CakePHP Cookbook Documentation

Release 3.5

Cake Software Foundation

Aug 22, 2017

Contents

1 CakePHP at a Glance **1**
- Conventions Over Configuration . . . 1
- The Model Layer . . . 1
- The View Layer . . . 2
- The Controller Layer . . . 2
- CakePHP Request Cycle . . . 3
- Just the Start . . . 4
- Additional Reading . . . 4

2 Quick Start Guide **11**
- Bookmarker Tutorial . . . 11
- Bookmarker Tutorial Part 2 . . . 18

3 3.0 Migration Guide **27**
- Requirements . . . 27
- Upgrade Tool . . . 27
- Application Directory Layout . . . 27
- CakePHP should be installed with Composer . . . 28
- Namespaces . . . 28
- Removed Constants . . . 28
- Configuration . . . 28
- New ORM . . . 28
- Basics . . . 29
- Debugging . . . 29
- Object settings/configuration . . . 29
- Cache . . . 29
- Core . . . 30
- Console . . . 31
- Shell / Task . . . 31
- Event . . . 32
- Log . . . 32
- Routing . . . 33
- Network . . . 34
- Sessions . . . 35

Network\Http . . . 35
Network\Email . . . 36
Controller . . . 36
Controller\Components . . . 37
Model . . . 39
TestSuite . . . 40
View . . . 41
View\Helper . . . 42
I18n . . . 46
L10n . . . 47
Testing . . . 47
Utility . . . 48

4 Tutorials & Examples **51**
Bookmarker Tutorial . . . 51
Bookmarker Tutorial Part 2 . . . 58
Blog Tutorial . . . 65
Blog Tutorial - Part 2 . . . 68
Blog Tutorial - Part 3 . . . 78
Blog Tutorial - Authentication and Authorization . . . 85
Content Management Tutorial . . . 92
CMS Tutorial - Creating the Database . . . 93
CMS Tutorial - Creating the Articles Controller . . . 96
CMS Tutorial - Tags and Users . . . 105
CMS Tutorial - Authentication . . . 112

5 Contributing **119**
Documentation . . . 119
Tickets . . . 126
Code . . . 127
Coding Standards . . . 130
Backwards Compatibility Guide . . . 140

6 Installation **143**
Requirements . . . 143
Installing CakePHP . . . 144
Permissions . . . 145
Development Server . . . 146
Production . . . 146
Fire It Up . . . 147
URL Rewriting . . . 147

7 Configuration **153**
Configuring your Application . . . 153
Additional Class Paths . . . 155
Inflection Configuration . . . 156
Environment Variables . . . 156
Configure Class . . . 157
Reading and writing configuration files . . . 158
Bootstrapping CakePHP . . . 160
Disabling Generic Tables . . . 161

8 Routing **163**
Quick Tour . . . 163
Connecting Routes . . . 165

Creating RESTful Routes . . . 178
Passed Arguments . . . 181
Generating URLs . . . 182
Redirect Routing . . . 183
Custom Route Classes . . . 183
Creating Persistent URL Parameters . . . 185
Handling Named Parameters in URLs . . . 185

9 Request & Response Objects **191**
Request . . . 191
Response . . . 198
Setting Cookies . . . 205
Setting Cross Origin Request Headers (CORS) . . . 205
Common Mistakes with Immutable Responses . . . 206
Cookie Collections . . . 206

10 Controllers **209**
The App Controller . . . 209
Request Flow . . . 210
Controller Actions . . . 210
Interacting with Views . . . 211
Redirecting to Other Pages . . . 213
Loading Additional Models . . . 214
Paginating a Model . . . 215
Configuring Components to Load . . . 215
Configuring Helpers to Load . . . 216
Request Life-cycle Callbacks . . . 216
More on Controllers . . . 217

11 Views **259**
The App View . . . 259
View Templates . . . 260
Using View Blocks . . . 263
Layouts . . . 265
Elements . . . 267
View Events . . . 270
Creating Your Own View Classes . . . 270
More About Views . . . 271

12 Database Access & ORM **369**
Quick Example . . . 369
More Information . . . 371

13 Bake Console **523**
Installation . . . 523

14 Caching **533**
Configuring Cache Class . . . 534
Writing to a Cache . . . 536
Reading From a Cache . . . 537
Deleting From a Cache . . . 537
Clearing Cached Data . . . 538
Using Cache to Store Counters . . . 538
Using Cache to Store Common Query Results . . . 539
Using Groups . . . 539

Globally Enable or Disable Cache . . . 540
Creating a Storage Engine for Cache . . . 540

15 Console Tools, Shells & Tasks **543**
The CakePHP Console . . . 543
Creating a Shell . . . 545
Shell Tasks . . . 546
Using Models in Your Shells . . . 547
Shell Helpers . . . 548
Invoking Other Shells from Your Shell . . . 548
Getting User Input . . . 549
Creating Files . . . 549
Console Output . . . 549
Stopping Shell Execution . . . 552
Hook Methods . . . 553
Configuring Options and Generating Help . . . 553
Renaming Commands . . . 562
Routing in Shells / CLI . . . 563
More Topics . . . 563

16 Debugging **575**
Basic Debugging . . . 575
Using the Debugger Class . . . 576
Outputting Values . . . 576
Logging With Stack Traces . . . 577
Generating Stack Traces . . . 577
Getting an Excerpt From a File . . . 577
Using Logging to Debug . . . 578
Debug Kit . . . 578

17 Deployment **579**
Moving files . . . 579
Adjust config/app.php . . . 579
Check Your Security . . . 580
Set Document Root . . . 580
Improve Your Application's Performance . . . 580
Deploying an update . . . 581

18 Email **583**
Basic Usage . . . 583
Configuration . . . 584
Setting Headers . . . 587
Sending Templated Emails . . . 587
Sending Attachments . . . 588
Using Transports . . . 589
Sending Messages Quickly . . . 590
Sending Emails from CLI . . . 590
Creating Reusable Emails . . . 591

19 Error & Exception Handling **593**
Error & Exception Configuration . . . 593
Creating your Own Error Handler . . . 594
Changing Fatal Error Behavior . . . 594
Exception Classes . . . 595
Built in Exceptions for CakePHP . . . 595

Using HTTP Exceptions in your Controllers . . . 598
Exception Renderer . . . 598
Creating your own Application Exceptions . . . 598
Extending and Implementing your own Exception Handlers . . . 599

20 Events System **603**
Example Event Usage . . . 603
Accessing Event Managers . . . 604
Core Events . . . 605
Registering Listeners . . . 605
Dispatching Events . . . 608
Additional Reading . . . 611

21 Internationalization & Localization **613**
Setting Up Translations . . . 613
Using Translation Functions . . . 615
Creating Your Own Translators . . . 619
Localizing Dates and Numbers . . . 622
Automatically Choosing the Locale Based on Request Data . . . 622

22 Logging **625**
Logging Configuration . . . 625
Error and Exception Logging . . . 627
Interacting with Log Streams . . . 627
Using the FileLog Adapter . . . 627
Logging to Syslog . . . 628
Writing to Logs . . . 628
Log API . . . 630
Logging Trait . . . 631
Using Monolog . . . 631

23 Modelless Forms **633**
Creating a Form . . . 633
Processing Request Data . . . 634
Setting Form Values . . . 635
Getting Form Errors . . . 635
Invalidating Individual Form Fields from Controller . . . 635
Creating HTML with FormHelper . . . 636

24 Plugins **637**
Installing a Plugin With Composer . . . 637
Loading a Plugin . . . 638
Plugin Configuration . . . 639
Using Plugins . . . 640
Creating Your Own Plugins . . . 640
Plugin Routes . . . 641
Plugin Controllers . . . 642
Plugin Models . . . 643
Plugin Views . . . 644
Plugin Assets . . . 645
Components, Helpers and Behaviors . . . 646
Expand Your Plugin . . . 646
Publish Your Plugin . . . 647

25 REST **649**

The Simple Setup . . . 649
Accepting Input in Other Formats . . . 651
RESTful Routing . . . 651

26 Security **653**
Security . . . 653

27 Sessions **657**
Session Configuration . . . 657
Built-in Session Handlers & Configuration . . . 658
Setting ini directives . . . 660
Creating a Custom Session Handler . . . 660
Accessing the Session Object . . . 662
Reading & Writing Session Data . . . 662
Destroying the Session . . . 663
Rotating Session Identifiers . . . 663
Flash Messages . . . 663

28 Testing **665**
Installing PHPUnit . . . 665
Test Database Setup . . . 666
Checking the Test Setup . . . 666
Test Case Conventions . . . 667
Creating Your First Test Case . . . 667
Running Tests . . . 669
Test Case Lifecycle Callbacks . . . 670
Fixtures . . . 671
Testing Table Classes . . . 676
Controller Integration Testing . . . 677
Console Integration Testing . . . 686
Testing Views . . . 690
Testing Components . . . 691
Testing Helpers . . . 692
Testing Events . . . 694
Creating Test Suites . . . 695
Creating Tests for Plugins . . . 695
Generating Tests with Bake . . . 697
Integration with Jenkins . . . 697

29 Validation **699**
Creating Validators . . . 699
Validating Data . . . 706
Validating Entities . . . 706
Core Validation Rules . . . 707

30 App Class **709**
Finding Classes . . . 709
Finding Paths to Namespaces . . . 709
Locating Plugins . . . 710
Locating Themes . . . 710
Loading Vendor Files . . . 710

31 Collections **713**
Quick Example . . . 713
List of Methods . . . 714

Iterating . . . 714
Filtering . . . 718
Aggregation . . . 719
Sorting . . . 723
Working with Tree Data . . . 724
Other Methods . . . 725

32 Folder & File **733**
Basic Usage . . . 733
Folder API . . . 733
File API . . . 737

33 Hash **741**
Hash Path Syntax . . . 741

34 Http Client **757**
Doing Requests . . . 757
Creating Multipart Requests with Files . . . 758
Sending Request Bodies . . . 759
Request Method Options . . . 759
Authentication . . . 760
Creating Scoped Clients . . . 761
Setting and Managing Cookies . . . 762
Response Objects . . . 763

35 Inflector **767**
Summary of Inflector Methods and Their Output . . . 767
Creating Plural & Singular Forms . . . 768
Creating CamelCase and under_scored Forms . . . 768
Creating Human Readable Forms . . . 768
Creating Table and Class Name Forms . . . 768
Creating Variable Names . . . 769
Creating URL Safe Strings . . . 769
Inflection Configuration . . . 769

36 Number **771**
Formatting Currency Values . . . 771
Setting the Default Currency . . . 772
Formatting Floating Point Numbers . . . 772
Formatting Percentages . . . 773
Interacting with Human Readable Values . . . 773
Formatting Numbers . . . 773
Format Differences . . . 775
Configure formatters . . . 775

37 Registry Objects **777**
Loading Objects . . . 777
Triggering Callbacks . . . 777
Disabling Callbacks . . . 778

38 Text **779**
Convert Strings into ASCII . . . 779
Creating URL Safe Strings . . . 780
Generating UUIDs . . . 780
Simple String Parsing . . . 781

