;
});
```

Often, the properties you need to extract a common key present in multiple arrays or objects that are deeply nested inside other structures. For those cases you can use the `{*}` matcher in the path key. This matcher is often helpful when matching HasMany and BelongsToMany association data:

```
$data = [
    [
        'name' => 'James',
        'phone_numbers' => [
            ['number' => 'number-1'],
            ['number' => 'number-2'],
            ['number' => 'number-3'],
        ],
    ],
    [
        'name' => 'James',
```

(continues on next page)

(continued from previous page)

```
        'phone_numbers' => [
            ['number' => 'number-4'],
            ['number' => 'number-5'],
        ],
    ],
];

$numbers = (new Collection($data))->extract('phone_numbers.{*}.number');
$result = $numbers->toList();
// $result contains ['number-1', 'number-2', 'number-3', 'number-4', 'number-5']
```

This last example uses `toList()` unlike other examples, which is important when we're getting results with possibly duplicate keys. By using `toList()` we'll be guaranteed to get all values even if there are duplicate keys.

Unlike *Cake\Utility\Hash::extract()* this method only supports the `{*}` wildcard. All other wildcard and attributes matchers are not supported.

`Cake\Collection\Collection::combine`(*$keyPath*, *$valuePath*, *$groupPath = null*)

Collections allow you to create a new collection made from keys and values in an existing collection. Both the key and value paths can be specified with dot notation paths:

```
$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');
$result = $combined->toArray();

// $result contains
[
    1 => 'foo',
    2 => 'bar',
    3 => 'baz',
];
```

You can also optionally use a `groupPath` to group results based on a path:

```
$combined = (new Collection($items))->combine('id', 'name', 'parent');
$result = $combined->toArray();

// $result contains
[
    'a' => [1 => 'foo', 3 => 'baz'],
    'b' => [2 => 'bar']
];
```

Finally you can use *closures* to build keys/values/groups paths dynamically, for example when working with entities and dates (converted to `I18n\DateTime` instances by the ORM) you may want to group results by date:

```
$combined = (new Collection($entities))->combine(
    'id',
    function ($entity) { return $entity; },
```

(continues on next page)

(continued from previous page)

```
    function ($entity) { return $entity->date->toDateString(); }
);
 $result = $combined->toArray();

// $result contains
[
    '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*)

You can stop the iteration at any point using the `stopWhen()` method. Calling it in a collection will create a new one that will stop yielding results if the passed callable returns true for one of the elements:

```
$items = [10, 20, 50, 1, 2];
$collection = new Collection($items);

$new = $collection->stopWhen(function ($value, $key) {
    // Stop on the first value bigger than 30
    return $value > 30;
});

// $result contains [10, 20];
$result = $new->toList();
```

Cake\Collection\Collection::**unfold**(*callable $callback*)

Sometimes the internal items of a collection will contain arrays or iterators with more items. If you wish to flatten the internal structure to iterate once over all elements you can use the `unfold()` method. It will create a new collection that will yield every single element nested in the collection:

```
$items = [[1, 2, 3], [4, 5]];
$collection = new Collection($items);
$new = $collection->unfold();

// $result contains [1, 2, 3, 4, 5];
$result = $new->toList();
```

When passing a callable to `unfold()` you can control what elements will be unfolded from each item in the original collection. This is useful for returning data from paginated services:

```
$pages = [1, 2, 3, 4];
$collection = new Collection($pages);
$items = $collection->unfold(function ($page, $key) {
    // An imaginary web service that returns a page of results
    return MyService::fetchPage($page)->toList();
});

$allPagesItems = $items->toList();
```

If you are using PHP 5.5+, you can use the `yield` keyword inside `unfold()` to return as many elements for each item in the collection as you may need:

```
$oddNumbers = [1, 3, 5, 7];
$collection = new Collection($oddNumbers);
$new = $collection->unfold(function ($oddNumber) {
    yield $oddNumber;
    yield $oddNumber + 1;
});

// $result contains [1, 2, 3, 4, 5, 6, 7, 8];
$result = $new->toList();
```

Cake\Collection\Collection::**chunk**(*$chunkSize*)

When dealing with large amounts of items in a collection, it may make sense to process the elements in batches instead of one by one. For splitting a collection into multiple arrays of a certain size, you can use the `chunk()` function:

```
$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]]
```

The `chunk` function is particularly useful when doing batch processing, for example with a database result:

```
$collection = new Collection($articles);
$collection->map(function ($article) {
        // Change a property in the article
        $article->property = 'changed';
    })
    ->chunk(20)
    ->each(function ($batch) {
        myBulkSave($batch); // This function will be called for each batch
    });
```

Cake\Collection\Collection::**chunkWithKeys**(*$chunkSize*)

Much like *chunk()*, `chunkWithKeys()` allows you to slice up a collection into smaller batches but with keys preserved. This is useful when chunking associative arrays:

```
$collection = new Collection([
    'a' => 1,
    'b' => 2,
    'c' => 3,
    'd' => [4, 5]
]);
$chunked = $collection->chunkWithKeys(2);
$result = $chunked->toList();

// $result contains
[
    ['a' => 1, 'b' => 2],
    ['c' => 3, 'd' => [4, 5]]
]
```

Filtering

Cake\Collection\Collection::**filter**(*$callback*)

Collections allow you to filter and create new collections based on the result of callback functions. You can use `filter()` to create a new collection of elements matching a criteria 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*)

The inverse of `filter()` is `reject()`. This method does a negative filter, removing elements that match the filter function:

```
$collection = new Collection($people);
$ladies = $collection->reject(function ($person, $key) {
    return $person->gender === 'male';
});
```

Cake\Collection\Collection::**every**(*$callback*)

You can do truth tests with filter functions. To see if every element in a collection matches a test you can use `every()`:

```
$collection = new Collection($people);
$allYoungPeople = $collection->every(function ($person) {
    return $person->age < 21;
});
```

Cake\Collection\Collection::**some**(*$callback*)

You can see if the collection contains at least one element matching a filter function using the `some()` method:

```
$collection = new Collection($people);
$hasYoungPeople = $collection->some(function ($person) {
    return $person->age < 21;
});
```

Cake\Collection\Collection::**match**(*$conditions*)

If you need to extract a new collection containing only the elements that contain a given set of properties, you should use the `match()` method:

```
$collection = new Collection($comments);
$commentsFromMark = $collection->match(['user.name' => 'Mark']);
```

Cake\Collection\Collection::**firstMatch**(*$conditions*)

The property name can be a dot-separated path. You can traverse into nested entities and match the values they contain. When you only need the first matching element from a collection, you can use `firstMatch()`:

```
$collection = new Collection($comments);
$comment = $collection->firstMatch([
    'user.name' => 'Mark',
    'active' => true
]);
```

As you can see from the above, both `match()` and `firstMatch()` allow you to provide multiple conditions to match on. In addition, the conditions can be for different paths, allowing you to express complex conditions to match against.

Aggregation

Cake\Collection\Collection::**reduce**(*$callback*, *$initial*)

The counterpart of a `map()` operation is usually a `reduce`. This function will help you build a single result out of all the elements in a collection:

```
$totalPrice = $collection->reduce(function ($accumulated, $orderLine) {
    return $accumulated + $orderLine->price;
}, 0);
```

In the above example, `$totalPrice` will be the sum of all single prices contained in the collection. Note the second argument for the `reduce()` function takes the initial value for the reduce operation you are performing:

```
$allTags = $collection->reduce(function ($accumulated, $article) {
    return array_merge($accumulated, $article->tags);
}, []);
```

Cake\Collection\Collection::**min**(*string|callable $callback*, *$type = SORT_NUMERIC*)

To extract the minimum value for a collection based on a property, just use the `min()` function. This will return the full element from the collection and not just the smallest value found:

```
$collection = new Collection($people);
$youngest = $collection->min('age');

echo $youngest->name;
```

You are also able to express the property to compare by providing a path or a callback function:

```
$collection = new Collection($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*)

The same can be applied to the `max()` function, which will return a single element from the collection having the highest property value:

```
$collection = new Collection($people);
$oldest = $collection->max('age');

$personOldestChild = $collection->max(function ($person) {
    return $person->child->age;
});

$personWithOldestDad = $collection->max('dad.age');
```

Cake\Collection\Collection::**sumOf**(*$path = null*)

Finally, the `sumOf()` method will return the sum of a property of all elements:

```
$collection = new Collection($people);
$sumOfAges =  $collection->sumOf('age');

$sumOfChildrenAges = $collection->sumOf(function ($person) {
    return $person->child->age;
});

$sumOfDadAges = $collection->sumOf('dad.age');
```

Cake\Collection\Collection::**avg**(*$path = null*)

Calculate the average value of the elements in the collection. Optionally provide a matcher path, or function to extract values to generate the average for:

```
$items = [
   ['invoice' => ['total' => 100]],
   ['invoice' => ['total' => 200]],
];

// $average contains 150
$average = (new Collection($items))->avg('invoice.total');
```

Cake\Collection\Collection::**median**(*$path = null*)

Calculate the median value of a set of elements. Optionally provide a matcher path, or function to extract values to generate the median for:

```
$items = [
  ['invoice' => ['total' => 400]],
  ['invoice' => ['total' => 500]],
  ['invoice' => ['total' => 100]],
  ['invoice' => ['total' => 333]],
  ['invoice' => ['total' => 200]],
];

// $median contains 333
$median = (new Collection($items))->median('invoice.total');
```

Grouping and Counting

Cake\Collection\Collection::**groupBy**(*$callback*)

Collection values can be grouped by different keys in a new collection when they share the same value for a property:

```
$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');
$result = $studentsByGrade->toArray();

// $result contains
[
  10 => [
    ['name' => 'Andrew', 'grade' => 10],
    ['name' => 'Stacy', 'grade' => 10]
  ],
  9 => [
    ['name' => 'Mark', 'grade' => 9],
    ['name' => 'Barbara', 'grade' => 9]
  ]
]
```

As usual, it is possible to provide either a dot-separated path for nested properties or your own callback function to generate the groups dynamically:

```
$commentsByUserId = $comments->groupBy('user.id');

$classResults = $students->groupBy(function ($student) {
    return $student->grade > 6 ? 'approved' : 'denied';
});
```

Cake\Collection\Collection::**countBy**(*$callback*)

If you only wish to know the number of occurrences per group, you can do so by using the `countBy()` method. It takes the same arguments as `groupBy` so it should be already familiar to you:

```
$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*)

There will be certain cases where you know an element is unique for the property you want to group by. If you wish a single result per group, you can use the function `indexBy()`:

```
$usersById = $users->indexBy('id');

// When converted to array result could look like
[
    1 => 'markstory',
    3 => 'jose_zap',
    4 => 'jrbasso'
]
```

As with the `groupBy()` function you can also use a property path or a callback:

```
$articlesByAuthorId = $articles->indexBy('author.id');

$filesByHash = $files->indexBy(function ($file) {
    return md5($file);
});
```

Cake\Collection\Collection::**zip**(*$items*)

The elements of different collections can be grouped together using the `zip()` method. It will return a new collection containing an array grouping the elements from each collection that are placed at the same position:

```
$odds = new Collection([1, 3, 5]);
$pairs = new Collection([2, 4, 6]);
$combined = $odds->zip($pairs)->toList(); // [[1, 2], [3, 4], [5, 6]]
```

You can also zip multiple collections at once:

```
$years = new Collection([2013, 2014, 2015, 2016]);
$salaries = [1000, 1500, 2000, 2300];
$increments = [0, 500, 500, 300];

$rows = $years->zip($salaries, $increments);
$result = $rows->toList();

// $result contains
[
    [2013, 1000, 0],
    [2014, 1500, 500],
    [2015, 2000, 500],
    [2016, 2300, 300]
]
```

As you can already see, the `zip()` method is very useful for transposing multidimensional arrays:

```
$data = [
    2014 => ['jan' => 100, 'feb' => 200],
    2015 => ['jan' => 300, 'feb' => 500],
    2016 => ['jan' => 400, 'feb' => 600],
];

// Getting jan and feb data together

$firstYear = new Collection(array_shift($data));
```

(continues on next page)

(continued from previous page)

```
$result = $firstYear->zip($data[0], $data[1])->toList();

// Or $firstYear->zip(...$data) in PHP >= 5.6

// $result contains
[
    [100, 300, 400],
    [200, 500, 600]
]
```

Sorting

Cake\Collection\Collection::**sortBy**(*$callback*, *$order = SORT_DESC*, *$sort = SORT_NUMERIC*)

Collection values can be sorted in ascending or descending order based on a column or custom function. To create a new sorted collection out of the values of another one, you can use `sortBy`:

```
$collection = new Collection($people);
$sorted = $collection->sortBy('age');
```

As seen above, you can sort by passing the name of a column or property that is present in the collection values. You are also able to specify a property path instead using the dot notation. The next example will sort articles by their author's name:

```
$collection = new Collection($articles);
$sorted = $collection->sortBy('author.name');
```

The `sortBy()` method is flexible enough to let you specify an extractor function that will let you dynamically select the value to use for comparing two different values in the collection:

```
$collection = new Collection($articles);
$sorted = $collection->sortBy(function ($article) {
    return $article->author->name . '-' . $article->title;
});
```

In order to specify in which direction the collection should be sorted, you need to provide either `SORT_ASC` or `SORT_DESC` as the second parameter for sorting in ascending or descending direction respectively. By default, collections are sorted in descending direction:

```
$collection = new Collection($people);
$sorted = $collection->sortBy('age', SORT_ASC);
```

Sometimes you will need to specify which type of data you are trying to compare so that you get consistent results. For this purpose, you should supply a third argument in the `sortBy()` function with one of the following constants:

- **SORT_NUMERIC**: For comparing numbers
- **SORT_STRING**: For comparing string values
- **SORT_NATURAL**: For sorting string containing numbers and you'd like those numbers to be order in a natural way. For example: showing "10" after "2".
- **SORT_LOCALE_STRING**: For comparing strings based on the current locale.

By default, `SORT_NUMERIC` is used:

```
$collection = new Collection($articles);
$sorted = $collection->sortBy('title', SORT_ASC, SORT_NATURAL);
```

> **Warning:** It is often expensive to iterate sorted collections more than once. If you plan to do so, consider converting the collection to an array or simply use the `compile()` method on it.