Formatting Strings . . . 781
Wrapping Text . . . 781
Highlighting Substrings . . . 782
Removing Links . . . 783
Truncating Text . . . 783
Truncating the Tail of a String . . . 784
Extracting an Excerpt . . . 785
Converting an Array to Sentence Form . . . 785

39 Date & Time **787**
Creating Time Instances . . . 788
Manipulation . . . 788
Formatting . . . 789
Conversion . . . 792
Comparing With the Present . . . 792
Comparing With Intervals . . . 792
Dates . . . 793
Immutable Dates and Times . . . 793
Accepting Localized Request Data . . . 794
Supported Timezones . . . 794

40 Xml **795**
Importing Data to Xml Class . . . 795
Transforming a XML String in Array . . . 796
Transforming an Array into a String of XML . . . 796

41 Constants & Functions **799**
Global Functions . . . 799
Core Definition Constants . . . 801
Timing Definition Constants . . . 801

42 Debug Kit **803**
Installation . . . 803
DebugKit Storage . . . 803
Toolbar Usage . . . 804
Using the History Panel . . . 804
Using The Mail Panel . . . 805
Developing Your Own Panels . . . 805

43 Migrations **809**
Installation . . . 809
Overview . . . 810
Creating Migrations . . . 811
Generating migrations from an existing database . . . 816
Generating a diff between two database states . . . 816
The commands . . . 817
Using Migrations In Plugins . . . 821
Running Migrations in a non-shell environment . . . 821
Tips and tricks . . . 822

44 Appendices **827**
3.x Migration Guide . . . 827
General Information . . . 863

PHP Namespace Index **867**

Index 869

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\TableRegistry;

$users = TableRegistry::get('Users');
$query = $users->find();
foreach ($query 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\TableRegistry;

$users = TableRegistry::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', ['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.

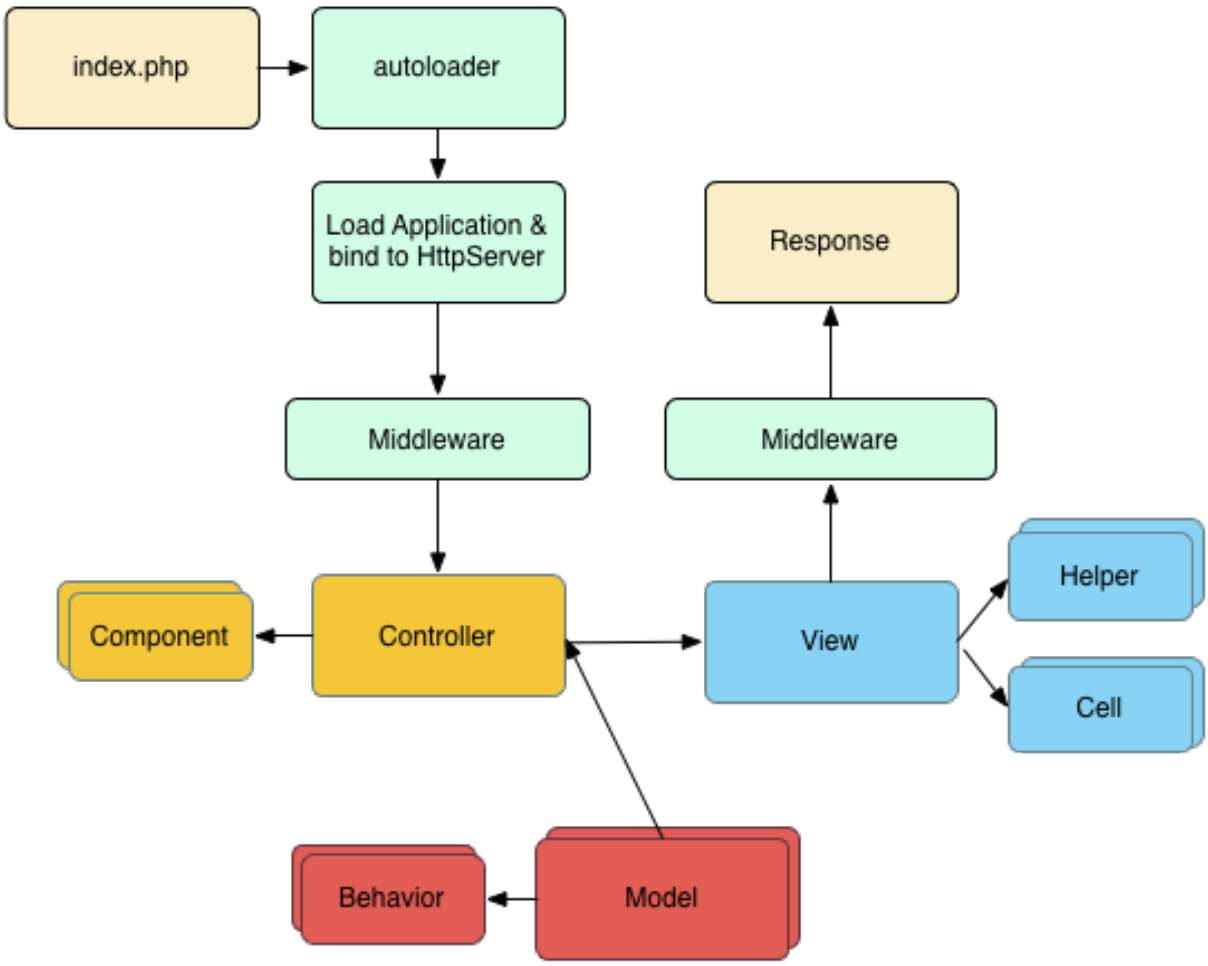
A controller can be seen as a manager that ensures that all resources needed for completing a task are delegated to the correct workers. It waits for petitions from clients, checks their validity according to authentication or authorization rules, delegates data fetching or processing to the model, selects 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 = $this->Users->newEntity();
    if ($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*.

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

Brazilian Portuguese

- Brazilian CakePHP Community[7]

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

Danish

- Danish CakePHP Slack Channel[8]

French

- French CakePHP Community[9]

German

- German CakePHP Slack Channel[10]
- German CakePHP Facebook Group[11]

Iranian

- Iranian CakePHP Community[12]

Dutch

- Dutch CakePHP Slack Channel[13]

Japanese

- Japanese CakePHP Slack Channel[14]
- Japanese CakePHP Facebook Group[15]

Portuguese

- Portuguese CakePHP Google Group[16]

Spanish

- Spanish CakePHP Slack Channel[17]
- Spanish CakePHP IRC Channel
- Spanish CakePHP Google Group[18]

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

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.

Controller Conventions

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

Public methods on Controllers are often exposed as 'actions' accessible through a web browser. For example the `/users/view` maps to the `view()` method of the `UsersController` out of the box. 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 `/article-categories/view-all` is the correct form to access the `ArticleCategoriesController::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-0 or PSR-4 standards 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.

Model and Database Conventions

Table class names are plural, CamelCased and end in `Table`. `UsersTable`, `ArticleCategoriesTable`, and `UserFavoritePagesTable` are all examples of conventional model names.

Table names corresponding to CakePHP models are plural and underscored. The underlying tables for the above mentioned models would be `users`, `article_categories`, and `user_favorite_pages`, respectively.

The convention is to use English words for table and column names. If you use words in another language, CakePHP might not be able to process the right inflections (from singular to plural and vice-versa). If you need to add your own language rules for some words, you can use the utility class *`Cake\Utility\Inflector`*. Besides defining those custom inflection rules, this class also allows you to check that CakePHP understands your custom syntax for plurals and singulars words. See the documentation about *Inflector* for more information.

Field names with two or more words are underscored: `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 `article_categories` whose name contains multiple words, the foreign key would be `article_category_id`.

Join tables, used in BelongsToMany relationships between models, should be named after the model tables they will join, arranged in alphabetical order (`articles_tags` rather than `tags_articles`).

In addition to use an auto-increment key as the primary key, you may also use UUID columns. CakePHP will create a unique 36 character UUID (*`Cake\Utility\Text::uuid()`*) whenever you save a new record using the `Table::save()` method.

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

The basic pattern is **src/Template/Controller/underscored_function_name.ctp**.

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

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

Now that you've been introduced to CakePHP's fundamentals, you might try a run through the *Bookmarker 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 (few) *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 files will be placed.
- 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. 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 publically 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.

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*.

Controller Contains your application's controllers and their components.

Locale Stores string files for internationalization.

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

Shell Contains the console commands and console tasks for your application. For more information see *Console Tools, Shells & Tasks*.

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

Template Presentational files are placed here: elements, error pages, layouts, and view template files.

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 bookmarking application.

Bookmarker Tutorial

This tutorial will walk you through the creation of a simple bookmarking application (bookmarker). To start with, we'll be installing CakePHP, creating our database, and using the tools CakePHP provides to get our application up fast.

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: CakePHP will be taking the reins from there. 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 have got an up to date PHP version:

```
php -v
```

You should at least have got installed PHP 5.6.0 (CLI) or higher. Your webserver's PHP version must also be of 5.6.0 or higher, and should best be the same version your command line interface (CLI) PHP version is of. If you'd like to see the completed application, checkout cakephp/bookmarker[19]. Let's get started!

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, it's as easy as running the following:

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

[19] https://github.com/cakephp/bookmarker-tutorial

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 **bookmarker** directory:

```
php composer.phar create-project --prefer-dist cakephp/app bookmarker
```

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\cakephp3):

```
composer self-update && composer create-project --prefer-dist cakephp/app bookmarker
```

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 something like the following:

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

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:

```
bin/cake server
```

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

[20] https://getcomposer.org/download/
[21] https://getcomposer.org/Composer-Setup.exe

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 checkmarks other than CakePHP being able to connect to your database. If not, you may need to install additional PHP extensions, or set directory permissions.

Creating the Database

Next, let's set up the database for our bookmarking application. If you haven't already done so, create an empty database for use in this tutorial, with a name of your choice, e.g. `cake_bookmarks`. You can execute the following SQL to create the necessary tables:

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

CREATE TABLE bookmarks (
    id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT NOT NULL,
    title VARCHAR(50),
    description TEXT,
    url TEXT,
    created DATETIME,
    modified DATETIME,
    FOREIGN KEY user_key (user_id) REFERENCES users(id)
);

CREATE TABLE tags (
    id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(255),
    created DATETIME,
    modified DATETIME,
    UNIQUE KEY (title)
);

CREATE TABLE bookmarks_tags (
    bookmark_id INT NOT NULL,
    tag_id INT NOT NULL,
    PRIMARY KEY (bookmark_id, tag_id),
    FOREIGN KEY tag_key(tag_id) REFERENCES tags(id),
    FOREIGN KEY bookmark_key(bookmark_id) REFERENCES bookmarks(id)
);
```

You may have noticed that the `bookmarks_tags` table used a composite primary key. CakePHP supports composite primary keys almost everywhere, making it easier to build multi-tenanted applications.

The table and column names we used were not arbitrary. By using CakePHP's *naming conventions*, we can leverage CakePHP better and avoid having to configure the framework. CakePHP is flexible enough to accommodate even inconsistent legacy database schemas, but adhering to the conventions will save you time.

Database Configuration

Next, let's tell CakePHP where our database is and how to connect to it. For many, this will be the first and last time you will need to configure anything.

The configuration should be pretty straightforward: just replace the values in the `Datasources.default` array in the **config/app.php** file with those that apply to your setup. A sample completed configuration array might look something like the following:

```
return [
    // More configuration above.
    'Datasources' => [
        'default' => [
            'className' => 'Cake\Database\Connection',
            'driver' => 'Cake\Database\Driver\Mysql',
            'persistent' => false,
            'host' => 'localhost',
            'username' => 'cakephp',
            'password' => 'AngelF00dC4k3~',
            'database' => 'cake_bookmarks',
            'encoding' => 'utf8',
            'timezone' => 'UTC',
            'cacheMetadata' => true,
        ],
    ],
    // More configuration below.
];
```

Once you've saved your **config/app.php** file, you should see that 'CakePHP is able to connect to the database' section have a checkmark.

Note: A copy of CakePHP's default configuration file is found in **config/app.default.php**.

Generating Scaffold Code

Because our database is following the CakePHP conventions, we can use the *bake console* application to quickly generate a basic application. In your command line run the following commands:

```
// On Windows you'll need to use bin\cake instead.
bin/cake bake all users
bin/cake bake all bookmarks
bin/cake bake all tags
```

This will generate the controllers, models, views, their corresponding test cases, and fixtures for our users, bookmarks and tags resources. If you've stopped your server, restart it and go to **http://localhost:8765/bookmarks**.

You should see a basic but functional application providing data access to your application's database tables. Once you're at the list of bookmarks, add a few users, bookmarks, and tags.

Note: If you see a Not Found (404) page, confirm that the Apache mod_rewrite module is loaded.

Adding Password Hashing

When you created your users (by visiting **http://localhost:8765/users**), you probably noticed that the passwords were stored in plain text. This is pretty bad from a security point of view, so let's get that fixed.

This is also a good time to talk about the model layer in CakePHP. In CakePHP, we separate the methods that operate on a collection of objects, and a single object into different classes. 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 convention based setter methods 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:

```
namespace App\Model\Entity;

use Cake\Auth\DefaultPasswordHasher; //include this line
use Cake\ORM\Entity;

class User extends Entity
{

    // Code from bake.

    protected function _setPassword($value)
    {
        $hasher = new DefaultPasswordHasher();
        return $hasher->hash($value);
    }
}
```

Now update one of the users you created earlier, if you change their password, you should see a hashed password instead of the original value on the list or view pages. CakePHP hashes passwords with bcrypt[22] by default. You can also use sha1 or md5 if you're working with an existing database.

Note: If the password doesn't get hashed, make sure you followed the same case for the password member of the class while naming the setter function

---

Getting Bookmarks with a Specific Tag

Now that we're storing passwords safely, we can build out some more interesting features in our application. Once you've amassed a collection of bookmarks, it is helpful to be able to search through them by tag. Next we'll implement a route, controller action, and finder method to search through bookmarks by tag.

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

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

Router::defaultRouteClass(DashedRoute::class);

// New route we're adding for our tagged action.
// The trailing `*` tells CakePHP that this action has
// passed parameters.
Router::scope(
```

[22] http://codahale.com/how-to-safely-store-a-password/

```
    '/bookmarks',
    ['controller' => 'Bookmarks'],
    function ($routes) {
        $routes->connect('/tagged/*', ['action' => 'tags']);
    }
);

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

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

The above defines a new 'route' which connects the **/bookmarks/tagged/** path, to `BookmarksController::tags()`. By defining routes, you can isolate how your URLs look, from how they are implemented. If we were to visit **http://localhost:8765/bookmarks/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/BookmarksController.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 BookmarksTable to find tagged bookmarks.
    $bookmarks = $this->Bookmarks->find('tagged', [
        'tags' => $tags
    ]);

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

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

Creating the Finder Method

In CakePHP we like to keep our controller actions slim, and put most of our application's logic in the models. If you were to visit the **/bookmarks/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/BookmarksTable.php** add the following:

```
// 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(Query $query, array $options)
{
    $bookmarks = $this->find()
        ->select(['id', 'url', 'title', 'description']);

    if (empty($options['tags'])) {
        $bookmarks
            ->leftJoinWith('Tags')
            ->where(['Tags.title IS' => null]);
    } else {
        $bookmarks
            ->innerJoinWith('Tags')
            ->where(['Tags.title IN ' => $options['tags']]);
    }

    return $bookmarks->group(['Bookmarks.id']);
}
```

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 `innerJoinWith()`, `where()` and `group()` methods which allow us to find distinct bookmarks that have a matching tag. When no tags are provided we use a `leftJoinWith()` and modify the 'where' condition, finding bookmarks without tags.

Creating the View

Now if you visit the **/bookmarks/tagged** URL, CakePHP will show an error letting you know that you have not made a view file. Next, let's build the view file for our `tags()` action. In **src/Template/Bookmarks/tags.ctp** put the following content:

```
<h1>
    Bookmarks tagged with
    <?= $this->Text->toList(h($tags)) ?>
</h1>

<section>
<?php foreach ($bookmarks as $bookmark): ?>
    <article>
        <!-- Use the HtmlHelper to create a link -->
        <h4><?= $this->Html->link($bookmark->title, $bookmark->url) ?></h4>
        <small><?= h($bookmark->url) ?></small>

        <!-- Use the TextHelper to format text -->
        <?= $this->Text->autoParagraph(h($bookmark->description)) ?>
    </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 user data to prevent HTML injection issues.

The **tags.ctp** 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 `$tags` and `$bookmarks` variables in our view. 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 templates as local variables.

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

So far, we've created a basic application to manage bookmarks, tags and users. However, everyone can see everyone else's tags. In the next chapter, we'll implement authentication and restrict the visible bookmarks to only those that belong to the current user.

Now continue to *Bookmarker Tutorial Part 2* to continue building your application or dive into the documentation to learn more about what CakePHP can do for you.

Bookmarker Tutorial Part 2

After finishing *the first part of this tutorial* you should have a very basic bookmarking application. In this chapter we'll be adding authentication and restricting the bookmarks each user can see/modify to only the ones they own.

Adding Login

In CakePHP, authentication is handled by *Components*. Components can be thought of as ways to create reusable chunks of controller code related to a specific feature or concept. Components can also hook into the controller's event life-cycle and interact with your application that way. To get started, we'll add the *AuthComponent* to our application. We'll pretty much want every method to require authentication, so we'll add AuthComponent in our AppController:

```
// In src/Controller/AppController.php
namespace App\Controller;

use Cake\Controller\Controller;

class AppController extends Controller
{
    public function initialize()
    {
        $this->loadComponent('Flash');
        $this->loadComponent('Auth', [
            'authenticate' => [
                'Form' => [
                    'fields' => [
                        'username' => 'email',
                        'password' => 'password'
                    ]
                ]
            ],
            'loginAction' => [
                'controller' => 'Users',
                'action' => 'login'
            ],
            'unauthorizedRedirect' => $this->referer() // If unauthorized, return
↪them to page they were just on
        ]);

        // Allow the display action so our pages controller
        // continues to work.
        $this->Auth->allow(['display']);
```

```
    }
}
```

We've just told CakePHP that we want to load the `Flash` and `Auth` components. In addition, we've customized the configuration of AuthComponent, as our users table uses `email` as the username. Now, if you go to any URL you'll be kicked to **/users/login**, which will show an error page as we have not written that code yet. So let's create the login action:

```
// In src/Controller/UsersController.php
public function login()
{
    if ($this->request->is('post')) {
        $user = $this->Auth->identify();
        if ($user) {
            $this->Auth->setUser($user);
            return $this->redirect($this->Auth->redirectUrl());
        }
        $this->Flash->error('Your username or password is incorrect.');
    }
}
```

And in **src/Template/Users/login.ctp** add the following:

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

Note: The `control()` method is available since 3.4. For prior versions you can use the `input()` method instead.

Now that we have a simple login form, we should be able to log in with one of the users that has a hashed password.

Note: If none of your users have hashed passwords, comment the `loadComponent('Auth')` line. Then go and edit the user, saving a new password for them.

You should now be able to log in. If not, make sure you are using a user that has a hashed password.

Adding Logout

Now that people can log in, you'll probably want to provide a way to log out as well. Again, in the `UsersController`, add the following code:

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

public function logout()
{
    $this->Flash->success('You are now logged out.');
```

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

This code whitelists the `logout` action as a public action, and implements the logout method. Now you can visit `/users/logout` to log out. You should then be sent to the login page.

Enabling Registrations

If you aren't logged in and you try to visit **/users/add** you will be kicked to the login page. We should fix that as we want to allow people to sign up for our application. In the `UsersController` add the following:

```
public function initialize()
{
    parent::initialize();
    // Add the 'add' action to the allowed actions list.
    $this->Auth->allow(['logout', 'add']);
}
```

The above tells `AuthComponent` that the `add()` action does *not* require authentication or authorization. You may want to take the time to clean up the **Users/add.ctp** 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 so they will not work as `AuthComponent` will deny you access to those controller actions.

Restricting Bookmark Access

Now that users can log in, we'll want to limit the bookmarks they can see to the ones they made. We'll do this using an 'authorization' adapter. Since our requirements are pretty simple, we can write some simple code in our `BookmarksController`. But before we do that, we'll want to tell the AuthComponent how our application is going to authorize actions. In your `AppController` add the following:

```
public function isAuthorized($user)
{
    return false;
}
```

Also, add the following to the configuration for `Auth` in your `AppController`:

```
'authorize' => 'Controller',
```

Your `initialize()` method should now look like:

```
public function initialize()
{
    $this->loadComponent('Flash');
    $this->loadComponent('Auth', [
        'authorize'=> 'Controller',//added this line
        'authenticate' => [
            'Form' => [
                'fields' => [
                    'username' => 'email',
                    'password' => 'password'
                ]
            ]
        ],
```

```
        'loginAction' => [
            'controller' => 'Users',
            'action' => 'login'
        ],
        'unauthorizedRedirect' => $this->referer()
    ]);

    // Allow the display action so our pages controller
    // continues to work.
    $this->Auth->allow(['display']);
}
```

We'll default to denying access, and incrementally grant access where it makes sense. First, we'll add the authorization logic for bookmarks. In your `BookmarksController` add the following:

```
public function isAuthorized($user)
{
    $action = $this->request->getParam('action');

    // The add and index actions are always allowed.
    if (in_array($action, ['index', 'add', 'tags'])) {
        return true;
    }
    // All other actions require an id.
    if (!$this->request->getParam('pass.0')) {
        return false;
    }