Working with Tree Data

`Cake\Collection\Collection::nest(`*$idPath*, *$parentPath*, *$nestingKey = 'children'*`)`

Not all data is meant to be represented in a linear way. Collections make it easier to construct and flatten hierarchical or nested structures. Creating a nested structure where children are grouped by a parent identifier property can be done with the `nest()` method.

Two parameters are required for this function. The first one is the property representing the item identifier. The second parameter is the name of the property representing the identifier for the parent item:

```
$collection = 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'],
]);
$nested = $collection->nest('id', 'parent_id');
$result = $nested->toList();

// $result contains
[
    [
        'id' => 1,
        'parent_id' => null,
        'name' => 'Birds',
        'children' => [
            ['id' => 2, 'parent_id' => 1, 'name' => 'Land Birds', 'children' => []],
            ['id' => 3, 'parent_id' => 1, 'name' => 'Eagle', 'children' => []],
            ['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' => []],
        ],
    ],
];
```

Children elements are nested inside the `children` property inside each of the items in the collection. This type of data representation is helpful for rendering menus or traversing elements up to certain level in the tree.

`Cake\Collection\Collection::listNested`(*$order = 'desc'*, *$nestingKey = 'children'*)

The inverse of `nest()` is `listNested()`. This method allows you to flatten a tree structure back into a linear structure. It takes two parameters; the first one is the traversing mode (asc, desc or leaves), and the second one is the name of the property containing the children for each element in the collection.

Taking the input the nested collection built in the previous example, we can flatten it:

```
$result = $nested->listNested()->toList();

// $result contains
[
    ['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']
]
```

By default, the tree is traversed from the root to the leaves. You can also instruct it to only return the leaf elements in the tree:

```
$result = $nested->listNested('leaves')->toList();

// $result contains
[
    ['id' => 2, 'parent_id' => 1, 'name' => 'Land Birds', 'children' => [], ],
    ['id' => 3, 'parent_id' => 1, 'name' => 'Eagle', 'children' => [], ],
    ['id' => 4, 'parent_id' => 1, 'name' => 'Seagull', 'children' => [], ],
    ['id' => 5, 'parent_id' => 6, 'name' => 'Clown Fish', 'children' => [], ],
]
```

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:

```
$result = $nested->listNested()->printer('name', 'id', '--')->toArray();

// $result contains
[
    1 => 'Birds',
    2 => '--Land Birds',
    3 => '--Eagle',
    4 => '--Seagull',
    6 => 'Fish',
    5 => '--Clown Fish',
]
```

The `printer()` method also lets you use a callback to generate the keys and or values:

```
$nested->listNested()->printer(
    function ($el) {
```

(continues on next page)

(continued from previous page)

```
        return $el->name;
    },
    function ($el) {
        return $el->id;
    }
);
```

Other Methods

`Cake\Collection\Collection::isEmpty()`

Allows you to see if a collection contains any elements:

```
$collection = new Collection([]);
// Returns true
$collection->isEmpty();

$collection = new Collection([1]);
// Returns false
$collection->isEmpty();
```

`Cake\Collection\Collection::contains(*$value*)`

Collections allow you to quickly check if they contain one particular value: by using the `contains()` method:

```
$items = ['a' => 1, 'b' => 2, 'c' => 3];
$collection = new Collection($items);
$hasThree = $collection->contains(3);
```

Comparisons are performed using the === operator. If you wish to do looser comparison types you can use the `some()` method.

`Cake\Collection\Collection::shuffle()`

Sometimes you may wish to show a collection of values in a random order. In order to create a new collection that will return each value in a randomized position, use the `shuffle`:

```
$collection = new Collection(['a' => 1, 'b' => 2, 'c' => 3]);

// This could return [2, 3, 1]
$collection->shuffle()->toList();
```

`Cake\Collection\Collection::transpose()`

When you transpose a collection, you get a new collection containing a row made of the each of the original columns:

```
$items = [
    ['Products', '2012', '2013', '2014'],
    ['Product A', '200', '100', '50'],
    ['Product B', '300', '200', '100'],
    ['Product C', '400', '300', '200'],
];
```

(continues on next page)

(continued from previous page)

```
$transpose = (new Collection($items))->transpose();
$result = $transpose->toList();

// $result contains
[
    ['Products', 'Product A', 'Product B', 'Product C'],
    ['2012', '200', '300', '400'],
    ['2013', '100', '200', '300'],
    ['2014', '50', '100', '200'],
]
```

Withdrawing Elements

`Cake\Collection\Collection::sample`(*$length = 10*)

Shuffling a collection is often useful when doing quick statistical analysis. Another common operation when doing this sort of task is withdrawing a few random values out of a collection so that more tests can be performed on those. For example, if you wanted to select 5 random users to which you'd like to apply some A/B tests to, you can use the `sample()` function:

```
$collection = new Collection($people);

// Withdraw maximum 20 random users from this collection
$testSubjects = $collection->sample(20);
```

`sample()` will take at most the number of values you specify in the first argument. If there are not enough elements in the collection to satisfy the sample, the full collection in a random order is returned.

`Cake\Collection\Collection::take`(*$length*, *$offset*)

Whenever you want to take a slice of a collection use the `take()` function, it will create a new collection with at most the number of values you specify in the first argument, starting from the position passed in the second argument:

```
$topFive = $collection->sortBy('age')->take(5);

// Take 5 people from the collection starting from position 4
$nextTopFive = $collection->sortBy('age')->take(5, 4);
```

Positions are zero-based, therefore the first position number is `0`.

`Cake\Collection\Collection::skip`(*$length*)

While the second argument of `take()` can help you skip some elements before getting them from the collection, you can also use `skip()` for the same purpose as a way to take the rest of the elements after a certain position:

```
$collection = new Collection([1, 2, 3, 4]);
$allExceptFirstTwo = $collection->skip(2)->toList(); // [3, 4]
```

`Cake\Collection\Collection::first`()

One of the most common uses of `take()` is getting the first element in the collection. A shortcut method for achieving the same goal is using the `first()` method:

```
$collection = new Collection([5, 4, 3, 2]);
$collection->first(); // Returns 5
```

Cake\Collection\Collection::**last**()

Similarly, you can get the last element of a collection using the `last()` method:

```
$collection = new Collection([5, 4, 3, 2]);
$collection->last(); // Returns 2
```

Expanding Collections

Cake\Collection\Collection::**append**(*array|Traversable $items*)

You can compose multiple collections into a single one. This enables you to gather data from various sources, concatenate it, and apply other collection functions to it very smoothly. The `append()` method will return a new collection containing the values from both sources:

```
$cakephpTweets = new Collection($tweets);
$myTimeline = $cakephpTweets->append($phpTweets);

// Tweets containing `cakefest` from both sources
$myTimeline->filter(function ($tweet) {
    return strpos($tweet, 'cakefest');
});
```

Cake\Collection\Collection::**appendItem**(*$value*, *$key*)

Allows you to append an item with an optional key to the collection. If you specify a key that already exists in the collection, the value will not be overwritten:

```
$cakephpTweets = new Collection($tweets);
$myTimeline = $cakephpTweets->appendItem($newTweet, 99);
```

Cake\Collection\Collection::**prepend**(*$items*)

The `prepend()` method will return a new collection containing the values from both sources:

```
$cakephpTweets = new Collection($tweets);
$myTimeline = $cakephpTweets->prepend($phpTweets);
```

Cake\Collection\Collection::**prependItem**(*$value*, *$key*)

Allows you to prepend an item with an optional key to the collection. If you specify a key that already exists in the collection, the value will not be overwritten:

```
$cakephpTweets = new Collection($tweets);
$myTimeline = $cakephpTweets->prependItem($newTweet, 99);
```

Warning: When appending from different sources, you can expect some keys from both collections to be the same. For example, when appending two simple arrays. This can present a problem when converting a collection to an array using `toArray()`. If you do not want values from one collection to override others in the previous one based on their key, make sure that you call `toList()` in order to drop the keys and preserve all values.

Modifiying Elements

Cake\Collection\Collection::**insert**(*$path*, *$items*)

At times, you may have two separate sets of data that you would like to insert the elements of one set into each of the elements of the other set. This is a very common case when you fetch data from a data source that does not support data-merging or joins natively.

Collections offer an `insert()` method that will allow you to insert each of the elements in one collection into a property inside each of the elements of another collection:

```
$users = [
    ['username' => 'mark'],
    ['username' => 'juan'],
    ['username' => 'jose']
];

$languages = [
    ['PHP', 'Python', 'Ruby'],
    ['Bash', 'PHP', 'Javascript'],
    ['Javascript', 'Prolog']
];

$merged = (new Collection($users))->insert('skills', $languages);
$result = $merged->toArray();

// $result contains
[
    ['username' => 'mark', 'skills' => ['PHP', 'Python', 'Ruby']],
    ['username' => 'juan', 'skills' => ['Bash', 'PHP', 'Javascript']],
    ['username' => 'jose', 'skills' => ['Javascript', 'Prolog']]
];
```

The first parameter for the `insert()` method is a dot-separated path of properties to follow so that the elements can be inserted at that position. The second argument is anything that can be converted to a collection object.

Please observe that elements are inserted by the position they are found, thus, the first element of the second collection is merged into the first element of the first collection.

If there are not enough elements in the second collection to insert into the first one, then the target property will not be present:

```
$languages = [
    ['PHP', 'Python', 'Ruby'],
    ['Bash', 'PHP', 'Javascript']
];

$merged = (new Collection($users))->insert('skills', $languages);
$result = $merged->toArray();

// $result contains
[
    ['username' => 'mark', 'skills' => ['PHP', 'Python', 'Ruby']],
    ['username' => 'juan', 'skills' => ['Bash', 'PHP', 'Javascript']],
    ['username' => 'jose']
];
```

The `insert()` method can operate array elements or objects implementing the `ArrayAccess` interface.

Making Collection Methods Reusable

Using closures for collection methods is great when the work to be done is small and focused, but it can get messy very quickly. This becomes more obvious when a lot of different methods need to be called or when the length of the closure methods is more than just a few lines.

There are also cases when the logic used for the collection methods can be reused in multiple parts of your application. It is recommended that you consider extracting complex collection logic to separate classes. For example, imagine a lengthy closure like this one:

```
$collection
        ->map(function ($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;
        });
```

This can be refactored by creating another class:

```
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(`*$callback*`)`

Sometimes a chain of collection method calls can become reusable in other parts of your application, but only if they are called in that specific order. In those cases you can use `through()` in combination with a class implementing

`__invoke` to distribute your handy data processing calls:

```
$collection
        ->map(new ShippingCostCalculator)
        ->map(new TotalOrderCalculator)
        ->map(new GiftCardPriceReducer)
        ->buffered()
       ...
```

The above method calls can be extracted into a new class so they don't need to be repeated every time:

```
class FinalCheckOutRowProcessor
{
        public function __invoke($collection)
        {
                return $collection
                        ->map(new ShippingCostCalculator)
                        ->map(new TotalOrderCalculator)
                        ->map(new GiftCardPriceReducer)
                        ->buffered()
                       ...
        }
}

// Now you can use the through() method to call all methods at once
$collection->through(new FinalCheckOutRowProcessor);
```

Optimizing Collections

`Cake\Collection\Collection::buffered()`

Collections often perform most operations that you create using its functions in a lazy way. This means that even though you can call a function, it does not mean it is executed right away. This is true for a great deal of functions in this class. Lazy evaluation allows you to save resources in situations where you don't use all the values in a collection. You might not use all the values when iteration stops early, or when an exception/failure case is reached early.

Additionally, lazy evaluation helps speed up some operations. Consider the following example:

```
$collection = new Collection($oneMillionItems);
$collection = $collection->map(function ($item) {
    return $item * 2;
});
$itemsToShow = $collection->take(30);
```

Had the collections not been lazy, we would have executed one million operations, even though we only wanted to show 30 elements out of it. Instead, our map operation was only applied to the 30 elements we used. We can also derive benefits from this lazy evaluation for smaller collections when we do more than one operation on them. For example: calling `map()` twice and then `filter()`.

Lazy evaluation comes with its downside too. You could be doing the same operations more than once if you optimize a collection prematurely. Consider this example:

```
$ages = $collection->extract('age');
```

(continues on next page)

(continued from previous page)

```
$youngerThan30 = $ages->filter(function ($item) {
    return $item < 30;
});

$olderThan30 = $ages->filter(function ($item) {
    return $item > 30;
});
```

If we iterate both `youngerThan30` and `olderThan30`, the collection would unfortunately execute the `extract()` operation twice. This is because collections are immutable and the lazy-extracting operation would be done for both filters.

Luckily we can overcome this issue with a single function. If you plan to reuse the values from certain operations more than once, you can compile the results into another collection using the `buffered()` function:

```
$ages = $collection->extract('age')->buffered();
$youngerThan30 = ...
$olderThan30 = ...
```

Now, when both collections are iterated, they will only call the extracting operation once.

Making Collections Rewindable

The `buffered()` method is also useful for converting non-rewindable iterators into collections that can be iterated more than once:

```
// In PHP 5.5+
public function results()
{
    ...
    foreach ($transientElements as $e) {
        yield $e;
    }
}
$rewindable = (new Collection(results()))->buffered();
```

Cloning Collections

`Cake\Collection\Collection::compile(`*$preserveKeys = true*`)`

Sometimes you need to get a clone of the elements from another collection. This is useful when you need to iterate the same set from different places at the same time. In order to clone a collection out of another use the `compile()` method:

```
$ages = $collection->extract('age')->compile();

foreach ($ages as $age) {
    foreach ($collection as $element) {
        echo h($element->name) . ' - ' . $age;
    }
}
```

CHAPTER 33

Hash

class Cake\Utility**Hash**

Array management, if done right, can be a very powerful and useful tool for building smarter, more optimized code. CakePHP offers a very useful set of static utilities in the Hash class that allow you to do just that.

CakePHP's Hash class can be called from any model or controller in the same way Inflector is called. Example: *Hash::combine()*.

Hash Path Syntax

The path syntax described below is used by all the methods in `Hash`. Not all parts of the path syntax are available in all methods. A path expression is made of any number of tokens. Tokens are composed of two groups. Expressions, are used to traverse the array data, while matchers are used to qualify elements. You apply matchers to expression elements.

Expression Types

Expression	Definition
`{n}`	Represents a numeric key. Will match any string or numeric key.
`{s}`	Represents a string. Will match any string value including numeric string values.
`{*}`	Matches any value.
`Foo`	Matches keys with the exact same value.

All expression elements are supported by all methods. In addition to expression elements, you can use attribute matching with certain methods. They are `extract()`, `combine()`, `format()`, `check()`, `map()`, `reduce()`, `apply()`, `sort()`, `insert()`, `remove()` and `nest()`.

Attribute Matching Types

Matcher	Definition
`[id]`	Match elements with a given array key.
`[id=2]`	Match elements with id equal to 2.
`[id!=2]`	Match elements with id not equal to 2.
`[id>2]`	Match elements with id greater than 2.
`[id>=2]`	Match elements with id greater than or equal to 2.
`[id<2]`	Match elements with id less than 2
`[id<=2]`	Match elements with id less than or equal to 2.
`[text=/.../]`	Match elements that have values matching the regular expression inside `...`.

static Cake\Utility\Hash::**get**(*array|\ArrayAccess $data*, *$path*, *$default = null*)

`get()` is a simplified version of `extract()`, it only supports direct path expressions. Paths with `{n}`, `{s}`, `{*}` or matchers are not supported. Use `get()` when you want exactly one value out of an array. If a matching path is not found the default value will be returned.

static Cake\Utility\Hash::**extract**(*array|\ArrayAccess $data*, *$path*)

`Hash::extract()` supports all expression, and matcher components of *Hash Path Syntax*. You can use extract to retrieve data from arrays or object implementing `ArrayAccess` interface, along arbitrary paths quickly without having to loop through the data structures. Instead you use path expressions to qualify which elements you want returned

```
// Common Usage:
$users = [
    ['id' => 1, 'name' => 'mark'],
    ['id' => 2, 'name' => 'jane'],
    ['id' => 3, 'name' => 'sally'],
    ['id' => 4, 'name' => 'jose'],
];
$results = Hash::extract($users, '{n}.id');
// $results equals:
// [1,2,3,4];
```

static Cake\Utility\Hash::**insert**(*array $data*, *$path*, *$values = null*)

Inserts `$values` into an array as defined by `$path`:

```
$a = [
    'pages' => ['name' => 'page']
];
$result = Hash::insert($a, 'files', ['name' => 'files']);
// $result now looks like:
[
    [pages] => [
        [name] => page
    ]
    [files] => [
        [name] => files
    ]
]
```

You can use paths using `{n}`, `{s}` and `{*}` to insert data into multiple points:

```
$users = Hash::insert($users, '{n}.new', 'value');
```

Attribute matchers work with `insert()` as well:

```
$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 now looks like:
    [
        ['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*)

Removes all elements from an array that match `$path`.

```
$a = [
    'pages' => ['name' => 'page'],
    'files' => ['name' => 'files']
];
$result = Hash::remove($a, 'files');
/* $result now looks like:
    [
        [pages] => [
            [name] => page
        ]

    ]
*/
```

Using `{n}`, `{s}` and `{*}` will allow you to remove multiple values at once. You can also use attribute matchers with `remove()`:

```
$data = [
    0 => ['clear' => true, 'Item' => ['id' => 1, 'title' => 'first']],
    1 => ['Item' => ['id' => 2, 'title' => 'second']],
    2 => ['Item' => ['id' => 3, 'title' => 'third']],
    3 => ['clear' => true, 'Item' => ['id' => 4, 'title' => 'fourth']],
    4 => ['Item' => ['id' => 5, 'title' => 'fifth']],
];
$result = Hash::remove($data, '{n}[clear].Item[id=4]');
/* $result now looks like:
    [
        ['clear' => true, 'Item' => ['id' => 1, 'title' => 'first']],
```

(continues on next page)

(continued from previous page)

```
        ['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*)

Creates an associative array using a `$keyPath` as the path to build its keys, and optionally `$valuePath` as path to get the values. If `$valuePath` is not specified, or doesn't match anything, values will be initialized to null. You can optionally group the values by what is obtained when following the path specified in `$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 now looks like:
    [
        [2] =>
        [14] =>
    ]
*/

$result = Hash::combine($a, '{n}.User.id', '{n}.User.Data.user');
/* $result now looks like:
    [
        [2] => 'mariano.iglesias'
        [14] => 'phpnut'
    ]
*/

$result = Hash::combine($a, '{n}.User.id', '{n}.User.Data');
```

(continues on next page)

(continued from previous page)

```
/* $result now looks like:
    [
        [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 now looks like:
    [
        [2] => Mariano Iglesias
        [14] => Larry E. Masters
    ]
*/

$result = Hash::combine($a, '{n}.User.id', '{n}.User.Data', '{n}.User.group_id');
/* $result now looks like:
    [
        [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 now looks like:
    [
        [1] => [
                [2] => Mariano Iglesias
        ]
        [2] => [
                [14] => Larry E. Masters
        ]
    ]
*/
```

(continues on next page)

(continued from previous page)

```
// As of 3.9.0 $keyPath can be null
$result = Hash::combine($a, null, '{n}.User.Data.name');
/* $result now looks like:
    [
        [0] => Mariano Iglesias
        [1] => Larry E. Masters
    ]
*/
```

You can provide arrays for both `$keyPath` and `$valuePath`. If you do this, the first value will be used as a format string, for values extracted by the other paths:

```
$result = Hash::combine(
    $a,
    '{n}.User.id',
    ['%s: %s', '{n}.User.Data.user', '{n}.User.Data.name'],
    '{n}.User.group_id'
);
/* $result now looks like:
    [
        [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 now looks like:
    [
        [mariano.iglesias: Mariano Iglesias] => 2
        [phpnut: Larry E. Masters] => 14
    ]
*/
```

static Cake\Utility\Hash::**format**(*array $data*, *array $paths*, *$format*)

Returns a series of values extracted from an array, formatted with a format string:

```
$data = [
    [
        'Person' => [
            'first_name' => 'Nate',
            'last_name' => 'Abele',
            'city' => 'Boston',
            'state' => 'MA',
            'something' => '42'
```

(continues on next page)

(continued from previous page)

```
        ]
    ],
    [
        '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',
            '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*)

Determines if one Hash or array contains the exact keys and values of another:

```
$a = [
    0 => ['name' => 'main'],
    1 => ['name' => 'about']
];
$b = [
    0 => ['name' => 'main'],
    1 => ['name' => 'about'],
```

(continues on next page)

(continued from previous page)

```
    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*)

Checks if a particular path is set in an array:

```
$set = [
    'My Index 1' => ['First' => 'The first item']
];
$result = Hash::check($set, 'My Index 1.First');
// $result == true

$result = Hash::check($set, 'My Index 1');
// $result == true

$set = [
    'My Index 1' => [
        'First' => [
            'Second' => [
                'Third' => [
                    'Fourth' => 'Heavy. Nesting.'
                ]
            ]
        ]
    ]
];
$result = Hash::check($set, 'My Index 1.First.Second');
// $result == true

$result = Hash::check($set, 'My Index 1.First.Second.Third');
// $result == true

$result = Hash::check($set, 'My Index 1.First.Second.Third.Fourth');
// $result == true

$result = Hash::check($set, 'My Index 1.First.Seconds.Third.Fourth');
// $result == false
```

static Cake\Utility\Hash::**filter**(*array $data, $callback = ['Hash', 'filter']*)

Filters empty elements out of array, excluding '0'. You can also supply a custom `$callback` to filter the array elements. The callback should return `false` to remove elements from the resulting array:

```
$data = [
    '0',
```

(continues on next page)

(continued from previous page)

```
    false,
    true,
    0,
    ['one thing', 'I can tell you', 'is you got to be', false]
];
$res = Hash::filter($data);

/* $res now looks like:
    [
        [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 = '.'*)

Collapses a multi-dimensional array into a single dimension:

```
$arr = [
    [
        'Post' => ['id' => '1', 'title' => 'First Post'],
        'Author' => ['id' => '1', 'user' => 'Kyle'],
    ],
    [
        'Post' => ['id' => '2', 'title' => 'Second Post'],
        'Author' => ['id' => '3', 'user' => 'Crystal'],
    ],
];
$res = Hash::flatten($arr);
/* $res now looks like:
    [
        [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 = '.'*)

Expands an array that was previously flattened with *Hash::flatten()*:

```
$data = [
    '0.Post.id' => 1,
```

(continues on next page)

(continued from previous page)

```
    '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 now looks like:
[
    [
        '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*])

This function can be thought of as a hybrid between PHP's `array_merge` and `array_merge_recursive`. The difference to the two is that if an array key contains another array then the function behaves recursive (unlike `array_merge`) but does not do if for keys containing strings (unlike `array_merge_recursive`).

Note: This function will work with an unlimited amount of arguments and typecasts non-array parameters into arrays.

```
$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 now looks like:
[
    [0] => [
```

(continues on next page)

(continued from previous page)

```
            [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*)

Checks to see if all the values in the array are numeric:

```
$data = ['one'];
$res = Hash::numeric(array_keys($data));
// $res is true

$data = [1 => 'one'];
$res = Hash::numeric($data);
// $res is false
```

static Cake\Utility\Hash::**dimensions**(*array $data*)

Counts the dimensions of an array. This method will only consider the dimension of the first element in the array:

```
$data = ['one', '2', 'three'];
$result = Hash::dimensions($data);
// $result == 1

$data = ['1' => '1.1', '2', '3'];
$result = Hash::dimensions($data);
// $result == 1

$data = ['1' => ['1.1' => '1.1.1'], '2', '3' => ['3.1' => '3.1.1']];
$result = Hash::dimensions($data);
// $result == 2

$data = ['1' => '1.1', '2', '3' => ['3.1' => '3.1.1']];
$result = Hash::dimensions($data);
// $result == 1

$data = ['1' => ['1.1' => '1.1.1'], '2', '3' => ['3.1' => ['3.1.1' => '3.1.1.1']]];
$result = Hash::dimensions($data);
// $result == 2
```

static Cake\Utility\Hash::**maxDimensions**(*array $data*)

Similar to *dimensions()*, however this method returns, the deepest number of dimensions of any element in

the array:

```
$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*)

Creates a new array, by extracting `$path`, and mapping `$function` across the results. You can use both expression and matching elements with this method:

```
// Call the noop function $this->noop() on every element of $data
$result = Hash::map($data, "{n}", [$this, 'noop']);

public function noop(array $array)
{
    // Do stuff to array and return the result
    return $array;
}
```

static Cake\Utility\Hash::**reduce**(*array $data*, *$path*, *$function*)

Creates a single value, by extracting `$path`, and reducing the extracted results with `$function`. You can use both expression and matching elements with this method.

static Cake\Utility\Hash::**apply**(*array $data*, *$path*, *$function*)

Apply a callback to a set of extracted values using `$function`. The function will get the extracted values as the first 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 now looks like:
    [
        '01-01-2016' => 1,
        '02-01-2016' => 1,
    ]
*/
```

static Cake\Utility\Hash::**sort**(*array $data*, *$path*, *$dir*, *$type = 'regular'*)

Sorts an array by any value, determined by a *Hash Path Syntax* Only expression elements are supported by this method:

```
$a = [
    0 => ['Person' => ['name' => 'Jeff']],
    1 => ['Shirt' => ['color' => 'black']]
];
$result = Hash::sort($a, '{n}.Person.name', 'asc');
/* $result now looks like:
```

(continues on next page)

(continued from previous page)

```
    [
        [0] => [
                [Shirt] => [
                        [color] => black
                ]
        ]
        [1] => [
                [Person] => [
                        [name] => Jeff
                ]
        ]
    ]
*/
```

`$dir` can be either `asc` or `desc`. `$type` can be one of the following values:

- `regular` for regular sorting.
- `numeric` for sorting values as their numeric equivalents.
- `string` for sorting values as their string value.
- `natural` for sorting values in a human friendly way. Will sort `foo10` below `foo2` as an example.

static Cake\Utility\Hash::**diff**(*array $data*, *array $compare*)

Computes the difference between two arrays:

```
$a = [
    0 => ['name' => 'main'],
    1 => ['name' => 'about']
];
$b = [
    0 => ['name' => 'main'],
    1 => ['name' => 'about'],
    2 => ['name' => 'contact']
];

$result = Hash::diff($a, $b);
/* $result now looks like:
    [
        [2] => [
                [name] => contact
        ]
    ]
*/
```

static Cake\Utility\Hash::**mergeDiff**(*array $data*, *array $compare*)

This function merges two arrays and pushes the differences in data to the bottom of the resultant array.