    // Check that the bookmark belongs to the current user.
    $id = $this->request->getParam('pass.0');
    $bookmark = $this->Bookmarks->get($id);
    if ($bookmark->user_id == $user['id']) {
        return true;
    }
    return parent::isAuthorized($user);
}
```

Now if you try to view, edit or delete a bookmark that does not belong to you, you should be redirected back to the page you came from. If no error message is displayed, add the following to your layout:

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

You should now see the authorization error messages.

Fixing List view and Forms

While view and delete are working, edit, add and index have a few problems:

1. When adding a bookmark you can choose the user.
2. When editing a bookmark you can choose the user.
3. The list page shows bookmarks from other users.

Let's tackle the add form first. To begin with remove the `control('user_id')` from **src/Template/Bookmarks/add.ctp**. With that removed, we'll also update the `add()` action from **src/Controller/BookmarksController.php** to look like:

```
public function add()
{
    $bookmark = $this->Bookmarks->newEntity();
    if ($this->request->is('post')) {
        $bookmark = $this->Bookmarks->patchEntity($bookmark, $this->request->
↪getData());
        $bookmark->user_id = $this->Auth->user('id');
        if ($this->Bookmarks->save($bookmark)) {
            $this->Flash->success('The bookmark has been saved.');
            return $this->redirect(['action' => 'index']);
        }
        $this->Flash->error('The bookmark could not be saved. Please, try again.');
    }
    $tags = $this->Bookmarks->Tags->find('list');
    $this->set(compact('bookmark', 'tags'));
    $this->set('_serialize', ['bookmark']);
}
```

By setting the entity property with the session data, we remove any possibility of the user modifying which user a bookmark is for. We'll do the same for the edit form and action. Your `edit()` action from **src/Controller/BookmarksController.php** should look like:

```
public function edit($id = null)
{
    $bookmark = $this->Bookmarks->get($id, [
        'contain' => ['Tags']
    ]);
    if ($this->request->is(['patch', 'post', 'put'])) {
        $bookmark = $this->Bookmarks->patchEntity($bookmark, $this->request->
↪getData());
        $bookmark->user_id = $this->Auth->user('id');
        if ($this->Bookmarks->save($bookmark)) {
            $this->Flash->success('The bookmark has been saved.');
            return $this->redirect(['action' => 'index']);
        }
        $this->Flash->error('The bookmark could not be saved. Please, try again.');
    }
    $tags = $this->Bookmarks->Tags->find('list');
    $this->set(compact('bookmark', 'tags'));
    $this->set('_serialize', ['bookmark']);
}
```

List View

Now, we only need to show bookmarks for the currently logged in user. We can do that by updating the call to `paginate()`. Make your `index()` action from **src/Controller/BookmarksController.php** look like:

```
public function index()
{
    $this->paginate = [
        'conditions' => [
            'Bookmarks.user_id' => $this->Auth->user('id'),
        ]
    ];
    $this->set('bookmarks', $this->paginate($this->Bookmarks));
    $this->set('_serialize', ['bookmarks']);
```

```
}
```

We should also update the `tags()` action and the related finder method, but we'll leave that as an exercise you can complete on your own.

Improving the Tagging Experience

Right now, adding new tags is a difficult process, as the `TagsController` disallows all access. Instead of allowing access, 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/Bookmark.php** add the following:

```
use Cake\Collection\Collection;

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

This will let us access the `$bookmark->tag_string` computed property. We'll use this property in controls later on. Remember to add the `tag_string` property to the `_accessible` list in your entity, as we'll want to 'save' it later on.

In **src/Model/Entity/Bookmark.php** add the `tag_string` to `$_accessible` this way:

```
protected $_accessible = [
    'user_id' => true,
    'title' => true,
    'description' => true,
    'url' => true,
    'user' => true,
    'tags' => true,
    'tag_string' => true,
];
```

Updating the Views

With the entity updated we can add a new control for our tags. In **src/Template/Bookmarks/add.ctp** and **src/Template/Bookmarks/edit.ctp**, replace the existing `tags._ids` control with the following:

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

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/BookmarksTable.php**:

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

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

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

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

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.

Wrapping Up

We've expanded our bookmarking application to handle authentication and basic authorization/access control scenarios. We've also added some nice UX improvements by leveraging the FormHelper and ORM capabilities.

Thanks for taking the time to explore CakePHP. Next, you can complete the *Blog Tutorial*, learn more about the *Database Access & ORM*, or you can peruse the /topics.

CHAPTER 3

3.0 Migration Guide

This page summarizes the changes from CakePHP 2.x that will assist in migrating a project to 3.0, as well as a reference to get up to date with the changes made to the core since the CakePHP 2.x branch. Be sure to read the other pages in this guide for all the new features and API changes.

Requirements

- CakePHP 3.x supports PHP Version 5.4.16 and above.
- CakePHP 3.x requires the mbstring extension.
- CakePHP 3.x requires the intl extension.

Warning: CakePHP 3.0 will not work if you do not meet the above requirements.

Upgrade Tool

While this document covers all the breaking changes and improvements made in CakePHP 3.0, we've also created a console application to help you complete some of the time consuming mechanical changes. You can get the upgrade tool from github[23].

Application Directory Layout

The application directory layout has changed and now follows PSR-4[24]. You should use the app skeleton[25] project as a reference point when updating your application.

[23] https://github.com/cakephp/upgrade
[24] http://www.php-fig.org/psr/psr-4/
[25] https://github.com/cakephp/app

CakePHP should be installed with Composer

Since CakePHP can no longer be installed via PEAR, or in a shared directory, those options are no longer supported. Instead you should use Composer[26] to install CakePHP into your application.

Namespaces

All of CakePHP's core classes are now namespaced and follow PSR-4 autoloading specifications. For example **src/Cache/Cache.php** is namespaced as `Cake\Cache\Cache`. Global constants and helper methods like *__()* and *debug()* are not namespaced for convenience sake.

Removed Constants

The following deprecated constants have been removed:

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

Configuration

Configuration in CakePHP 3.0 is significantly different than in previous versions. You should read the *Configuration* documentation for how configuration is done in 3.0.

You can no longer use `App::build()` to configure additional class paths. Instead you should map additional paths using your application's autoloader. See the section on *Additional Class Paths* for more information.

Three new configure variables provide the path configuration for plugins, views and locale files. You can add multiple paths to `App.paths.templates`, `App.paths.plugins`, `App.paths.locales` to configure multiple paths for templates, plugins and locale files respectively.

The config key `www_root` has been changed to `wwwRoot` for consistency. Please adjust your **app.php** config file as well as any usage of `Configure::read('App.wwwRoot')`.

New ORM

CakePHP 3.0 features a new ORM that has been re-built from the ground up. The new ORM is significantly different and incompatible with the previous one. Upgrading to the new ORM will require extensive changes in any application that is being upgraded. See the new *Database Access & ORM* documentation for information on how to use the new ORM.

[26] http://getcomposer.org

Basics

- `LogError()` was removed, it provided no benefit and is rarely/never used.
- The following global functions have been removed: `config()`, `cache()`, `clearCache()`, `convertSlashes()`, `am()`, `fileExistsInPath()`, `sortByKey()`.

Debugging

- `Configure::write('debug', $bool)` does not support 0/1/2 anymore. A simple boolean is used instead to switch debug mode on or off.

Object settings/configuration

- Objects used in CakePHP now have a consistent instance-configuration storage/retrieval system. Code which previously accessed for example: `$object->settings` should instead be updated to use `$object->config()`.

Cache

- `Memcache` engine has been removed, use `Cake\Cache\Cache\Engine\Memcached` instead.
- Cache engines are now lazy loaded upon first use.
- `Cake\Cache\Cache::engine()` has been added.
- *`Cake\Cache\Cache::enabled()`* has been added. This replaced the `Cache.disable` configure option.
- *`Cake\Cache\Cache::enable()`* has been added.
- *`Cake\Cache\Cache::disable()`* has been added.
- Cache configurations are now immutable. If you need to change configuration you must first drop the configuration and then re-create it. This prevents synchronization issues with configuration options.
- `Cache::set()` has been removed. It is recommended that you create multiple cache configurations to replace runtime configuration tweaks previously possible with `Cache::set()`.
- All `CacheEngine` subclasses now implement a `config()` method.
- *`Cake\Cache\Cache::readMany()`*, *`Cake\Cache\Cache::deleteMany()`*, and *`Cake\Cache\Cache::writeMany()`* were added.

All `Cake\Cache\Cache\CacheEngine` methods now honor/are responsible for handling the configured key prefix. The *`Cake\Cache\CacheEngine::write()`* no longer permits setting the duration on write - the duration is taken from the cache engine's runtime config. Calling a cache method with an empty key will now throw an `InvalidArgumentException`, instead of returning `false`.

Core

App

- `App::pluginPath()` has been removed. Use `CakePlugin::path()` instead.
- `App::build()` has been removed.
- `App::location()` has been removed.
- `App::paths()` has been removed.
- `App::load()` has been removed.
- `App::objects()` has been removed.
- `App::RESET` has been removed.
- `App::APPEND` has been removed.
- `App::PREPEND` has been removed.
- `App::REGISTER` has been removed.

Plugin

- `Cake\Core\Plugin::load()` does not setup an autoloader unless you set the `autoload` option to `true`.
- When loading plugins you can no longer provide a callable.
- When loading plugins you can no longer provide an array of config files to load.

Configure

- `Cake\Configure\PhpReader` renamed to `Cake\Core\Configure\EnginePhpConfig`
- `Cake\Configure\IniReader` renamed to `Cake\Core\Configure\EngineIniConfig`
- `Cake\Configure\ConfigReaderInterface` renamed to `Cake\Core\Configure\ConfigEngineInterface`
- *`Cake\Core\Configure::consume()`* was added.
- *`Cake\Core\Configure::load()`* now expects the file name without extension suffix as this can be derived from the engine. E.g. using PhpConfig use `app` to load **app.php**.
- Setting a `$config` variable in PHP config file is deprecated. `Cake\Core\Configure\EnginePhpConfig` now expects the config file to return an array.
- A new config engine `Cake\Core\Configure\EngineJsonConfig` has been added.

Object

The `Object` class has been removed. It formerly contained a grab bag of methods that were used in various places across the framework. The most useful of these methods have been extracted into traits. You can use the *`Cake\Log\LogTrait`* to access the `log()` method. The `Cake\Routing\RequestActionTrait` provides `requestAction()`.

Console

The `cake` executable has been moved from the **app/Console** directory to the **bin** directory within the application skeleton. You can now invoke CakePHP's console with `bin/cake`.

TaskCollection Replaced

This class has been renamed to `Cake\Console\TaskRegistry`. See the section on *Registry Objects* for more information on the features provided by the new class. You can use the `cake upgrade rename_collections` to assist in upgrading your code. Tasks no longer have access to callbacks, as there were never any callbacks to use.

Shell

- `Shell::__construct()` has changed. It now takes an instance of `Cake\Console\ConsoleIo`.
- `Shell::param()` has been added as convenience access to the params.

Additionally all shell methods will be transformed to camel case when invoked. For example, if you had a `hello_world()` method inside a shell and invoked it with `bin/cake my_shell hello_world`, you will need to rename the method to `helloWorld`. There are no changes required in the way you invoke commands.

ConsoleOptionParser

- `ConsoleOptionParser::merge()` has been added to merge parsers.

ConsoleInputArgument

- `ConsoleInputArgument::isEqualTo()` has been added to compare two arguments.

Shell / Task

Shells and Tasks have been moved from `Console/Command` and `Console/Command/Task` to `Shell` and `Shell/Task`.

ApiShell Removed

The ApiShell was removed as it didn't provide any benefit over the file source itself and the online documentation/API[27].

SchemaShell Removed

The SchemaShell was removed as it was never a complete database migration implementation and better tools such as Phinx[28] have emerged. It has been replaced by the CakePHP Migrations Plugin[29] which acts as a wrapper between CakePHP and Phinx[30].

[27] https://api.cakephp.org/
[28] https://phinx.org/
[29] https://github.com/cakephp/migrations
[30] https://phinx.org/

ExtractTask

- `bin/cake i18n extract` no longer includes untranslated validation messages. If you want translated validation messages you should wrap those messages in *__()* calls like any other content.

BakeShell / TemplateTask

- Bake is no longer part of the core source and is superseded by CakePHP Bake Plugin[31]
- Bake templates have been moved under **src/Template/Bake**.
- The syntax of Bake templates now uses erb-style tags (`<% %>`) to denote templating logic, allowing php code to be treated as plain text.
- The `bake view` command has been renamed `bake template`.

Event

The `getEventManager()` method, was removed on all objects that had it. An `eventManager()` method is now provided by the `EventManagerTrait`. The `EventManagerTrait` contains the logic of instantiating and keeping a reference to a local event manager.

The Event subsystem has had a number of optional features removed. When dispatching events you can no longer use the following options:

- `passParams` This option is now enabled always implicitly. You cannot turn it off.
- `break` This option has been removed. You must now stop events.
- `breakOn` This option has been removed. You must now stop events.

Log

- Log configurations are now immutable. If you need to change configuration you must first drop the configuration and then re-create it. This prevents synchronization issues with configuration options.
- Log engines are now lazily loaded upon the first write to the logs.
- `Cake\Log\Log::engine()` has been added.
- The following methods have been removed from *`Cake\Log\Log`* :: `defaultLevels()`, `enabled()`, `enable()`, `disable()`.
- You can no longer create custom levels using `Log::levels()`.
- When configuring loggers you should use `'levels'` instead of `'types'`.
- You can no longer specify custom log levels. You must use the default set of log levels. You should use logging scopes to create custom log files or specific handling for different sections of your application. Using a non-standard log level will now throw an exception.
- *`Cake\Log\LogTrait`* was added. You can use this trait in your classes to add the `log()` method.
- The logging scope passed to *`Cake\Log\Log::write()`* is now forwarded to the log engines' `write()` method in order to provide better context to the engines.

[31] https://github.com/cakephp/bake

- Log engines are now required to implement `Psr\Log\LogInterface` instead of Cake's own `LogInterface`. In general, if you extended `Cake\Log\Engine\BaseEngine` you just need to rename the `write()` method to `log()`.
- `Cake\Log\Engine\FileLog` now writes files in `ROOT/logs` instead of `ROOT/tmp/logs`.

Routing

Named Parameters

Named parameters were removed in 3.0. Named parameters were added in 1.2.0 as a 'pretty' version of query string parameters. While the visual benefit is arguable, the problems named parameters created are not.

Named parameters required special handling in CakePHP as well as any PHP or JavaScript library that needed to interact with them, as named parameters are not implemented or understood by any library *except* CakePHP. The additional complexity and code required to support named parameters did not justify their existence, and they have been removed. In their place you should use standard query string parameters or passed arguments. By default `Router` will treat any additional parameters to `Router::url()` as query string arguments.

Since many applications will still need to parse incoming URLs containing named parameters. `Cake\Routing\Router::parseNamedParams()` has been added to allow backwards compatibility with existing URLs.

RequestActionTrait

- `Cake\Routing\RequestActionTrait::requestAction()` has had some of the extra options changed:
  - `options[url]` is now `options[query]`.
  - `options[data]` is now `options[post]`.
  - Named parameters are no longer supported.

Router

- Named parameters have been removed, see above for more information.
- The `full_base` option has been replaced with the `_full` option.
- The `ext` option has been replaced with the `_ext` option.
- `_scheme`, `_port`, `_host`, `_base`, `_full`, `_ext` options added.
- String URLs are no longer modified by adding the plugin/controller/prefix names.
- The default fallback route handling was removed. If no routes match a parameter set `/` will be returned.
- Route classes are responsible for *all* URL generation including query string parameters. This makes routes far more powerful and flexible.
- Persistent parameters were removed. They were replaced with `Cake\Routing\Router::urlFilter()` which allows a more flexible way to mutate URLs being reverse routed.
- `Router::parseExtensions()` has been removed. Use *`Cake\Routing\Router::extensions()`* instead. This method **must** be called before routes are connected. It won't modify existing routes.

- `Router::setExtensions()` has been removed. Use *`Cake\Routing\Router::extensions()`* instead.
- `Router::resourceMap()` has been removed.
- The `[method]` option has been renamed to `_method`.
- The ability to match arbitrary headers with `[]` style parameters has been removed. If you need to parse/match on arbitrary conditions consider using custom route classes.
- `Router::promote()` has been removed.
- `Router::parse()` will now raise an exception when a URL cannot be handled by any route.
- `Router::url()` will now raise an exception when no route matches a set of parameters.
- Routing scopes have been introduced. Routing scopes allow you to keep your routes file DRY and give Router hints on how to optimize parsing & reverse routing URLs.

Route

- `CakeRoute` was re-named to `Route`.
- The signature of `match()` has changed to `match($url,$context = [])` See `Cake\Routing\Route::match()` for information on the new signature.

Dispatcher Filters Configuration Changed

Dispatcher filters are no longer added to your application using `Configure`. You now append them with `Cake\Routing\DispatcherFactory`. This means if your application used `Dispatcher.filters`, you should now use `Cake\Routing\DispatcherFactory::add()`.

In addition to configuration changes, dispatcher filters have had some conventions updated, and features added. See the *Dispatcher Filters* documentation for more information.

FilterAssetFilter

- Plugin & theme assets handled by the AssetFilter are no longer read via `include` instead they are treated as plain text files. This fixes a number of issues with JavaScript libraries like TinyMCE and environments with short_tags enabled.
- Support for the `Asset.filter` configuration and hooks were removed. This feature should be replaced with a plugin or dispatcher filter.

Network

Request

- `CakeRequest` has been renamed to `Cake\Network\Request`.
- `Cake\Network\Request::port()` was added.
- `Cake\Network\Request::scheme()` was added.
- `Cake\Network\Request::cookie()` was added.

- `Cake\Network\Request::$trustProxy` was added. This makes it easier to put CakePHP applications behind load balancers.
- `Cake\Network\Request::$data` is no longer merged with the prefixed data key, as that prefix has been removed.
- `Cake\Network\Request::env()` was added.
- `Cake\Network\Request::acceptLanguage()` was changed from static method to non-static.
- Request detector for "mobile" has been removed from the core. Instead the app template adds detectors for "mobile" and "tablet" using `MobileDetect` lib.
- The method `onlyAllow()` has been renamed to `allowMethod()` and no longer accepts "var args". All method names need to be passed as first argument, either as string or array of strings.

Response

- The mapping of mimetype `text/plain` to extension `csv` has been removed. As a consequence `Cake\Controller\Component\RequestHandlerComponent` doesn't set extension to `csv` if `Accept` header contains mimetype `text/plain` which was a common annoyance when receiving a jQuery XHR request.

Sessions

The session class is no longer static, instead the session can be accessed through the request object. See the *Sessions* documentation for using the session object.

- `Cake\Network\Session` and related session classes have been moved under the `Cake\Network` namespace.
- `SessionHandlerInterface` has been removed in favor of the one provided by PHP itself.
- The property `Session::$requestCountdown` has been removed.
- The session checkAgent feature has been removed. It caused a number of bugs when chrome frame, and flash player are involved.
- The conventional sessions database table name is now `sessions` instead of `cake_sessions`.
- The session cookie timeout is automatically updated in tandem with the timeout in the session data.
- The path for session cookie now defaults to app's base path instead of "/". A new configuration variable `Session.cookiePath` has been added to customize the cookie path.
- A new convenience method `Cake\Network\Session::consume()` has been added to allow reading and deleting session data in a single step.
- The default value of `Cake\Network\Session::clear()`'s argument `$renew` has been changed from `true` to `false`.

Network\Http

- `HttpSocket` is now `Cake\Network\Http\Client`.

- HttpClient has been re-written from the ground up. It has a simpler/easier to use API, support for new authentication systems like OAuth, and file uploads. It uses PHP's stream APIs so there is no requirement for cURL. See the *Http Client* documentation for more information.

Network\Email

- `Cake\Network\Email\Email::config()` is now used to define configuration profiles. This replaces the `EmailConfig` classes in previous versions.
- `Cake\Network\Email\Email::profile()` replaces `config()` as the way to modify per instance configuration options.
- `Cake\Network\Email\Email::drop()` has been added to allow the removal of email configuration.
- `Cake\Network\Email\Email::configTransport()` has been added to allow the definition of transport configurations. This change removes transport options from delivery profiles and allows you to re-use transports across email profiles.
- `Cake\Network\Email\Email::dropTransport()` has been added to allow the removal of transport configuration.

Controller

Controller

- The `$helpers`, `$components` properties are now merged with **all** parent classes not just `AppController` and the plugin AppController. The properties are merged differently now as well. Instead of all settings in all classes being merged together, the configuration defined in the child class will be used. This means that if you have some configuration defined in your AppController, and some configuration defined in a subclass, only the configuration in the subclass will be used.
- `Controller::httpCodes()` has been removed, use `Cake\Network\Response::httpCodes()` instead.
- `Controller::disableCache()` has been removed, use `Cake\Network\Response::disableCache()` instead.
- `Controller::flash()` has been removed. This method was rarely used in real applications and served no purpose anymore.
- `Controller::validate()` and `Controller::validationErrors()` have been removed. They were left over methods from the 1.x days where the concerns of models + controllers were far more intertwined.
- `Controller::loadModel()` now loads table objects.
- The `Controller::$scaffold` property has been removed. Dynamic scaffolding has been removed from CakePHP core. An improved scaffolding plugin, named CRUD, can be found here: https://github.com/FriendsOfCake/crud
- The `Controller::$ext` property has been removed. You now have to extend and override the `View::$_ext` property if you want to use a non-default view file extension.
- The `Controller::$methods` property has been removed. You should now use `Controller::isAction()` to determine whether or not a method name is an action. This change was made to allow easier customization of what is and is not counted as an action.