Example 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']];
```

(continues on next page)

(continued from previous page)

```
$res = Hash::mergeDiff($array1, $array2);

/* $res now looks like:
    [
        [ModelOne] => [
                [id] => 1001
                [field_one] => a1.m1.f1
                [field_two] => a1.m1.f2
                [field_three] => a3.m1.f3
            ]
    ]
*/
```

Example 2

```
$array1 = ["a" => "b", 1 => 20938, "c" => "string"];
$array2 = ["b" => "b", 3 => 238, "c" => "string", ["extra_field"]];
$res = Hash::mergeDiff($array1, $array2);
/* $res now looks like:
    [
        [a] => b
        [1] => 20938
        [c] => string
        [b] => b
        [3] => 238
        [4] => [
                [0] => extra_field
        ]
    ]
*/
```

static Cake\Utility\Hash::**normalize**(*array $data*, *$assoc = true*, *$default = null*)

Normalizes an array. If `$assoc` is `true`, the resulting array will be normalized to be an associative array. Numeric keys with values, will be converted to string keys with `$default` values. Normalizing an array, makes using the results with *Hash::merge()* easier:

```
$a = ['Tree', 'CounterCache',
    'Upload' => [
        'folder' => 'products',
        'fields' => ['image_1_id', 'image_2_id']
    ]
];
$result = Hash::normalize($a);
/* $result now looks like:
    [
        [Tree] => null
        [CounterCache] => null
        [Upload] => [
                [folder] => products
                [fields] => [
                        [0] => image_1_id
                        [1] => image_2_id
```

(continues on next page)

(continued from previous page)

```
                ]
            ]
        ]
*/

$b = [
    'Cacheable' => ['enabled' => false],
    'Limit',
    'Bindable',
    'Validator',
    'Transactional',
];
$result = Hash::normalize($b);
/* $result now looks like:
    [
        [Cacheable] => [
                [enabled] => false
        ]

        [Limit] => null
        [Bindable] => null
        [Validator] => null
        [Transactional] => null
    ]
*/
```

Changed in version 4.5.0: The `$default` parameter was added.

static Cake\Utility\Hash::**nest**(*array $data*, *array $options = []*)

Takes a flat array set, and creates a nested, or threaded data structure.

Options:

- `children` The key name to use in the result set for children. Defaults to 'children'.
- `idPath` The path to a key that identifies each entry. Should be compatible with *Hash::extract()*. Defaults to `{n}.$alias.id`
- `parentPath` The path to a key that identifies the parent of each entry. Should be compatible with *Hash::extract()*. Defaults to `{n}.$alias.parent_id`
- `root` The id of the desired top-most result.

For example, if you had the following array of data:

```
$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]],
```

(continues on next page)

(continued from previous page)

```
    ['ThreadPost' => ['id' => 10, 'parent_id' => 6]]
];

$result = Hash::nest($data, ['root' => 6]);
/* $result now looks like:
    [
        (int) 0 => [
            'ThreadPost' => [
                'id' => (int) 6,
                'parent_id' => null
            ],
            'children' => [
                (int) 0 => [
                    'ThreadPost' => [
                        'id' => (int) 7,
                        'parent_id' => (int) 6
                    ],
                    'children' => []
                ],
                (int) 1 => [
                    'ThreadPost' => [
                        'id' => (int) 8,
                        'parent_id' => (int) 6
                    ],
                    'children' => []
                ],
                (int) 2 => [
                    'ThreadPost' => [
                        'id' => (int) 9,
                        'parent_id' => (int) 6
                    ],
                    'children' => []
                ],
                (int) 3 => [
                    'ThreadPost' => [
                        'id' => (int) 10,
                        'parent_id' => (int) 6
                    ],
                    'children' => []
                ]
            ]
        ]
    ]
    */
```

CHAPTER 34

Http Client

class Cake\Http**Client**(*mixed $config = []*)

CakePHP includes a PSR-18 compliant HTTP client which can be used for making requests. It is a great way to communicate with webservices, and remote APIs.

Doing Requests

Doing requests is simple and straight forward. Doing a GET request looks like:

```
use Cake\Http\Client;

$http = new Client();

// Simple get
$response = $http->get('http://example.com/test.html');

// Simple get with querystring
$response = $http->get('http://example.com/search', ['q' => 'widget']);

// Simple get with querystring & additional headers
$response = $http->get('http://example.com/search', ['q' => 'widget'], [
  'headers' => ['X-Requested-With' => 'XMLHttpRequest'],
]);
```

Doing POST and PUT requests is equally simple:

```
// Send a POST request with application/x-www-form-urlencoded encoded data
$http = new Client();
```

(continues on next page)

(continued from previous page)

```
$response = $http->post('http://example.com/posts/add', [
  'title' => 'testing',
  'body' => 'content in the post',
]);

// Send a PUT request with application/x-www-form-urlencoded encoded data
$response = $http->put('http://example.com/posts/add', [
  'title' => 'testing',
  'body' => 'content in the post',
]);

// Other methods as well.
$http->delete(...);
$http->head(...);
$http->patch(...);
```

If you have created a PSR-7 request object you can send it using `sendRequest()`:

```
use Cake\Http\Client;
use Cake\Http\Client\Request as ClientRequest;

$request = new ClientRequest(
    'http://example.com/search',
    ClientRequest::METHOD_GET
);
$client = new Client();
$response = $client->sendRequest($request);
```

Creating Multipart Requests with Files

You can include files in request bodies by including a filehandle in the array:

```
$http = new Client();
$response = $http->post('http://example.com/api', [
  'image' => fopen('/path/to/a/file', 'r'),
]);
```

The filehandle will be read until its end; it will not be rewound before being read.

Building Multipart Request Bodies

There may be times when you need to build a request body in a very specific way. In these situations you can often use `Cake\Http\Client\FormData` to craft the specific multipart HTTP request you want:

```
use Cake\Http\Client\FormData;

$data = new FormData();

// Create an XML part
$xml = $data->newPart('xml', $xmlString);
```

(continues on next page)

(continued from previous page)

```
// Set the content type.
$xml->type('application/xml');
$data->add($xml);

// Create a file upload with addFile()
// This will append the file to the form data as well.
$file = $data->addFile('upload', fopen('/some/file.txt', 'r'));
$file->contentId('abc123');
$file->disposition('attachment');

// Send the request.
$response = $http->post(
    'http://example.com/api',
    (string)$data,
    ['headers' => ['Content-Type' => $data->contentType()]]
);
```

Sending Request Bodies

When dealing with REST APIs you often need to send request bodies that are not form encoded. Http\Client exposes this through the type option:

```
// Send a JSON request body.
$http = new Client();
$response = $http->post(
  'http://example.com/tasks',
  json_encode($data),
  ['type' => 'json']
);
```

The `type` key can either be a one of 'json', 'xml' or a full mime type. When using the `type` option, you should provide the data as a string. If you're doing a GET request that needs both querystring parameters and a request body you can do the following:

```
// Send a JSON body in a GET request with query string parameters.
$http = new Client();
$response = $http->get(
  'http://example.com/tasks',
  ['q' => 'test', '_content' => json_encode($data)],
  ['type' => 'json']
);
```

Request Method Options

Each HTTP method takes an `$options` parameter which is used to provide addition request information. The following keys can be used in `$options`:

- `headers` - Array of additional headers
- `cookie` - Array of cookies to use.
- `proxy` - Array of proxy information.
- `auth` - Array of authentication data, the `type` key is used to delegate to an authentication strategy. By default Basic auth is used.
- `ssl_verify_peer` - defaults to `true`. Set to `false` to disable SSL certification verification (not recommended).
- `ssl_verify_peer_name` - defaults to `true`. Set to `false` to disable host name verification when verifying SSL certificates (not recommended).
- `ssl_verify_depth` - defaults to 5. Depth to traverse in the CA chain.
- `ssl_verify_host` - defaults to `true`. Validate the SSL certificate against the host name.
- `ssl_cafile` - defaults to built in cafile. Overwrite to use custom CA bundles.
- `timeout` - Duration to wait before timing out in seconds.
- `type` - Send a request body in a custom content type. Requires `$data` to either be a string, or the `_content` option to be set when doing GET requests.
- `redirect` - Number of redirects to follow. Defaults to `false`.
- `curl` - An array of additional curl options (if the curl adapter is used), for example, `[CURLOPT_SSLKEY => 'key.pem']`.

The options parameter is always the 3rd parameter in each of the HTTP methods. They can also be used when constructing `Client` to create *scoped clients*.

Authentication

`Cake\Http\Client` supports a few different authentication systems. Different authentication strategies can be added by developers. Auth strategies are called before the request is sent, and allow headers to be added to the request context.

Using Basic Authentication

An example of basic authentication:

```
$http = new Client();
$response = $http->get('http://example.com/profile/1', [], [
  'auth' => ['username' => 'mark', 'password' => 'secret'],
]);
```

By default `Cake\Http\Client` will use basic authentication if there is no `'type'` key in the auth option.

Using Digest Authentication

An example of basic authentication:

```
$http = new Client();
$response = $http->get('http://example.com/profile/1', [], [
    'auth' => [
        'type' => 'digest',
        'username' => 'mark',
        'password' => 'secret',
        'realm' => 'myrealm',
        'nonce' => 'onetimevalue',
        'qop' => 1,
        'opaque' => 'someval',
    ],
]);
```

By setting the 'type' key to 'digest', you tell the authentication subsystem to use digest authentication. Digest authentication supports the following algorithms:

- MD5
- SHA-256
- SHA-512-256
- MD5-sess
- SHA-256-sess
- SHA-512-256-sess

The algorithm will be automatically chosen based on the server challenge.

OAuth 1 Authentication

Many modern web-services require OAuth authentication to access their APIs. The included OAuth authentication assumes that you already have your consumer key and consumer secret:

```
$http = new Client();
$response = $http->get('http://example.com/profile/1', [], [
    'auth' => [
        'type' => 'oauth',
        'consumerKey' => 'bigkey',
        'consumerSecret' => 'secret',
        'token' => '...',
        'tokenSecret' => '...',
        'realm' => 'tickets',
    ],
]);
```

OAuth 2 Authentication

Because OAuth2 is often a single header, there is not a specialized authentication adapter. Instead you can create a client with the access token:

```
$http = new Client([
    'headers' => ['Authorization' => 'Bearer ' . $accessToken],
]);
$response = $http->get('https://example.com/api/profile/1');
```

Proxy Authentication

Some proxies require authentication to use them. Generally this authentication is Basic, but it can be implemented by any authentication adapter. By default Http\Client will assume Basic authentication, unless the type key is set:

```
$http = new Client();
$response = $http->get('http://example.com/test.php', [], [
    'proxy' => [
        'username' => 'mark',
        'password' => 'testing',
        'proxy' => '127.0.0.1:8080',
    ],
]);
```

The second proxy parameter must be a string with an IP or a domain without protocol. The username and password information will be passed through the request headers, while the proxy string will be passed through stream_context_create()[180].

Creating Scoped Clients

Having to re-type the domain name, authentication and proxy settings can become tedious & error prone. To reduce the chance for mistake and relieve some of the tedium, you can create scoped clients:

```
// Create a scoped client.
$http = new Client([
    'host' => 'api.example.com',
    'scheme' => 'https',
    'auth' => ['username' => 'mark', 'password' => 'testing'],
]);

// Do a request to api.example.com
$response = $http->get('/test.php');
```

If your scoped client only needs information from the URL you can use `createFromUrl()`:

```
$http = Client::createFromUrl('https://api.example.com/v1/test');
```

The above would create a client instance with the `protocol`, `host`, and `basePath` options set.

The following information can be used when creating a scoped client:

[180] https://php.net/manual/en/function.stream-context-create.php

- host
- basePath
- scheme
- proxy
- auth
- port
- cookies
- timeout
- ssl_verify_peer
- ssl_verify_depth
- ssl_verify_host

Any of these options can be overridden by specifying them when doing requests. host, scheme, proxy, port are overridden in the request URL:

```
// Using the scoped client we created earlier.
$response = $http->get('http://foo.com/test.php');
```

The above will replace the domain, scheme, and port. However, this request will continue using all the other options defined when the scoped client was created. See *Request Method Options* for more information on the options supported.

Setting and Managing Cookies

Http\Client can also accept cookies when making requests. In addition to accepting cookies, it will also automatically store valid cookies set in responses. Any response with cookies, will have them stored in the originating instance of Http\Client. The cookies stored in a Client instance are automatically included in future requests to domain + path combinations that match:

```
$http = new Client([
    'host' => 'cakephp.org'
]);

// Do a request that sets some cookies
$response = $http->get('/');

// Cookies from the first request will be included
// by default.
$response2 = $http->get('/changelogs');
```

You can always override the auto-included cookies by setting them in the request's `$options` parameters:

```
// Replace a stored cookie with a custom value.
$response = $http->get('/changelogs', [], [
    'cookies' => ['sessionid' => '123abc'],
]);
```

You can add cookie objects to the client after creating it using the `addCookie()` method:

```
use Cake\Http\Cookie\Cookie;

$http = new Client([
    'host' => 'cakephp.org'
]);
$http->addCookie(new Cookie('session', 'abc123'));
```

Response Objects

class Cake\Http\Client**Response**

Response objects have a number of methods for inspecting the response data.

Reading Response Bodies

You read the entire response body as a string:

```
// Read the entire response as a string.
$response->getStringBody();
```

You can also access the stream object for the response and use its methods:

```
// Get a Psr\Http\Message\StreamInterface containing the response body
$stream = $response->getBody();

// Read a stream 100 bytes at a time.
while (!$stream->eof()) {
    echo $stream->read(100);
}
```

Reading JSON and XML Response Bodies

Since JSON and XML responses are commonly used, response objects provide a way to use accessors to read decoded data. JSON data is decoded into an array, while XML data is decoded into a `SimpleXMLElement` tree:

```
// Get some XML
$http = new Client();
$response = $http->get('http://example.com/test.xml');
$xml = $response->getXml();

// Get some JSON
$http = new Client();
$response = $http->get('http://example.com/test.json');
$json = $response->getJson();
```

The decoded response data is stored in the response object, so accessing it multiple times has no additional cost.

Accessing Response Headers

You can access headers through a few different methods. Header names are always treated as case-insensitive values when accessing them through methods:

```
// Get all the headers as an associative array.
$response->getHeaders();

// Get a single header as an array.
$response->getHeader('content-type');

// Get a header as a string
$response->getHeaderLine('content-type');

// Get the response encoding
$response->getEncoding();
```

Accessing Cookie Data

You can read cookies with a few different methods depending on how much data you need about the cookies:

```
// Get all cookies (full data)
$response->getCookies();

// Get a single cookie's value.
$response->getCookie('session_id');

// Get a the complete data for a single cookie
// includes value, expires, path, httponly, secure keys.
$response->getCookieData('session_id');
```

Checking the Status Code

Response objects provide a few methods for checking status codes:

```
// Was the response a 20x
$response->isOk();

// Was the response a 30x
$response->isRedirect();

// Get the status code
$response->getStatusCode();
```

Changing Transport Adapters

By default `Http\Client` will prefer using a `curl` based transport adapter. If the curl extension is not available a stream based adapter will be used instead. You can force select a transport adapter using a constructor option:

```
use Cake\Http\Client\Adapter\Stream;

$client = new Client(['adapter' => Stream::class]);
```

Testing

trait Cake\Http\TestSuite**HttpClientTrait**

In tests you will often want to create mock responses to external APIs. You can use the `HttpClientTrait` to define responses to the requests your application is making:

```
use Cake\Http\TestSuite\HttpClientTrait;
use Cake\TestSuite\TestCase;

class CartControllerTests extends TestCase
{
    use HttpClientTrait;

    public function testCheckout()
    {
        // Mock a POST request that will be made.
        $this->mockClientPost(
            'https://example.com/process-payment',
            $this->newClientResponse(200, [], json_encode(['ok' => true]))
        );
        $this->post("/cart/checkout");
        // Do assertions.
    }
}
```

There are methods to mock the most commonly used HTTP methods:

```
$this->mockClientGet(...);
$this->mockClientPatch(...);
$this->mockClientPost(...);
$this->mockClientPut(...);
$this->mockClientDelete(...);
```

Cake\Http\TestSuite\HttpClientTrait::**newClientResponse**(*int $code = 200*, *array $headers = []*, *string $body = ''*)

As seen above you can use the `newClientResponse()` method to create responses for the requests your application will make. The headers need to be a list of strings:

```
$headers = [
    'Content-Type: application/json',
```

(continues on next page)

(continued from previous page)

```
    'Connection: close',
];
$response = $this->newClientResponse(200, $headers, $body)
```

CHAPTER 35

Inflector

class Cake\Utility**Inflector**

The Inflector class takes a string and can manipulate it to handle word variations such as pluralization or camelizing and is normally accessed statically. Example: `Inflector::pluralize('example')` returns "examples".

You can try out the inflections online at inflector.cakephp.org[181] or sandbox.dereuromark.de[182].

Summary of Inflector Methods and Their Output

Quick summary of the Inflector built-in methods and the results they output when provided a multi-word argument:

[181] https://inflector.cakephp.org/
[182] https://sandbox.dereuromark.de/sandbox/inflector

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

Creating Plural & Singular Forms

static Cake\Utility\Inflector::**singularize**(*$singular*)

static Cake\Utility\Inflector::**pluralize**(*$singular*)

Both `pluralize` and `singularize()` work on most English nouns. If you need to support other languages, you can use *Inflection Configuration* to customize the rules used:

```
// Apples
echo Inflector::pluralize('Apple');
```

Note: `pluralize()` should not be used on a noun that is already in its plural form.

```
// Person
echo Inflector::singularize('People');
```

Note: `singularize()` should not be used on a noun that is already in its singular form.

Creating CamelCase and under_scored Forms

static Cake\Utility\Inflector::**camelize**(*$underscored*)

static Cake\Utility\Inflector::**underscore**(*$camelCase*)

These methods are useful when creating class names, or property names:

```
// ApplePie
Inflector::camelize('Apple_pie')

// apple_pie
Inflector::underscore('ApplePie');
```

It should be noted that underscore will only convert camelCase formatted words. Words that contains spaces will be lower-cased, but will not contain an underscore.

Creating Human Readable Forms

static Cake\Utility\Inflector::**humanize**(*$underscored*)

This method is useful when converting underscored forms into "Title Case" forms for human readable values:

```
// Apple Pie
Inflector::humanize('apple_pie');
```

Creating Table and Class Name Forms

static Cake\Utility\Inflector::**classify**(*$underscored*)

static Cake\Utility\Inflector::**dasherize**(*$dashed*)

static Cake\Utility\Inflector::**tableize**(*$camelCase*)

When generating code, or using CakePHP's conventions you may need to inflect table names or class names:

```
// UserProfileSetting
Inflector::classify('user_profile_settings');

// user-profile-setting
Inflector::dasherize('UserProfileSetting');

// user_profile_settings
Inflector::tableize('UserProfileSetting');
```

Creating Variable Names

static Cake\Utility\Inflector::**variable**(*$underscored*)

Variable names are often useful when doing meta-programming tasks that involve generating code or doing work based on conventions:

```
// applePie
Inflector::variable('apple_pie');
```

Inflection Configuration

CakePHP's naming conventions can be really nice - you can name your database table `big_boxes`, your model `BigBoxes`, your controller `BigBoxesController`, and everything just works together automatically. The way CakePHP knows how to tie things together is by *inflecting* the words between their singular and plural forms.

There are occasions (especially for our non-English speaking friends) where you may run into situations where CakePHP's inflector (the class that pluralizes, singularizes, camelCases, and under_scores) might not work as you'd like. If CakePHP won't recognize your Foci or Fish, you can tell CakePHP about your special cases.

Loading Custom Inflections

static Cake\Utility\Inflector::**rules**(*$type*, *$rules*, *$reset = false*)

Define new inflection and transliteration rules for Inflector to use. Often, this method is used in your **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
```

The supplied rules will be merged into the respective inflection sets defined in `Cake/Utility/Inflector`, with the added rules taking precedence over the core rules. You can use `Inflector::reset()` to clear rules and restore the original Inflector state.

CHAPTER 36

Number

class Cake\I18n**Number**

If you need `NumberHelper` functionalities outside of a `View`, use the `Number` class:

```
namespace App\Controller;

use Cake\I18n\Number;

class UsersController extends AppController
{
    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('Authentication.Authentication');
    }

    public function afterLogin()
    {
        $identity = $this->Authentication->getIdentity();
        $storageUsed = $identity->storage_used;
        if ($storageUsed > 5000000) {
            // Notify users of quota
            $this->Flash->success(__('You are using {0} storage', Number::toReadableSize(
↪$storageUsed)));
        }
    }
}
```

All of these functions return the formatted number; they do not automatically echo the output into the view.

Formatting Currency Values

Cake\I18n\Number::**currency**(*mixed $value*, *string $currency = null*, *array $options = []*)

This method is used to display a number in common currency formats (EUR, GBP, USD), based on the 3-letter ISO 4217 currency code. Usage in a view looks like:

```
// Called as NumberHelper
echo $this->Number->currency($value, $currency);

// Called as Number
echo Number::currency($value, $currency);
```

The first parameter, `$value`, should be a floating point number that represents the amount of money you are expressing. The second parameter is a string used to choose a predefined currency formatting scheme:

$currency	1234.56, formatted by currency type
EUR	€1.234,56
GBP	£1,234.56
USD	$1,234.56

The third parameter is an array of options for further defining the output. The following options are available:

Option	Description
before	Text to display before the rendered number.
after	Text to display after the rendered number.
zero	The text to use for zero values; can be a string or a number. ie. 0, 'Free!'.
places	Number of decimal places to use, ie. 2
precision	Maximal number of decimal places to use, ie. 2
locale	The locale name to use for formatting number, ie. "fr_FR".
fractionSymbol	String to use for fraction numbers, ie. ' cents'.
fractionPosition	Either 'before' or 'after' to place the fraction symbol.
pattern	An ICU number pattern to use for formatting the number ie. #,###.00
useIntlCode	Set to `true` to replace the currency symbol with the international currency code.

If `$currency` value is `null`, the default currency will be retrieved from `Cake\I18n\Number::defaultCurrency()`. To format currencies in an accounting format you should set the currency format:

```
Number::setDefaultCurrencyFormat(Number::FORMAT_CURRENCY_ACCOUNTING);
```

Setting the Default Currency

Cake\I18n\Number::**setDefaultCurrency**(*$currency*)

Setter for the default currency. This removes the need to always pass the currency to *Cake\I18n\Number::currency()* and change all currency outputs by setting other default. If `$currency` is set to `null`, it will clear the currently stored value.

Getting the Default Currency

`Cake\I18n\Number::getDefaultCurrency()`

Getter for the default currency. If default currency was set earlier using `setDefaultCurrency()`, then that value will be returned. By default, it will retrieve the `intl.default_locale` ini value if set and `'en_US'` if not.

Formatting Floating Point Numbers

`Cake\I18n\Number::precision(`*float $value*, *int $precision = 3*, *array $options = []*`)`

This method displays a number with the specified amount of precision (decimal places). It will round in order to maintain the level of precision defined.

```
// Called as NumberHelper
echo $this->Number->precision(456.91873645, 2);

// Outputs
456.92

// Called as Number
echo Number::precision(456.91873645, 2);
```

Formatting Percentages

`Cake\I18n\Number::toPercentage(`*mixed $value*, *int $precision = 2*, *array $options = []*`)`

Option	Description
multiply	Boolean to indicate whether the value has to be multiplied by 100. Useful for decimal percentages.

Like *Cake\I18n\Number::precision()*, this method formats a number according to the supplied precision (where numbers are rounded to meet the given precision). This method also expresses the number as a percentage and appends the output with a percent sign.

```
// Called as NumberHelper. Output: 45.69%
echo $this->Number->toPercentage(45.691873645);

// Called as Number. Output: 45.69%
echo Number::toPercentage(45.691873645);

// Called with multiply. Output: 45.7%
echo Number::toPercentage(0.45691, 1, [
    'multiply' => true
]);
```

Interacting with Human Readable Values

`Cake\I18n\Number::toReadableSize(string $size)`

This method formats data sizes in human readable forms. It provides a shortcut way to convert bytes to KB, MB, GB, and TB. The size is displayed with a two-digit precision level, according to the size of data supplied (i.e. higher sizes are expressed in larger terms):

```
// Called as 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

// Called as 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
```

Formatting Numbers

`Cake\I18n\Number::format(mixed $value, array $options = [])`

This method gives you much more control over the formatting of numbers for use in your views (and is used as the main method by most of the other NumberHelper methods). Using this method might looks like:

```
// Called as NumberHelper
$this->Number->format($value, $options);

// Called as Number
Number::format($value, $options);
```

The `$value` parameter is the number that you are planning on formatting for output. With no `$options` supplied, the number 1236.334 would output as 1,236. Note that the default precision is zero decimal places.

The `$options` parameter is where the real magic for this method resides.

- If you pass an integer then this becomes the amount of precision or places for the function.
- If you pass an associated array, you can use the following keys:

Option	Description
places	Number of decimal places to use, ie. 2
precision	Maximum number of decimal places to use, ie. 2
pattern	An ICU number pattern to use for formatting the number ie. #,###.00
locale	The locale name to use for formatting number, ie. “fr_FR”.
before	Text to display before the rendered number.
after	Text to display after the rendered number.

Example:

```
// Called as NumberHelper
echo $this->Number->format('123456.7890', [
    'places' => 2,
    'before' => '¥ ',
    'after' => ' !'
]);
// Output '¥ 123,456.79 !'

echo $this->Number->format('123456.7890', [
    'locale' => 'fr_FR'
]);
// Output '123 456,79 !'

// Called as Number
echo Number::format('123456.7890', [
    'places' => 2,
    'before' => '¥ ',
    'after' => ' !'
]);
// Output '¥ 123,456.79 !'

echo Number::format('123456.7890', [
    'locale' => 'fr_FR'
]);
// Output '123 456,79 !'
```

Cake\I18n\Number::**ordinal**(*mixed $value*, *array $options = []*)

This method will output an ordinal number.

Examples:

```
echo Number::ordinal(1);
// Output '1st'

echo Number::ordinal(2);
// Output '2nd'

echo Number::ordinal(2, [
    'locale' => 'fr_FR'
]);
// Output '2e'

echo Number::ordinal(410);
// Output '410th'
```

Format Differences

Cake\I18n\Number::**formatDelta**(*mixed $value*, *array $options = []*)

This method displays differences in value as a signed number:

```
// Called as NumberHelper
$this->Number->formatDelta($value, $options);

// Called as Number
Number::formatDelta($value, $options);
```

The `$value` parameter is the number that you are planning on formatting for output. With no `$options` supplied, the number 1236.334 would output as 1,236. Note that the default precision is zero decimal places.

The `$options` parameter takes the same keys as *Number::format()* itself:

Option	Description
places	Number of decimal places to use, ie. 2
precision	Maximum number of decimal places to use, ie. 2
locale	The locale name to use for formatting number, ie. “fr_FR”.
before	Text to display before the rendered number.
after	Text to display after the rendered number.

Example:

```
// Called as NumberHelper
echo $this->Number->formatDelta('123456.7890', [
    'places' => 2,
    'before' => '[',
    'after' => ']'
]);
// Output '[+123,456.79]'

// Called as Number
echo Number::formatDelta('123456.7890', [
    'places' => 2,
    'before' => '[',
    'after' => ']'
]);
// Output '[+123,456.79]'
```

Configure formatters

Cake\I18n\Number::**config**(*string $locale*, *int $type = NumberFormatter::DECIMAL*, *array $options = []*)

This method allows you to configure formatter defaults which persist across calls to various methods.

Example:

```
Number::config('en_IN', \NumberFormatter::CURRENCY, [
    'pattern' => '#,##,##0'
]);
```

CHAPTER 37

Registry Objects

The registry classes provide a simple way to create and retrieve loaded instances of a given object type. There are registry classes for Components, Helpers, Tasks, and Behaviors.

While the examples below will use Components, the same behavior can be expected for Helpers, Behaviors, and Tasks in addition to Components.

Loading Objects

Objects can be loaded on-the-fly using add<registry-object>() Example:

```
$this->loadComponent('Acl.Acl');
$this->addHelper('Flash')
```

This will result in the `Acl` property and `Flash` helper being loaded. Configuration can also be set on-the-fly. Example:

```
$this->loadComponent('Cookie', ['name' => 'sweet']);
```

Any keys and values provided will be passed to the Component's constructor. The one exception to this rule is `className`. Classname is a special key that is used to alias objects in a registry. This allows you to have component names that do not reflect the classnames, which can be helpful when extending core components:

```
$this->Flash = $this->loadComponent('Flash', ['className' => 'MyCustomFlash']);
$this->Flash->error(); // Actually using MyCustomFlash::error();
```

Triggering Callbacks

Callbacks are not provided by registry objects. You should use the *events system* to dispatch any events/callbacks for your application.

Disabling Callbacks

In previous versions, collection objects provided a `disable()` method to disable objects from receiving callbacks. You should use the features in the events system to accomplish this now. For example, you could disable component callbacks in the following way:

```
// Remove MyComponent from callbacks.
$this->getEventManager()->off($this->MyComponent);

// Re-enable MyComponent for callbacks.
$this->getEventManager()->on($this->MyComponent);
```

CHAPTER 38

Text

class Cake\Utility**Text**

The Text class includes convenience methods for creating and manipulating strings and is normally accessed statically. Example: `Text::uuid()`.