- The `Controller::$Components` property has been removed and replaced with `_components`. If you need to load components at runtime you should use `$this->loadComponent()` on your controller.
- The signature of *`Cake\Controller\Controller::redirect()`* has been changed to `Controller::redirect(string|array $url,int $status = null)`. The 3rd argument `$exit` has been dropped. The method can no longer send response and exit script, instead it returns a `Response` instance with appropriate headers set.
- The `base`, `webroot`, `here`, `data`, `action`, and `params` magic properties have been removed. You should access all of these properties on `$this->request` instead.
- Underscore prefixed controller methods like `_someMethod()` are no longer treated as private methods. Use proper visibility keywords instead. Only public methods can be used as controller actions.

Scaffold Removed

The dynamic scaffolding in CakePHP has been removed from CakePHP core. It was infrequently used, and never intended for production use. An improved scaffolding plugin, named CRUD, can be found here: https://github.com/FriendsOfCake/crud

ComponentCollection Replaced

This class has been renamed to `Cake\Controller\ComponentRegistry`. See the section on *Registry Objects* for more information on the features provided by the new class. You can use the `cake upgrade rename_collections` to assist in upgrading your code.

Component

- The `_Collection` property is now `_registry`. It contains an instance of `Cake\Controller\ComponentRegistry` now.
- All components should now use the `config()` method to get/set configuration.
- Default configuration for components should be defined in the `$_defaultConfig` property. This property is automatically merged with any configuration provided to the constructor.
- Configuration options are no longer set as public properties.
- The `Component::initialize()` method is no longer an event listener. Instead, it is a post-constructor hook like `Table::initialize()` and `Controller::initialize()`. The new `Component::beforeFilter()` method is bound to the same event that `Component::initialize()` used to be. The initialize method should have the following signature `initialize(array $config)`.

Controller\Components

CookieComponent

- Uses `Cake\Network\Request::cookie()` to read cookie data, this eases testing, and allows for ControllerTestCase to set cookies.
- Cookies encrypted in previous versions of CakePHP using the `cipher()` method are now un-readable because `Security::cipher()` has been removed. You will need to re-encrypt cookies with the `rijndael()` or `aes()` method before upgrading.

- `CookieComponent::type()` has been removed and replaced with configuration data accessed through `config()`.
- `write()` no longer takes `encryption` or `expires` parameters. Both of these are now managed through config data. See *Cookie* for more information.
- The path for cookies now defaults to app's base path instead of "/".

AuthComponent

- `Default` is now the default password hasher used by authentication classes. It uses exclusively the bcrypt hashing algorithm. If you want to continue using SHA1 hashing used in 2.x use `'passwordHasher' => 'Weak'` in your authenticator configuration.
- A new `FallbackPasswordHasher` was added to help users migrate old passwords from one algorithm to another. Check AuthComponent's documentation for more info.
- `BlowfishAuthenticate` class has been removed. Just use `FormAuthenticate`
- `BlowfishPasswordHasher` class has been removed. Use `DefaultPasswordHasher` instead.
- The `loggedIn()` method has been removed. Use `user()` instead.
- Configuration options are no longer set as public properties.
- The methods `allow()` and `deny()` no longer accept "var args". All method names need to be passed as first argument, either as string or array of strings.
- The method `login()` has been removed and replaced by `setUser()` instead. To login a user you now have to call `identify()` which returns user info upon successful identification and then use `setUser()` to save the info to session for persistence across requests.
- `BaseAuthenticate::_password()` has been removed. Use a `PasswordHasher` class instead.
- `BaseAuthenticate::logout()` has been removed.
- `AuthComponent` now triggers two events `Auth.afterIdentify` and `Auth.logout` after a user has been identified and before a user is logged out respectively. You can set callback functions for these events by returning a mapping array from `implementedEvents()` method of your authenticate class.

ACL related classes were moved to a separate plugin. Password hashers, Authentication and Authorization providers where moved to the `\Cake\Auth` namespace. You are required to move your providers and hashers to the `App\Auth` namespace as well.

RequestHandlerComponent

- The following methods have been removed from RequestHandler component:: `isAjax()`, `isFlash()`, `isSSL()`, `isPut()`, `isPost()`, `isGet()`, `isDelete()`. Use the `Cake\Network\Request::is()` method instead with relevant argument.
- `RequestHandler::setContent()` was removed, use `Cake\Network\Response::type()` instead.
- `RequestHandler::getReferer()` was removed, use `Cake\Network\Request::referer()` instead.
- `RequestHandler::getClientIP()` was removed, use `Cake\Network\Request::clientIp()` instead.
- `RequestHandler::getAjaxVersion()` was removed.

- `RequestHandler::mapType()` was removed, use `Cake\Network\Response::mapType()` instead.
- Configuration options are no longer set as public properties.

SecurityComponent

- The following methods and their related properties have been removed from Security component: `requirePost()`, `requireGet()`, `requirePut()`, `requireDelete()`. Use the `Cake\Network\Request::allowMethod()` instead.
- `SecurityComponent::$disabledFields()` has been removed, use `SecurityComponent::$unlockedFields()`.
- The CSRF related features in SecurityComponent have been extracted and moved into a separate CsrfComponent. This allows you to use CSRF protection without having to use form tampering prevention.
- Configuration options are no longer set as public properties.
- The methods `requireAuth()` and `requireSecure()` no longer accept "var args". All method names need to be passed as first argument, either as string or array of strings.

SessionComponent

- `SessionComponent::setFlash()` is deprecated. You should use *Flash* instead.

Error

Custom ExceptionRenderers are now expected to either return a `Cake\Network\Response` object or string when rendering errors. This means that any methods handling specific exceptions must return a response or string value.

Model

The Model layer in 2.x has been entirely re-written and replaced. You should review the *New ORM Upgrade Guide* for information on how to use the new ORM.

- The `Model` class has been removed.
- The `BehaviorCollection` class has been removed.
- The `DboSource` class has been removed.
- The `Datasource` class has been removed.
- The various datasource classes have been removed.

ConnectionManager

- ConnectionManager has been moved to the `Cake\Datasource` namespace.
- ConnectionManager has had the following methods removed:
  - `sourceList`
  - `getSourceName`

  - `loadDataSource`
  - `enumConnectionObjects`

- `Database\ConnectionManager::config()` has been added and is now the only way to configure connections.
- `Database\ConnectionManager::get()` has been added. It replaces `getDataSource()`.
- `Database\ConnectionManager::configured()` has been added. It and `config()` replace `sourceList()` & `enumConnectionObjects()` with a more standard and consistent API.
- `ConnectionManager::create()` has been removed. It can be replaced by `config($name,$config)` and `get($name)`.

Behaviors

- Underscore prefixed behavior methods like `_someMethod()` are no longer treated as private methods. Use proper visibility keywords instead.

TreeBehavior

The TreeBehavior was completely re-written to use the new ORM. Although it works the same as in 2.x, a few methods were renamed or removed:

- `TreeBehavior::children()` is now a custom finder `find('children')`.
- `TreeBehavior::generateTreeList()` is now a custom finder `find('treeList')`.
- `TreeBehavior::getParentNode()` was removed.
- `TreeBehavior::getPath()` is now a custom finder `find('path')`.
- `TreeBehavior::reorder()` was removed.
- `TreeBehavior::verify()` was removed.

TestSuite

TestCase

- `_normalizePath()` has been added to allow path comparison tests to run across all operation systems regarding their DS settings (\ in Windows vs / in UNIX, for example).

The following assertion methods have been removed as they have long been deprecated and replaced by their new PHPUnit counterpart:

- `assertEqual()` in favor of `assertEquals()`
- `assertNotEqual()` in favor of `assertNotEquals()`
- `assertIdentical()` in favor of `assertSame()`
- `assertNotIdentical()` in favor of `assertNotSame()`
- `assertPattern()` in favor of `assertRegExp()`
- `assertNoPattern()` in favor of `assertNotRegExp()`
- `assertReference()` if favor of `assertSame()`

- `assertIsA()` in favor of `assertInstanceOf()`

Note that some methods have switched the argument order, e.g. `assertEqual($is, $expected)` should now be `assertEquals($expected, $is)`.

The following assertion methods have been deprecated and will be removed in the future:

- `assertWithinMargin()` in favor of `assertWithinRange()`
- `assertTags()` in favor of `assertHtml()`

Both method replacements also switched the argument order for a consistent assert method API with `$expected` as first argument.

The following assertion methods have been added:

- `assertNotWithinRange()` as counter part to `assertWithinRange()`

View

Themes are now Basic Plugins

Having themes and plugins as ways to create modular application components has proven to be limited, and confusing. In CakePHP 3.0, themes no longer reside **inside** the application. Instead they are standalone plugins. This solves a few problems with themes:

- You could not put themes *in* plugins.
- Themes could not provide helpers, or custom view classes.

Both these issues are solved by converting themes into plugins.

View Folders Renamed

The folders containing view files now go under **src/Template** instead of **src/View**. This was done to separate the view files from files containing php classes (eg. Helpers, View classes).

The following View folders have been renamed to avoid naming collisions with controller names:

- `Layouts` is now `Layout`
- `Elements` is now `Element`
- `Errors` is now `Error`
- `Emails` is now `Email` (same for `Email` inside `Layout`)

HelperCollection Replaced

This class has been renamed to `Cake\View\HelperRegistry`. See the section on *Registry Objects* for more information on the features provided by the new class. You can use the `cake upgrade rename_collections` to assist in upgrading your code.

View Class

- The `plugin` key has been removed from `$options` argument of *`Cake\View\View::element()`*. Specify the element name as `SomePlugin.element_name` instead.
- `View::getVar()` has been removed, use `Cake\View\View::get()` instead.
- `View::$ext` has been removed and instead a protected property `View::$_ext` has been added.
- `View::addScript()` has been removed. Use *Using View Blocks* instead.
- The `base`, `webroot`, `here`, `data`, `action`, and `params` magic properties have been removed. You should access all of these properties on `$this->request` instead.
- `View::start()` no longer appends to an existing block. Instead it will overwrite the block content when end is called. If you need to combine block contents you should fetch the block content when calling start a second time, or use the capturing mode of `append()`.
- `View::prepend()` no longer has a capturing mode.
- `View::startIfEmpty()` has been removed. Now that start() always overwrites startIfEmpty serves no purpose.
- The `View::$Helpers` property has been removed and replaced with `_helpers`. If you need to load helpers at runtime you should use `$this->addHelper()` in your view files.
- `View` will now raise `Cake\View\Exception\MissingTemplateException` when templates are missing instead of `MissingViewException`.

ViewBlock

- `ViewBlock::append()` has been removed, use `Cake\ViewViewBlock::concat()` instead. However, `View::append()` still exists.

JsonView

- By default JSON data will have HTML entities encoded now. This prevents possible XSS issues when JSON view content is embedded in HTML files.
- `Cake\View\JsonView` now supports the `_jsonOptions` view variable. This allows you to configure the bit-mask options used when generating JSON.