If you need *Cake\View\Helper\TextHelper* functionalities outside of a `View`, use the `Text` class:

```
namespace App\Controller;

use Cake\Utility\Text;

class UsersController extends AppController
{
    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('Auth')
    };

    public function afterLogin()
    {
        $message = $this->Users->find('new_message')->first();
        if (!empty($message)) {
            // Notify user of new message
            $this->Flash->success(__(
                'You have a new message: {0}',
                Text::truncate($message['Message']['body'], 255, ['html' => true])
            ));
        }
    }
}
```

Convert Strings into ASCII

static Cake\Utility\Text::**transliterate**(*$string*, *$transliteratorId = null*)

Transliterate by default converts all characters in provided string into equivalent ASCII characters. The method expects UTF-8 encoding. The character conversion can be controlled using transliteration identifiers which you can pass using the `$transliteratorId` argument or change the default identifier string using `Text::setTransliteratorId()`. ICU transliteration identifiers are basically of form `<source script>:<target script>` and you can specify multiple conversion pairs separated by `;`. You can find more info about transliterator identifiers here[183]:

```
// apple puree
Text::transliterate('apple purée');

// Ubermensch (only latin characters are transliterated)
Text::transliterate('Übérmensch', 'Latin-ASCII;');
```

Creating URL Safe Strings

static Cake\Utility\Text::**slug**(*$string*, *$options = []*)

Slug transliterates all characters into ASCII versions and converting unmatched characters and spaces to dashes. The slug method expects UTF-8 encoding.

You can provide an array of options that controls slug. `$options` can also be a string in which case it will be used as replacement string. The supported options are:

- `replacement` Replacement string, defaults to '-'.
- `transliteratorId` A valid tranliterator id string. If default `null` `Text::$_defaultTransliteratorId` to be used. If `false` no transliteration will be done, only non words will be removed.
- `preserve` Specific non-word character to preserve. Defaults to `null`. For example, this option can be set to '.' to generate clean file names:

```
// apple-puree
Text::slug('apple purée');

// apple_puree
Text::slug('apple purée', '_');

// foo-bar.tar.gz
Text::slug('foo bar.tar.gz', ['preserve' => '.']);
```

[183] https://unicode-org.github.io/icu/userguide/transforms/general/#transliterator-identifiers

Generating UUIDs

static Cake\Utility\Text::**uuid**

The UUID method is used to generate unique identifiers as per **RFC 4122**[184]. The UUID is a 128-bit string in the format of `485fc381-e790-47a3-9794-1337c0a8fe68`.

```
Text::uuid(); // 485fc381-e790-47a3-9794-1337c0a8fe68
```

Simple String Parsing

static Cake\Utility\Text::**tokenize**(*$data*, *$separator = ','*, *$leftBound = '('*, *$rightBound = ')'*)

Tokenizes a string using `$separator`, ignoring any instance of `$separator` that appears between `$leftBound` and `$rightBound`.

This method can be useful when splitting up data that has regular formatting such as tag lists:

```
$data = "cakephp 'great framework' php";
$result = Text::tokenize($data, ' ', "'", "'");
// Result contains
['cakephp', "'great framework'", 'php'];
```

Cake\Utility\Text::**parseFileSize**(*string $size*, *$default*)

This method unformats a number from a human-readable byte size to an integer number of bytes:

```
$int = Text::parseFileSize('2GB');
```

Formatting Strings

static Cake\Utility\Text::**insert**(*$string*, *$data*, *$options = []*)

The insert method is used to create string templates and to allow for key/value replacements:

```
Text::insert(
    'My name is :name and I am :age years old.',
    ['name' => 'Bob', 'age' => '65']
);
// Returns: "My name is Bob and I am 65 years old."
```

static Cake\Utility\Text::**cleanInsert**(*$string*, *$options = []*)

Cleans up a `Text::insert` formatted string with given `$options` depending on the 'clean' key in `$options`. The default method used is text but html is also available. The goal of this function is to replace all whitespace and unneeded markup around placeholders that did not get replaced by `Text::insert`.

You can use the following options in the options array:

[184] https://datatracker.ietf.org/doc/html/rfc4122.html

```
$options = [
    'clean' => [
        'method' => 'text', // or html
    ],
    'before' => '',
    'after' => ''
];
```

Wrapping Text

static Cake\Utility\Text::**wrap**(*$text*, *$options = []*)

Wraps a block of text to a set width and indents blocks as well. Can intelligently wrap text so words are not sliced across lines:

```
$text = 'This is the song that never ends.';
$result = Text::wrap($text, 22);

// Returns
This is the song that
never ends.
```

You can provide an array of options that control how wrapping is done. The supported options are:

- `width` The width to wrap to. Defaults to 72.
- `wordWrap` Whether or not to wrap whole words. Defaults to `true`.
- `indent` The character to indent lines with. Defaults to ‘’.
- `indentAt` The line number to start indenting text. Defaults to 0.

static Cake\Utility\Text::**wrapBlock**(*$text*, *$options = []*)

If you need to ensure that the total width of the generated block won’t exceed a certain length even with internal indentation, you need to use `wrapBlock()` instead of `wrap()`. This is particularly useful to generate text for the console for example. It accepts the same options as `wrap()`:

```
$text = 'This is the song that never ends. This is the song that never ends.';
$result = Text::wrapBlock($text, [
    'width' => 22,
    'indent' => ' →  ',
    'indentAt' => 1
]);

// Returns
This is the song that
 →  never ends. This
 →  is the song that
 →  never ends.
```

Highlighting Substrings

Cake\Utility\Text::**highlight**(*string $haystack*, *string $needle*, *array $options = []*)

Highlights `$needle` in `$haystack` using the `$options['format']` string specified or a default string.

Options:

- `format` string - The piece of HTML with the phrase that will be highlighted
- `html` bool - If `true`, will ignore any HTML tags, ensuring that only the correct text is highlighted

Example:

```
// Called as TextHelper
echo $this->Text->highlight(
    $lastSentence,
    'using',
    ['format' => '<span class="highlight">\1</span>']
);

// Called as Text
use Cake\Utility\Text;

echo Text::highlight(
    $lastSentence,
    'using',
    ['format' => '<span class="highlight">\1</span>']
);
```

Output:

Removing Links

Cake\Utility\Text::**stripLinks**(*$text*)

Strips the supplied `$text` of any HTML links.

Truncating Text

Cake\Utility\Text::**truncate**(*string $text*, *int $length = 100*, *array $options*)

If `$text` is longer than `$length`, this method truncates it at `$length` and adds a suffix consisting of `'ellipsis'`, if defined. If `'exact'` is passed as `false`, the truncation will occur at the first whitespace after the point at which `$length` is exceeded. If `'html'` is passed as `true`, HTML tags will be respected and will not be cut off.

`$options` is used to pass all extra parameters, and has the following possible keys by default, all of which are optional:

```
[
    'ellipsis' => '...',
    'exact' => true,
    'html' => false
]
```

Example:

```
// Called as TextHelper
echo $this->Text->truncate(
    'The killer crept forward and tripped on the rug.',
    22,
    [
        'ellipsis' => '...',
        'exact' => false
    ]
);

// Called as Text
use Cake\Utility\Text;

echo Text::truncate(
    'The killer crept forward and tripped on the rug.',
    22,
    [
        'ellipsis' => '...',
        'exact' => false
    ]
);
```

Output:

```
The killer crept...
```

Truncating the Tail of a String

`Cake\Utility\Text::tail`(*string $text*, *int $length = 100*, *array $options*)

If `$text` is longer than `$length`, this method removes an initial substring with length consisting of the difference and prepends a prefix consisting of `'ellipsis'`, if defined. If `'exact'` is passed as `false`, the truncation will occur at the first whitespace prior to the point at which truncation would otherwise take place.

`$options` is used to pass all extra parameters, and has the following possible keys by default, all of which are optional:

```
[
    'ellipsis' => '...',
    'exact' => true
]
```

Example:

```
$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'

// Called as TextHelper
echo $this->Text->tail(
    $sampleText,
    70,
```

(continues on next page)

(continued from previous page)

```
    [
        'ellipsis' => '...',
        'exact' => false
    ]
);

// Called as Text
use Cake\Utility\Text;

echo Text::tail(
    $sampleText,
    70,
    [
        'ellipsis' => '...',
        'exact' => false
    ]
);
```

Output:

```
...a TV, a C# program that can divide by zero, death metal t-shirts
```

Extracting an Excerpt

Cake\Utility\Text::**excerpt**(*string $haystack*, *string $needle*, *integer $radius=100*, *string $ellipsis="..."*)

Extracts an excerpt from `$haystack` surrounding the `$needle` with a number of characters on each side determined by `$radius`, and prefix/suffix with `$ellipsis`. This method is especially handy for search results. The query string or keywords can be shown within the resulting document.

```
// Called as TextHelper
echo $this->Text->excerpt($lastParagraph, 'method', 50, '...');

// Called as Text
use Cake\Utility\Text;

echo Text::excerpt($lastParagraph, 'method', 50, '...');
```

Output:

```
... by $radius, and prefix/suffix with $ellipsis. This method is especially
handy for search results. The query...
```

Converting an Array to Sentence Form

Cake\Utility\Text::**toList**(*array $list*, *$and='and'*, *$separator=', '*)

Creates a comma-separated list where the last two items are joined with 'and':

```
$colors = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet'];

// Called as TextHelper
echo $this->Text->toList($colors);

// Called as Text
use Cake\Utility\Text;

echo Text::toList($colors);
```

Output:

```
red, orange, yellow, green, blue, indigo and violet
```

CHAPTER 39

Date & Time

class Cake\I18n**DateTime**

If you need `TimeHelper` functionalities outside of a `View`, use the `DateTime` class:

```
use Cake\I18n\DateTime;

class UsersController extends AppController
{
    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('Authentication.Authentication');
    }

    public function afterLogin()
    {
        $identity = $this->Authentication->getIdentity();
        $time = new DateTime($identity->date_of_birth);
        if ($time->isToday()) {
            // Greet user with a happy birthday message
            $this->Flash->success(__('Happy birthday to you...'));
        }
    }
}
```

Under the hood, CakePHP uses Chronos[185] to power its `DateTime` utility. Anything you can do with `Chronos` and PHP's `DateTime`, you can do with `DateTime` and `Date`.

For more details on Chronos please see the API documentation[186].

[185] https://github.com/cakephp/chronos
[186] https://api.cakephp.org/chronos/1.0/

Creating DateTime Instances

`DateTime` are immutable objects that are useful when you want to prevent accidental changes to data, or when you want to avoid order based dependency issues.

There are a few ways to create `DateTime` instances:

```
use Cake\I18n\DateTime;

// Create from a string datetime.
$time = DateTime::createFromFormat(
    'Y-m-d H:i:s',
    '2021-01-31 22:11:30',
    'America/New_York'
);

// Create from a timestamp and set timezone
$time = DateTime::createFromTimestamp(1612149090, 'America/New_York');

// Get the current time.
$time = DateTime::now();

// Or just use 'new'
$time = new DateTime('2021-01-31 22:11:30', 'America/New_York');

$time = new DateTime('2 hours ago');
```

The `DateTime` class constructor can take any parameter that the internal `DateTimeImmutable` PHP class can. When passing a number or numeric string, it will be interpreted as a UNIX timestamp.

In test cases, you can mock out `now()` using `setTestNow()`:

```
// Fixate time.
$time = new DateTime('2021-01-31 22:11:30');
DateTime::setTestNow($time);

// Outputs '2021-01-31 22:11:30'
$now = DateTime::now();
echo $now->i18nFormat('yyyy-MM-dd HH:mm:ss');

// Outputs '2021-01-31 22:11:30'
$now = DateTime::parse('now');
echo $now->i18nFormat('yyyy-MM-dd HH:mm:ss');
```

Manipulation

Remember, `DateTime` instance always return a new instance from setters instead of modifying itself:

```
$time = DateTime::now();

// Create and reassign a new instance
$newTime = $time->year(2013)
    ->month(10)
    ->day(31);
// Outputs '2013-10-31 22:11:30'
echo $newTime->i18nFormat('yyyy-MM-dd HH:mm:ss');
```

You can also use the methods provided by PHP's built-in `DateTime` class:

```
$time = $time->setDate(2013, 10, 31);
```

Failing to reassign the new `DateTime` instances will result in the original, unmodified instance being used:

```
$time->year(2013)
    ->month(10)
    ->day(31);
// Outputs '2021-01-31 22:11:30'
echo $time->i18nFormat('yyyy-MM-dd HH:mm:ss');
```

You can create another instance with modified dates, through subtraction and addition of their components:

```
$time = DateTime::create(2021, 1, 31, 22, 11, 30);
$newTime = $time->subDays(5)
    ->addHours(-2)
    ->addMonth(1);
// Outputs '2/26/21, 8:11 PM'
echo $newTime;

// Using strtotime strings.
$newTime = $time->modify('+1 month -5 days -2 hours');
// Outputs '2/26/21, 8:11 PM'
echo $newTime;
```

You can get the internal components of a date by accessing its properties:

```
$time = DateTime::create(2021, 1, 31, 22, 11, 30);
echo $time->year; // 2021
echo $time->month; // 1
echo $time->day; // 31
echo $time->timezoneName; // America/New_York
```

Formatting

static Cake\I18n\DateTime::**setJsonEncodeFormat**(*$format*)

This method sets the default format used when converting an object to json:

```
DateTime::setJsonEncodeFormat('yyyy-MM-dd HH:mm:ss');  // For any immutable DateTime
Date::setJsonEncodeFormat('yyyy-MM-dd HH:mm:ss');  // For any mutable Date

$time = DateTime::parse('2021-01-31 22:11:30');
echo json_encode($time);   // Outputs '2021-01-31 22:11:30'

Date::setJsonEncodeFormat(static function($time) {
    return $time->format(DATE_ATOM);
});
```

Note: This method must be called statically.

Note: Be aware that this is not a PHP Datetime string format! You need to use a ICU date formatting string as specified in the following resource: https://unicode-org.github.io/icu/userguide/format_parse/datetime/#datetime-format-syntax.

Changed in version 4.1.0: The `callable` parameter type was added.

Cake\I18n\DateTime::**i18nFormat**(*$format = null*, *$timezone = null*, *$locale = null*)

A very common thing to do with `Time` instances is to print out formatted dates. CakePHP makes this a snap:

```
$time = DateTime::parse('2021-01-31 22:11:30');

// Prints a localized datetime stamp. Outputs '1/31/21, 10:11 PM'
echo $time;

// Outputs '1/31/21, 10:11 PM' for the en-US locale
echo $time->i18nFormat();

// Use the full date and time format. Outputs 'Sunday, January 31, 2021 at 10:11:30 PM␣
↪Eastern Standard Time'
echo $time->i18nFormat(\IntlDateFormatter::FULL);

// Use full date but short time format. Outputs 'Sunday, January 31, 2021 at 10:11 PM'
echo $time->i18nFormat([\IntlDateFormatter::FULL, \IntlDateFormatter::SHORT]);

// Outputs '2021-Jan-31 22:11:30'
echo $time->i18nFormat('yyyy-MMM-dd HH:mm:ss');
```

It is possible to specify the desired format for the string to be displayed. You can either pass IntlDateFormatter constants[187] as the first argument of this function, or pass a full ICU date formatting string as specified in the following resource: https://unicode-org.github.io/icu/userguide/format_parse/datetime/#datetime-format-syntax.

You can also format dates with non-gregorian calendars:

[187] https://www.php.net/manual/en/class.intldateformatter.php

```
// On ICU version 66.1
$time = DateTime::create(2021, 1, 31, 22, 11, 30);

// Outputs 'Sunday, Bahman 12, 1399 AP at 10:11:30 PM Eastern Standard Time'
echo $time->i18nFormat(\IntlDateFormatter::FULL, null, 'en-IR@calendar=persian');

// Outputs 'Sunday, January 31, 3 Reiwa at 10:11:30 PM Eastern Standard Time'
echo $time->i18nFormat(\IntlDateFormatter::FULL, null, 'en-JP@calendar=japanese');

// Outputs 'Sunday, Twelfth Month 19, 2020(geng-zi) at 10:11:30 PM Eastern Standard Time'
echo $time->i18nFormat(\IntlDateFormatter::FULL, null, 'en-CN@calendar=chinese');

// Outputs 'Sunday, Jumada II 18, 1442 AH at 10:11:30 PM Eastern Standard Time'
echo $time->i18nFormat(\IntlDateFormatter::FULL, null, 'en-SA@calendar=islamic');
```

The following calendar types are supported:

- japanese
- buddhist
- chinese
- persian
- indian
- islamic
- hebrew
- coptic
- ethiopic

Note: For constant strings i.e. IntlDateFormatter::FULL Intl uses ICU library that feeds its data from CLDR (https://cldr.unicode.org/) which version may vary depending on PHP installation and give different results.

`Cake\I18n\DateTime::nice()`

Print out a predefined 'nice' format:

```
$time = DateTime::parse('2021-01-31 22:11:30', new \DateTimeZone('America/New_York'));

// Outputs 'Jan 31, 2021, 10:11 PM' in en-US
echo $time->nice();
```

You can alter the timezone in which the date is displayed without altering the `DateTime` object itself. This is useful when you store dates in one timezone, but want to display them in a user's own timezone:

```
// Outputs 'Monday, February 1, 2021 at 4:11:30 AM Central European Standard Time'
echo $time->i18nFormat(\IntlDateFormatter::FULL, 'Europe/Paris');

// Outputs 'Monday, February 1, 2021 at 12:11:30 PM Japan Standard Time'
echo $time->i18nFormat(\IntlDateFormatter::FULL, 'Asia/Tokyo');
```

(continues on next page)

(continued from previous page)

```
// Timezone is unchanged. Outputs 'America/New_York'
echo $time->timezoneName;
```

Leaving the first parameter as `null` will use the default formatting string:

```
// Outputs '2/1/21, 4:11 AM'
echo $time->i18nFormat(null, 'Europe/Paris');
```

Finally, it is possible to use a different locale for displaying a date:

```
// Outputs 'lundi 1 février 2021 à 04:11:30 heure normale d'Europe centrale'
echo $time->i18nFormat(\IntlDateFormatter::FULL, 'Europe/Paris', 'fr-FR');

// Outputs '1 févr. 2021 à 04:11'
echo $time->nice('Europe/Paris', 'fr-FR');
```

Setting the Default Locale and Format String

The default locale in which dates are displayed when using `nice` `i18nFormat` is taken from the directive intl.default_locale[188]. You can, however, modify this default at runtime:

```
DateTime::setDefaultLocale('es-ES');
Date::setDefaultLocale('es-ES');

// Outputs '31 ene. 2021 22:11'
echo $time->nice();
```

From now on, datetimes will be displayed in the Spanish preferred format unless a different locale is specified directly in the formatting method.

Likewise, it is possible to alter the default formatting string to be used for `i18nFormat`:

```
DateTime::setToStringFormat(\IntlDateFormatter::SHORT); // For any DateTime
Date::setToStringFormat(\IntlDateFormatter::SHORT); // For any Date

// The same method exists on Date, and DateTime
DateTime::setToStringFormat([
    \IntlDateFormatter::FULL,
    \IntlDateFormatter::SHORT
]);
// Outputs 'Sunday, January 31, 2021 at 10:11 PM'
echo $time;

// The same method exists on Date and DateTime
DateTime::setToStringFormat("EEEE, MMMM dd, yyyy 'at' KK:mm:ss a");
// Outputs 'Sunday, January 31, 2021 at 10:11:30 PM'
echo $time;
```

It is recommended to always use the constants instead of directly passing a date format string.

[188] https://www.php.net/manual/en/intl.configuration.php#ini.intl.default-locale

Note: Be aware that this is not a PHP Datetime string format! You need to use a ICU date formatting string as specified in the following resource: https://unicode-org.github.io/icu/userguide/format_parse/datetime/#datetime-format-syntax.

Formatting Relative Times

`Cake\I18n\DateTime::timeAgoInWords(array $options = [])`

Often it is useful to print times relative to the present:

```
$time = new DateTime('Jan 31, 2021');
// On June 12, 2021, this would output '4 months, 1 week, 6 days ago'
echo $time->timeAgoInWords(
    ['format' => 'MMM d, YYY', 'end' => '+1 year']
);
```

The `end` option lets you define at which point after which relative times should be formatted using the `format` option. The `accuracy` option lets us control what level of detail should be used for each interval range:

```
// Outputs '4 months ago'
echo $time->timeAgoInWords([
    'accuracy' => ['month' => 'month'],
    'end' => '1 year'
]);
```

By setting `accuracy` to a string, you can specify what is the maximum level of detail you want output:

```
$time = new DateTime('+23 hours');
// Outputs 'in about a day'
echo $time->timeAgoInWords([
    'accuracy' => 'day'
]);
```

Conversion

`Cake\I18n\DateTime::toQuarter()`

Once created, you can convert `DateTime` instances into timestamps or quarter values:

```
$time = new DateTime('2021-01-31');
echo $time->toQuarter();  // Outputs '1'
echo $time->toUnixString();  // Outputs '1612069200'
```

Comparing With the Present

Cake\I18n\DateTime::**isYesterday**()

Cake\I18n\DateTime::**isThisWeek**()

Cake\I18n\DateTime::**isThisMonth**()

Cake\I18n\DateTime::**isThisYear**()

You can compare a `DateTime` instance with the present in a variety of ways:

```
$time = new DateTime('+3 days');

debug($time->isYesterday());
debug($time->isThisWeek());
debug($time->isThisMonth());
debug($time->isThisYear());
```

Each of the above methods will return `true`/`false` based on whether or not the `DateTime` instance matches the present.

Comparing With Intervals

Cake\I18n\DateTime::**isWithinNext**(*$interval*)

You can see if a `DateTime` instance falls within a given range using `wasWithinLast()` and `isWithinNext()`:

```
$time = new DateTime('+3 days');

// Within 2 days. Outputs 'false'
debug($time->isWithinNext('2 days'));

// Within 2 next weeks. Outputs 'true'
debug($time->isWithinNext('2 weeks'));
```

Cake\I18n\DateTime::**wasWithinLast**(*$interval*)

You can also compare a `DateTime` instance within a range in the past:

```
$time = new DateTime('-72 hours');

// Within past 2 days. Outputs 'false'
debug($time->wasWithinLast('2 days'));

// Within past 3 days. Outputs 'true'
debug($time->wasWithinLast('3 days'));

// Within past 2 weeks. Outputs 'true'
debug($time->wasWithinLast('2 weeks'));
```

Date

The immutable `Date` class in CakePHP implements a similar API and methods as *Cake\I18n\DateTime* does. The main difference between `DateTime` and `Date` is that `Date` does not track time components. As an example:

```
use Cake\I18n\Date;

$date = new Date('2021-01-31');

$newDate = $date->modify('+2 hours');
// Outputs '2021-01-31 00:00:00'
echo $newDate->format('Y-m-d H:i:s');

$newDate = $date->addHours(36);
// Outputs '2021-01-31 00:00:00'
echo $newDate->format('Y-m-d H:i:s');

$newDate = $date->addDays(10);
// Outputs '2021-02-10 00:00:00'
echo $newDate->format('Y-m-d H:i:s');
```

Attempts to modify the timezone on a `Date` instance are also ignored:

```
use Cake\I18n\Date;
$date = new Date('2021-01-31', new \DateTimeZone('America/New_York'));
$newDate = $date->setTimezone(new \DateTimeZone('Europe/Berlin'));

// Outputs 'America/New_York'
echo $newDate->format('e');
```

Mutable Dates and Times

class Cake\I18n**Time**

class Cake\I18n**Date**

CakePHP uses mutable date and time classes that implement the same interface as their immutable siblings. Immutable objects are useful when you want to prevent accidental changes to data, or when you want to avoid order based dependency issues. Take the following code:

```
use Cake\I18n\Time;
$time = new Time('2015-06-15 08:23:45');
$time->modify('+2 hours');

// This method also modifies the $time instance
$this->someOtherFunction($time);

// Output here is unknown.
echo $time->format('Y-m-d H:i:s');
```

If the method call was re-ordered, or if `someOtherFunction` changed the output could be unexpected. The mutability of our object creates temporal coupling. If we were to use immutable objects, we could avoid this issue:

```
use Cake\I18n\DateTime;
$time = new DateTime('2015-06-15 08:23:45');
$time = $time->modify('+2 hours');

// This method's modifications don't change $time
$this->someOtherFunction($time);

// Output here is known.
echo $time->format('Y-m-d H:i:s');
```

Immutable dates and times are useful in entities as they prevent accidental modifications, and force changes to be explicit. Using immutable objects helps the ORM to more easily track changes, and ensure that date and datetime columns are persisted correctly:

```
// This change will be lost when the article is saved.
$article->updated->modify('+1 hour');

// By replacing the time object the property will be saved.
$article->updated = $article->updated->modify('+1 hour');
```

Accepting Localized Request Data

When creating text inputs that manipulate dates, you'll probably want to accept and parse localized datetime strings. See the *Parsing Localized Datetime Data*.

Supported Timezones

CakePHP supports all valid PHP timezones. For a list of supported timezones, see this page[189].

[189] https://php.net/manual/en/timezones.php

CHAPTER 40

Xml

class Cake\Utility**Xml**

The Xml class allows you to transform arrays into SimpleXMLElement or DOMDocument objects, and back into arrays again.

Loading XML documents

static Cake\Utility\Xml::**build**(*$input*, *array $options = []*)

You can load XML-ish data using `Xml::build()`. Depending on your `$options` parameter, this method will return a SimpleXMLElement (default) or DOMDocument object. You can use `Xml::build()` to build XML objects from a variety of sources. For example, you can load XML from strings:

```
$text = '<?xml version="1.0" encoding="utf-8"?>
<post>
    <id>1</id>
    <title>Best post</title>
    <body> ... </body>
</post>';
$xml = Xml::build($text);
```

You can also build Xml objects from local files by overriding the default option:

```
// Local file
$xml = Xml::build('/home/awesome/unicorns.xml', ['readFile' => true]);
```

You can also build Xml objects using an array:

```
$data = [
    'post' => [
        'id' => 1,
        'title' => 'Best post',
        'body' => ' ... ',
    ]
];
$xml = Xml::build($data);
```

If your input is invalid, the Xml class will throw an exception:

```
$xmlString = 'What is XML?';
try {
    $xmlObject = Xml::build($xmlString); // Here will throw an exception
} catch (\Cake\Utility\Exception\XmlException $e) {
    throw new InternalErrorException();
}
```

Note: DOMDocument[190] and SimpleXML[191] implement different APIs. Be sure to use the correct methods on the object you request from Xml.

Loading HTML documents

HTML documents can be parsed into `SimpleXmlElement` or `DOMDocument` objects with `loadHtml()`:

```
$html = Xml::loadHtml($htmlString, ['return' => 'domdocument']);
```

By default entity loading and huge document parsing are disabled. These modes can be enabled with the `loadEntities` and `parseHuge` options respectively.

Transforming a XML String in Array

`toArray($obj);`

Converting XML strings into arrays is simple with the Xml class as well. By default you'll get a SimpleXml object back:

```
$xmlString = '<?xml version="1.0"?><root><child>value</child></root>';
$xmlArray = Xml::toArray(Xml::build($xmlString));
```

If your XML is invalid a `Cake\Utility\Exception\XmlException` will be raised.