XmlView

- `Cake\View\XmlView` now supports the `_xmlOptions` view variable. This allows you to configure the options used when generating XML.

View\Helper

- The `$settings` property is now called `$_config` and should be accessed through the `config()` method.
- Configuration options are no longer set as public properties.
- `Helper::clean()` was removed. It was never robust enough to fully prevent XSS. instead you should escape content with *`h`* or use a dedicated library like htmlPurifier.

- `Helper::output()` was removed. This method was deprecated in 2.x.
- Methods `Helper::webroot()`, `Helper::url()`, `Helper::assetUrl()`, `Helper::assetTimestamp()` have been moved to new *`Cake\View\Helper\UrlHelper`* helper. `Helper::url()` is now available as *`Cake\View\Helper\UrlHelper::build()`*.
- Magic accessors to deprecated properties have been removed. The following properties now need to be accessed from the request object:
  - base
  - here
  - webroot
  - data
  - action
  - params

Helper

Helper has had the following methods removed:

- `Helper::setEntity()`
- `Helper::entity()`
- `Helper::model()`
- `Helper::field()`
- `Helper::value()`
- `Helper::_name()`
- `Helper::_initInputField()`
- `Helper::_selectedArray()`

These methods were part used only by FormHelper, and part of the persistent field features that have proven to be problematic over time. FormHelper no longer relies on these methods and the complexity they provide is not necessary anymore.

The following methods have been removed:

- `Helper::_parseAttributes()`
- `Helper::_formatAttribute()`

These methods can now be found on the `StringTemplate` class that helpers frequently use. See the `StringTemplateTrait` for an easy way to integrate string templates into your own helpers.

FormHelper

FormHelper has been entirely rewritten for 3.0. It features a few large changes:

- FormHelper works with the new ORM. But has an extensible system for integrating with other ORMs or datasources.
- FormHelper features an extensible widget system that allows you to create new custom input widgets and augment the built-in ones.

- String templates are the foundation of the helper. Instead of munging arrays together everywhere, most of the HTML FormHelper generates can be customized in one central place using template sets.

In addition to these larger changes, some smaller breaking changes have been made as well. These changes should help streamline the HTML FormHelper generates and reduce the problems people had in the past:

- The `data[` prefix was removed from all generated inputs. The prefix serves no real purpose anymore.
- The various standalone input methods like `text()`, `select()` and others no longer generate id attributes.
- The `inputDefaults` option has been removed from `create()`.
- Options `default` and `onsubmit` of `create()` have been removed. Instead one should use JavaScript event binding or set all required js code for `onsubmit`.
- `end()` can no longer make buttons. You should create buttons with `button()` or `submit()`.
- `FormHelper::tagIsInvalid()` has been removed. Use `isFieldError()` instead.
- `FormHelper::inputDefaults()` has been removed. You can use `templates()` to define/augment the templates FormHelper uses.
- The `wrap` and `class` options have been removed from the `error()` method.
- The `showParents` option has been removed from select().
- The `div`, `before`, `after`, `between` and `errorMessage` options have been removed from `input()`. You can use templates to update the wrapping HTML. The `templates` option allows you to override the loaded templates for one input.
- The `separator`, `between`, and `legend` options have been removed from `radio()`. You can use templates to change the wrapping HTML now.
- The `format24Hours` parameter has been removed from `hour()`. It has been replaced with the `format` option.
- The `minYear`, and `maxYear` parameters have been removed from `year()`. Both of these parameters can now be provided as options.
- The `dateFormat` and `timeFormat` parameters have been removed from `datetime()`. You can use the template to define the order the inputs should be displayed in.
- The `submit()` has had the `div`, `before` and `after` options removed. You can customize the `submitContainer` template to modify this content.
- The `inputs()` method no longer accepts `legend` and `fieldset` in the `$fields` parameter, you must use the `$options` parameter. It now also requires `$fields` parameter to be an array. The `$blacklist` parameter has been removed, the functionality has been replaced by specifying `'field' => false` in the `$fields` parameter.
- The `inline` parameter has been removed from postLink() method. You should use the `block` option instead. Setting `block => true` will emulate the previous behavior.
- The `timeFormat` parameter for `hour()`, `time()` and `dateTime()` now defaults to 24, complying with ISO 8601.
- The `$confirmMessage` argument of *`Cake\View\Helper\FormHelper::postLink()`* has been removed. You should now use key `confirm` in `$options` to specify the message.
- Checkbox and radio input types are now rendered *inside* of label elements by default. This helps increase compatibility with popular CSS libraries like Bootstrap[32] and Foundation[33].

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

- Templates tags are now all camelBacked. Pre-3.0 tags `formstart`, `formend`, `hiddenblock` and `inputsubmit` are now `formStart`, `formEnd`, `hiddenBlock` and `inputSubmit`. Make sure you change them if they are customized in your app.

It is recommended that you review the *Form* documentation for more details on how to use the FormHelper in 3.0.

HtmlHelper

- `HtmlHelper::useTag()` has been removed, use `tag()` instead.
- `HtmlHelper::loadConfig()` has been removed. Customizing the tags can now be done using `templates()` or the `templates` setting.
- The second parameter `$options` for `HtmlHelper::css()` now always requires an array as documented.
- The first parameter `$data` for `HtmlHelper::style()` now always requires an array as documented.
- The `inline` parameter has been removed from meta(), css(), script(), scriptBlock() methods. You should use the `block` option instead. Setting `block => true` will emulate the previous behavior.
- `HtmlHelper::meta()` now requires `$type` to be a string. Additional options can further on be passed as `$options`.
- `HtmlHelper::nestedList()` now requires `$options` to be an array. The forth argument for the tag type has been removed and included in the `$options` array.
- The `$confirmMessage` argument of *`Cake\View\Helper\HtmlHelper::link()`* has been removed. You should now use key `confirm` in `$options` to specify the message.

PaginatorHelper

- `link()` has been removed. It was no longer used by the helper internally. It had low usage in user land code, and no longer fit the goals of the helper.
- `next()` no longer has 'class', or 'tag' options. It no longer has disabled arguments. Instead templates are used.
- `prev()` no longer has 'class', or 'tag' options. It no longer has disabled arguments. Instead templates are used.
- `first()` no longer has 'after', 'ellipsis', 'separator', 'class', or 'tag' options.
- `last()` no longer has 'after', 'ellipsis', 'separator', 'class', or 'tag' options.
- `numbers()` no longer has 'separator', 'tag', 'currentTag', 'currentClass', 'class', 'tag', 'ellipsis' options. These options are now facilitated through templates. It also requires the `$options` parameter to be an array now.
- The `%page%` style placeholders have been removed from *`Cake\View\Helper\PaginatorHelper::counter()`*. Use `{{page}}` style placeholders instead.
- `url()` has been renamed to `generateUrl()` to avoid method declaration clashes with `Helper::url()`.

By default all links and inactive texts are wrapped in `<li>` elements. This helps make CSS easier to write, and improves compatibility with popular CSS frameworks.

Instead of the various options in each method, you should use the templates feature. See the *PaginatorHelper Templates* documentation for information on how to use templates.

TimeHelper

- `TimeHelper::__set()`, `TimeHelper::__get()`, and `TimeHelper::__isset()` were removed. These were magic methods for deprecated attributes.
- `TimeHelper::serverOffset()` has been removed. It promoted incorrect time math practices.
- `TimeHelper::niceShort()` has been removed.

NumberHelper

- `NumberHelper::format()` now requires `$options` to be an array.

SessionHelper

- The `SessionHelper` has been deprecated. You can use `$this->request->session()` directly, and the flash message functionality has been moved into *Flash* instead.

JsHelper

- `JsHelper` and all associated engines have been removed. It could only generate a very small subset of JavaScript code for selected library and hence trying to generate all JavaScript code using just the helper often became an impediment. It's now recommended to directly use JavaScript library of your choice.

CacheHelper Removed

CacheHelper has been removed. The caching functionality it provided was non-standard, limited and incompatible with non-HTML layouts and data views. These limitations meant a full rebuild would be necessary. Edge Side Includes have become a standardized way to implement the functionality CacheHelper used to provide. However, implementing Edge Side Includes[34] in PHP has a number of limitations and edge cases. Instead of building a sub-par solution, we recommend that developers needing full response caching use Varnish[35] or Squid[36] instead.

I18n

The I18n subsystem was completely rewritten. In general, you can expect the same behavior as in previous versions, specifically if you are using the `__()` family of functions.

Internally, the `I18n` class uses `Aura\Intl`, and appropriate methods are exposed to access the specific features of this library. For this reason most methods inside `I18n` were removed or renamed.

Due to the use of `ext/intl`, the L10n class was completely removed. It provided outdated and incomplete data in comparison to the data available from the `Locale` class in PHP.

The default application language will no longer be changed automatically by the browser accepted language nor by having the `Config.language` value set in the browser session. You can, however, use a dispatcher filter to get automatic language switching from the `Accept-Language` header sent by the browser:

[34] http://en.wikipedia.org/wiki/Edge_Side_Includes
[35] http://varnish-cache.org
[36] http://squid-cache.org

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

There is no built-in replacement for automatically selecting the language by setting a value in the user session.

The default formatting function for translated messages is no longer `sprintf`, but the more advanced and feature rich `MessageFormatter` class. In general you can rewrite placeholders in messages as follows:

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

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

You can avoid rewriting your messages by using the old `sprintf` formatter:

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

Additionally, the `Config.language` value was removed and it can no longer be used to control the current language of the application. Instead, you can use the `I18n` class:

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

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

- The methods below have been moved:
  - From `Cake\I18n\Multibyte::utf8()` to `Cake\Utility\Text::utf8()`
  - From `Cake\I18n\Multibyte::ascii()` to `Cake\Utility\Text::ascii()`
  - From `Cake\I18n\Multibyte::checkMultibyte()` to `Cake\Utility\Text::isMultibyte()`
- Since CakePHP now requires the mbstring extension, the `Multibyte` class has been removed.
- Error messages throughout CakePHP are no longer passed through I18n functions. This was done to simplify the internals of CakePHP and reduce overhead. The developer facing messages are rarely, if ever, actually translated - so the additional overhead reaps very little benefit.

L10n

- `Cake\I18n\L10n` 's constructor now takes a `Cake\Network\Request` instance as argument.

Testing

- The `TestShell` has been removed. CakePHP, the application skeleton and newly baked plugins all use `phpunit` to run tests.
- The webrunner (webroot/test.php) has been removed. CLI adoption has greatly increased since the initial release of 2.x. Additionaly, CLI runners offer superior integration with IDE's and other automated tooling.

  If you find yourself in need of a way to run tests from a browser you should checkout VisualPHPUnit[37]. It offers many additional features over the old webrunner.

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

- `ControllerTestCase` is deprecated and will be removed for CakePHP 3.0.0. You should use the new *Controller Integration Testing* features instead.
- Fixtures should now be referenced using their plural form:

```
// Instead of
$fixtures = ['app.article'];

// You should use
$fixtures = ['app.articles'];
```

Utility

Set Class Removed

The Set class has been removed, you should use the Hash class instead now.