[190] https://php.net/domdocument
[191] https://php.net/simplexml

Transforming an Array into a String of XML

```
$xmlArray = ['root' => ['child' => 'value']];
// You can use Xml::build() too.
$xmlObject = Xml::fromArray($xmlArray, ['format' => 'tags']);
$xmlString = $xmlObject->asXML();
```

Your array must have only one element in the "top level" and it can not be numeric. If the array is not in this format, Xml will throw an exception. Examples of invalid arrays:

```
// Top level with numeric key
[
    ['key' => 'value']
];

// Multiple keys in top level
[
    'key1' => 'first value',
    'key2' => 'other value'
];
```

By default array values will be output as XML tags. If you want to define attributes or text values you can prefix the keys that are supposed to be attributes with `@`. For value text, use `@` as the key:

```
$xmlArray = [
    'project' => [
        '@id' => 1,
        'name' => 'Name of project, as tag',
        '@' => 'Value of project',
    ],
];
$xmlObject = Xml::fromArray($xmlArray);
$xmlString = $xmlObject->asXML();
```

The content of `$xmlString` will be:

```
<?xml version="1.0"?>
<project id="1">Value of project<name>Name of project, as tag</name></project>
```

Using Namespaces

To use XML Namespaces, create a key in your array with the name `xmlns:` in a generic namespace or input the prefix `xmlns:` in a custom namespace. See the samples:

```
$xmlArray = [
    'root' => [
        'xmlns:' => 'https://cakephp.org',
        'child' => 'value',
    ]
];
$xml1 = Xml::fromArray($xmlArray);
```

(continues on next page)

(continued from previous page)

```
$xmlArray(
    'root' => [
        'tag' => [
            'xmlns:pref' => 'https://cakephp.org',
            'pref:item' => [
                'item 1',
                'item 2'
            ]
        ]
    ]
);
$xml2 = Xml::fromArray($xmlArray);
```

The value of `$xml1` and `$xml2` will be, respectively:

```
<?xml version="1.0"?>
<root xmlns="https://cakephp.org"><child>value</child>

<?xml version="1.0"?>
<root><tag xmlns:pref="https://cakephp.org"><pref:item>item 1</pref:item><pref:item>item␣
↪2</pref:item></tag></root>
```

Creating a Child

After you have created your XML document, you just use the native interfaces for your document type to add, remove, or manipulate child nodes:

```
// Using SimpleXML
$myXmlOriginal = '<?xml version="1.0"?><root><child>value</child></root>';
$xml = Xml::build($myXmlOriginal);
$xml->root->addChild('young', 'new value');

// Using 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);
```

Tip: After manipulating your XML using SimpleXMLElement or DomDocument you can use `Xml::toArray()` without a problem.

CHAPTER 41

Constants & Functions

While most of your day-to-day work in CakePHP will be utilizing core classes and methods, CakePHP features a number of global convenience functions that may come in handy. Many of these functions are for use with CakePHP classes (loading model or component classes), but many others make working with arrays or strings a little easier.

We'll also cover some of the constants available in CakePHP applications. Using these constants will help make upgrades more smooth, but are also convenient ways to point to certain files or directories in your CakePHP application.

Global Functions

Here are CakePHP's globally available functions. Most of them are just convenience wrappers for other CakePHP functionality, such as debugging and translating content. By default only namespaced functions are autoloaded, however you can optionally load global aliases by adding:

```
require CAKE . 'functions.php';
```

To your application's `config/bootstrap.php`. Doing this will load global aliases for *all* functions listed below.

__(string $string_id, [$formatArgs])

This function handles localization in CakePHP applications. The `$string_id` identifies the ID for a translation. You can supply additional arguments to replace placeholders in your string:

```
__('You have {0} unread messages', $number);
```

You can also provide a name-indexed array of replacements:

```
__('You have {unread} unread messages', ['unread' => $number]);
```

Note: Check out the *Internationalization & Localization* section for more information.

__d(*string $domain*, *string $msg*, *mixed $args = null*)

Allows you to override the current domain for a single message lookup.

Useful when internationalizing a plugin: `echo __d('plugin_name', 'This is my plugin');`

Note: Make sure to use the underscored version of the plugin name here as domain.

__dn(*string $domain*, *string $singular*, *string $plural*, *integer $count*, *mixed $args = null*)

Allows you to override the current domain for a single plural message lookup. Returns correct plural form of message identified by `$singular` and `$plural` for count `$count` from domain `$domain`.

__dx(*string $domain*, *string $context*, *string $msg*, *mixed $args = null*)

Allows you to override the current domain for a single message lookup. It also allows you to specify a context.

The context is a unique identifier for the translations string that makes it unique within the same domain.

__dxn(*string $domain*, *string $context*, *string $singular*, *string $plural*, *integer $count*, *mixed $args = null*)

Allows you to override the current domain for a single plural message lookup. It also allows you to specify a context. Returns correct plural form of message identified by `$singular` and `$plural` for count `$count` from domain `$domain`. Some languages have more than one form for plural messages dependent on the count.

The context is a unique identifier for the translations string that makes it unique within the same domain.

__n(*string $singular*, *string $plural*, *integer $count*, *mixed $args = null*)

Returns correct plural form of message identified by `$singular` and `$plural` for count `$count`. Some languages have more than one form for plural messages dependent on the count.

__x(*string $context*, *string $msg*, *mixed $args = null*)

The context is a unique identifier for the translations string that makes it unique within the same domain.

__xn(*string $context*, *string $singular*, *string $plural*, *integer $count*, *mixed $args = null*)

Returns correct plural form of message identified by `$singular` and `$plural` for count `$count` from domain `$domain`. It also allows you to specify a context. Some languages have more than one form for plural messages dependent on the count.

The context is a unique identifier for the translations string that makes it unique within the same domain.

collection(*mixed $items*)

Convenience wrapper for instantiating a new *Cake\Collection\Collection* object, wrapping the passed argument. The `$items` parameter takes either a `Traversable` object or an array.

debug(*mixed $var*, *boolean $showHtml = null*, *$showFrom = true*)

If the core `$debug` variable is `true`, `$var` is printed out. If `$showHTML` is `true` or left as `null`, the data is rendered to be browser-friendly. If `$showFrom` is not set to `false`, the debug output will start with the line from which it was called. Also see *Debugging*

dd(*mixed $var*, *boolean $showHtml = null*)

It behaves like `debug()`, but execution is also halted. If the core `$debug` variable is `true`, `$var` is printed. If `$showHTML` is `true` or left as `null`, the data is rendered to be browser-friendly. Also see *Debugging*

pr(*mixed $var*)

Convenience wrapper for `print_r()`, with the addition of wrapping <pre> tags around the output.

pj(*mixed $var*)

JSON pretty print convenience function, with the addition of wrapping <pre> tags around the output.

It is meant for debugging the JSON representation of objects and arrays.

env(*string $key*, *string $default = null*)

Gets an environment variable from available sources. Used as a backup if `$_SERVER` or `$_ENV` are disabled.

This function also emulates `PHP_SELF` and `DOCUMENT_ROOT` on unsupporting servers. In fact, it's a good idea to always use `env()` instead of `$_SERVER` or `getenv()` (especially if you plan to distribute the code), since it's a full emulation wrapper.

h(*string $text*, *boolean $double = true*, *string $charset = null*)

Convenience wrapper for `htmlspecialchars()`.

pluginSplit(*string $name*, *boolean $dotAppend = false*, *string $plugin = null*)

Splits a dot syntax plugin name into its plugin and class name. If `$name` does not have a dot, then index 0 will be `null`.

Commonly used like `list($plugin, $name) = pluginSplit('Users.User');`

namespaceSplit(*string $class*)

Split the namespace from the classname.

Commonly used like `list($namespace, $className) = namespaceSplit('Cake\Core\App');`

Core Definition Constants

Most of the following constants refer to paths in your application.

constant Cake\Core**APP**

Absolute path to your application directory, including a trailing slash.

constant Cake\Core**APP_DIR**

Equals `app` or the name of your application directory.

constant Cake\Core**CACHE**

Path to the cache files directory. It can be shared between hosts in a multi-server setup.

constant Cake\Core**CAKE**

Path to the cake directory.

constant Cake\Core**CAKE_CORE_INCLUDE_PATH**

Path to the root lib directory.

constant Cake\Core**CONFIG**

Path to the config directory.

constant Cake\Core**CORE_PATH**

Path to the CakePHP directory with ending directory slash.

constant Cake\Core**DS**

Short for PHP's `DIRECTORY_SEPARATOR`, which is `/` on Linux and `\` on Windows.

constant Cake\Core**LOGS**

Path to the logs directory.

constant Cake\Core**RESOURCES**

Path to the resources directory.

constant Cake\Core**ROOT**

Path to the root directory.

constant Cake\Core**TESTS**

Path to the tests directory.

constant Cake\Core**TMP**

Path to the temporary files directory.

constant Cake\Core**WWW_ROOT**

Full path to the webroot.

Timing Definition Constants

constant Cake\Core**TIME_START**

Unix timestamp in microseconds as a float from when the application started.

CHAPTER 42

Chronos

This page has moved[192].

[192] https://book.cakephp.org/chronos/2.x/en/

CHAPTER 43

Debug Kit

This page has moved[193].

[193] https://book.cakephp.org/debugkit/5.x/en/

CHAPTER 44

Migrations

This page has moved[194].

[194] https://book.cakephp.org/migrations/4/

CHAPTER 45

ElasticSearch

This page has moved[195].

[195] https://book.cakephp.org/elasticsearch/3/en/

CHAPTER 46

Appendices

Appendices contain information regarding the new features introduced in each version and the migration path between versions.

5.x Migration Guide

Backwards Compatibility Shimming

If you need/want to shim 4.x behavior, or partially migrate in steps, check out the Shim plugin[196] that can help mitigate some BC breaking changes.

Forwards Compatibility Shimming

Forwards compatibility shimming can prepare your 4.x app for the next major release (5.x).

If you already want to shim 5.x behavior into 4.x, check out the Shim plugin[197]. This plugin aims to mitigate some backwards compatibility breakage and help backport features from 5.x to 4.x. The closer your 3.x app is to 4.x, the smaller will be the diff of changes, and the smoother will be the final upgrade.

[196] https://github.com/dereuromark/cakephp-shim
[197] https://github.com/dereuromark/cakephp-shim

General Information

CakePHP Development Process

CakePHP projects broadly follow semver[198]. This means that:

- Releases are numbered in the form of **A.B.C**
- **A** releases are *major releases*. They contain breaking changes and will require non-trivial amounts of work to upgrade to from a lower **A** release.
- **A.B** releases are *feature releases*. Each version will be backwards compatible but may introduce new deprecations. If a breaking change is absolutely required it will be noted in the migration guide for that release.
- **A.B.C** releases are *patch* releases. They should be backwards compatible with the previous patch release. The exception to this rule is if a security issue is discovered and the only solution is to break an existing API.

See the *Backwards Compatibility Guide* for what we consider to be backwards compatible and a breaking changes.

Major Releases

Major releases introduce new features and can remove functionality deprecated in an earlier release. These releases live in `next` branches that match their version number such as `5.next`. Once released they are promoted into `master` and then `5.next` branch is used for future feature releases.

Feature Releases

Feature releases are where new features or extensions to existing features are shipped. Each release series receiving updates will have a `next` branch. For example `4.next`. If you would like to contribute a new feature please target these branches.

Patch Releases

Patch releases fix bugs in existing code/documentation and should always be compatible with earlier patch releases from the same feature release. These releases are created from the stable branches. Stable branches are often named after the release series such as `3.x`.

Release Cadence

- *Major Releases* are delivered approximately every two to three years. This timeframe forces us to be deliberate and considerate with our breaking changes and gives time for the community to keep up without feeling like they are being left behind.
- *Feature Releases* are delivered every five to eight months.
- *Patch Releases* Are initially delivered every two weeks. As a feature release matures this cadence relaxes to a monthly schedule.

[198] https://semver.org/

Deprecation Policy

Before a feature can be removed in a major release it needs to be deprecated. When a behavior is deprecated in release **A.x** it will continue to work for remainder of all **A.x** releases. Deprecations are generally indicated via PHP warnings. You can enable deprecation warnings by adding `E_USER_DEPRECATED` to your application's `Error.level` value.

Once deprecated behavior is not removed until the next major release. For example behavior deprecated in `4.1` will be removed in `5.0`.

Glossary

CDN
: Content Delivery Network. A 3rd party vendor you can pay to help distribute your content to data centers around the world. This helps put your static assets closer to geographically distributed users.

columns
: Used in the ORM when referring to the table columns in an database table.

CSRF
: Cross Site Request Forgery. Prevents replay attacks, double submissions and forged requests from other domains.

DI Container
: In `Application::services()` you can configure application services and their dependencies. Application services are automatically injected into Controller actions, and Command Constructors. See /development/dependency-injection.

DSN
: Data Source Name. A connection string format that is formed like a URI. CakePHP supports DSNs for Cache, Database, Log and Email connections.

dot notation
: Dot notation defines an array path, by separating nested levels with `.` For example:

```
Cache.default.engine
```

Would point to the following value:

```
[
    'Cache' => [
        'default' => [
            'engine' => 'File'
        ]
    ]
]
```

DRY
: Don't repeat yourself. Is a principle of software development aimed at reducing repetition of information of all kinds. In CakePHP DRY is used to allow you to code things once and re-use them across your application.

fields
: A generic term used to describe both entity properties, or database columns. Often used in conjunction with the FormHelper.

HTML attributes
: An array of key => values that are composed into HTML attributes. For example:

```
// Given
['class' => 'my-class', 'target' => '_blank']

// Would generate
class="my-class" target="_blank"
```

If an option can be minimized or accepts its name as the value, then `true` can be used:

```
// Given
['checked' => true]

// Would generate
checked="checked"
```

PaaS
Platform as a Service. Platform as a Service providers will provide cloud based hosting, database and caching resources. Some popular providers include Heroku, EngineYard and PagodaBox

properties
Used when referencing columns mapped onto an ORM entity.

plugin syntax
Plugin syntax refers to the dot separated class name indicating classes are part of a plugin:

```
// The plugin is "DebugKit", and the class name is "Toolbar".
'DebugKit.Toolbar'

// The plugin is "AcmeCorp/Tools", and the class name is "Toolbar".
'AcmeCorp/Tools.Toolbar'
```

routes.php
A file in `config` directory that contains routing configuration. This file is included before each request is processed. It should connect all the routes your application needs so requests can be routed to the correct controller + action.

routing array
An array of attributes that are passed to `Router::url()`. They typically look like:

```
['controller' => 'Posts', 'action' => 'view', 5]
```

PHP Namespace Index

C

Index

Symbols

A

B

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z