Folder & File

The folder and file classes have been renamed:

- `Cake\Utility\File` renamed to *`Cake\Filesystem\File`*
- `Cake\Utility\Folder` renamed to *`Cake\Filesystem\Folder`*

Inflector

- The default value for `$replacement` argument of *`Cake\Utility\Inflector::slug()`* has been changed from underscore (_) to dash (-). Using dashes to separate words in URLs is the popular choice and also recommended by Google.
- Transliterations for *`Cake\Utility\Inflector::slug()`* have changed. If you use custom transliterations you will need to update your code. Instead of regular expressions, transliterations use simple string replacement. This yielded significant performance improvements:

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

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

- Separate set of uninflected and irregular rules for pluralization and singularization have been removed. Instead we now have a common list for each. When using *`Cake\Utility\Inflector::rules()`* with type 'singular' and 'plural' you can no longer use keys like 'uninflected', 'irregular' in `$rules` argument array.

You can add / overwrite the list of uninflected and irregular rules using *Cake\Utility\Inflector::rules()* by using values 'uninflected' and 'irregular' for `$type` argument.

Sanitize

- `Sanitize` class has been removed.

Security

- `Security::cipher()` has been removed. It is insecure and promoted bad cryptographic practices. You should use `Security::encrypt()` instead.
- The Configure value `Security.cipherSeed` is no longer required. With the removal of `Security::cipher()` it serves no use.
- Backwards compatibility in `Cake\Utility\Security::rijndael()` for values encrypted prior to CakePHP 2.3.1 has been removed. You should re-encrypt values using `Security::encrypt()` and a recent version of CakePHP 2.x before migrating.
- The ability to generate a blowfish hash has been removed. You can no longer use type "blowfish" for `Security::hash()`. One should just use PHP's *password_hash()* and *password_verify()* to generate and verify blowfish hashes. The compability library ircmaxell/password-compat[38] which is installed along with CakePHP provides these functions for PHP < 5.5.
- OpenSSL is now used over mcrypt when encrypting/decrypting data. This change provides better performance and future proofs CakePHP against distros dropping support for mcrypt.
- `Security::rijndael()` is deprecated and only available when using mcrypt.

Warning: Data encrypted with Security::encrypt() in previous versions is not compatible with the openssl implementation. You should *set the implementation to mcrypt* when upgrading.

Time

- `CakeTime` has been renamed to *Cake\I18n\Time*.
- `CakeTime::serverOffset()` has been removed. It promoted incorrect time math practises.
- `CakeTime::niceShort()` has been removed.
- `CakeTime::convert()` has been removed.
- `CakeTime::convertSpecifiers()` has been removed.
- `CakeTime::dayAsSql()` has been removed.
- `CakeTime::daysAsSql()` has been removed.
- `CakeTime::fromString()` has been removed.
- `CakeTime::gmt()` has been removed.
- `CakeTime::toATOM()` has been renamed to `toAtomString`.
- `CakeTime::toRSS()` has been renamed to `toRssString`.

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

- `CakeTime::toUnix()` has been renamed to `toUnixString`.
- `CakeTime::wasYesterday()` has been renamed to `isYesterday` to match the rest of the method naming.
- `CakeTime::format()` Does not use `sprintf` format strings anymore, you can use `i18nFormat` instead.
- `Time::timeAgoInWords()` now requires `$options` to be an array.

Time is not a collection of static methods anymore, it extends `DateTime` to inherit all its methods and adds location aware formatting functions with the help of the `intl` extension.

In general, expressions looking like this:

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

Can be migrated by rewriting it to:

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

Number

The Number library was rewritten to internally use the `NumberFormatter` class.

- `CakeNumber` has been renamed to *`Cake\I18n\Number`*.
- `Number::format()` now requires `$options` to be an array.
- `Number::addFormat()` was removed.
- `Number::fromReadableSize()` has been moved to *`Cake\Utility\Text::parseFileSize()`*.

Validation

- The range for `Validation::range()` now is inclusive if `$lower` and `$upper` are provided.
- `Validation::ssn()` has been removed.

Xml

- `Xml::build()` now requires `$options` to be an array.
- `Xml::build()` no longer accepts a URL. If you need to create an XML document from a URL, use *Http\Client*.

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[39] and the Bakery[40] for existing applications and components.

Bookmarker Tutorial

This tutorial will walk you through the creation of a simple bookmarking application (bookmarker). To start with, we'll be installing CakePHP, creating our database, and using the tools CakePHP provides to get our application up fast.

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: CakePHP will be taking the reins from there. 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 have got an up to date PHP version:

```
php -v
```

You should at least have got installed PHP 5.6.0 (CLI) or higher. Your webserver's PHP version must also be of 5.6.0 or higher, and should best be the same version your command line interface (CLI) PHP version is of. If you'd like to see the completed application, checkout cakephp/bookmarker[41]. Let's get started!

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, it's as easy as running the following:

[39] https://plugins.cakephp.org/
[40] https://bakery.cakephp.org/
[41] https://github.com/cakephp/bookmarker-tutorial

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

Or, you can download `composer.phar` from the Composer website[42].

Then simply type the following line in your terminal from your installation directory to install the CakePHP application skeleton in the **bookmarker** directory:

```
php composer.phar create-project --prefer-dist cakephp/app bookmarker
```

If you downloaded and ran the Composer Windows Installer[43], then type the following line in your terminal from your installation directory (ie. C:\wamp\www\dev\cakephp3):

```
composer self-update && composer create-project --prefer-dist cakephp/app bookmarker
```

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 something like the following:

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

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:

```
bin/cake server
```

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

[42] https://getcomposer.org/download/
[43] https://getcomposer.org/Composer-Setup.exe

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 checkmarks other than CakePHP being able to connect to your database. If not, you may need to install additional PHP extensions, or set directory permissions.

Creating the Database

Next, let's set up the database for our bookmarking application. If you haven't already done so, create an empty database for use in this tutorial, with a name of your choice, e.g. `cake_bookmarks`. You can execute the following SQL to create the necessary tables:

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

CREATE TABLE bookmarks (
    id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT NOT NULL,
    title VARCHAR(50),
    description TEXT,
    url TEXT,
    created DATETIME,
    modified DATETIME,
    FOREIGN KEY user_key (user_id) REFERENCES users(id)
);

CREATE TABLE tags (
    id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(255),
    created DATETIME,
    modified DATETIME,
    UNIQUE KEY (title)
);

CREATE TABLE bookmarks_tags (
    bookmark_id INT NOT NULL,
    tag_id INT NOT NULL,
    PRIMARY KEY (bookmark_id, tag_id),
    FOREIGN KEY tag_key(tag_id) REFERENCES tags(id),
    FOREIGN KEY bookmark_key(bookmark_id) REFERENCES bookmarks(id)
);
```

You may have noticed that the `bookmarks_tags` table used a composite primary key. CakePHP supports composite primary keys almost everywhere, making it easier to build multi-tenanted applications.

The table and column names we used were not arbitrary. By using CakePHP's *naming conventions*, we can leverage CakePHP better and avoid having to configure the framework. CakePHP is flexible enough to accommodate even inconsistent legacy database schemas, but adhering to the conventions will save you time.

Database Configuration

Next, let's tell CakePHP where our database is and how to connect to it. For many, this will be the first and last time you will need to configure anything.

The configuration should be pretty straightforward: just replace the values in the `Datasources.default` array in the **config/app.php** file with those that apply to your setup. A sample completed configuration array might look something like the following:

```
return [
    // More configuration above.
    'Datasources' => [
        'default' => [
            'className' => 'Cake\Database\Connection',
            'driver' => 'Cake\Database\Driver\Mysql',
            'persistent' => false,
            'host' => 'localhost',
            'username' => 'cakephp',
            'password' => 'AngelF00dC4k3~',
            'database' => 'cake_bookmarks',
            'encoding' => 'utf8',
            'timezone' => 'UTC',
            'cacheMetadata' => true,
        ],
    ],
    // More configuration below.
];
```

Once you've saved your **config/app.php** file, you should see that 'CakePHP is able to connect to the database' section have a checkmark.

Note: A copy of CakePHP's default configuration file is found in **config/app.default.php**.

Generating Scaffold Code

Because our database is following the CakePHP conventions, we can use the *bake console* application to quickly generate a basic application. In your command line run the following commands:

```
// On Windows you'll need to use bin\cake instead.
bin/cake bake all users
bin/cake bake all bookmarks
bin/cake bake all tags
```

This will generate the controllers, models, views, their corresponding test cases, and fixtures for our users, bookmarks and tags resources. If you've stopped your server, restart it and go to **http://localhost:8765/bookmarks**.

You should see a basic but functional application providing data access to your application's database tables. Once you're at the list of bookmarks, add a few users, bookmarks, and tags.

Note: If you see a Not Found (404) page, confirm that the Apache mod_rewrite module is loaded.

Adding Password Hashing

When you created your users (by visiting **http://localhost:8765/users**), you probably noticed that the passwords were stored in plain text. This is pretty bad from a security point of view, so let's get that fixed.

This is also a good time to talk about the model layer in CakePHP. In CakePHP, we separate the methods that operate on a collection of objects, and a single object into different classes. 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 convention based setter methods 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:

```
namespace App\Model\Entity;

use Cake\Auth\DefaultPasswordHasher; //include this line
use Cake\ORM\Entity;

class User extends Entity
{

    // Code from bake.

    protected function _setPassword($value)
    {
        $hasher = new DefaultPasswordHasher();
        return $hasher->hash($value);
    }
}
```

Now update one of the users you created earlier, if you change their password, you should see a hashed password instead of the original value on the list or view pages. CakePHP hashes passwords with bcrypt[44] by default. You can also use sha1 or md5 if you're working with an existing database.

Note: If the password doesn't get hashed, make sure you followed the same case for the password member of the class while naming the setter function

---

Getting Bookmarks with a Specific Tag

Now that we're storing passwords safely, we can build out some more interesting features in our application. Once you've amassed a collection of bookmarks, it is helpful to be able to search through them by tag. Next we'll implement a route, controller action, and finder method to search through bookmarks by tag.

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

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

Router::defaultRouteClass(DashedRoute::class);

// New route we're adding for our tagged action.
// The trailing `*` tells CakePHP that this action has
// passed parameters.
Router::scope(
```

[44] http://codahale.com/how-to-safely-store-a-password/

```
    '/bookmarks',
    ['controller' => 'Bookmarks'],
    function ($routes) {
        $routes->connect('/tagged/*', ['action' => 'tags']);
    }
);

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

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

The above defines a new 'route' which connects the **/bookmarks/tagged/** path, to `BookmarksController::tags()`. By defining routes, you can isolate how your URLs look, from how they are implemented. If we were to visit **http://localhost:8765/bookmarks/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/BookmarksController.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 BookmarksTable to find tagged bookmarks.
    $bookmarks = $this->Bookmarks->find('tagged', [
        'tags' => $tags
    ]);

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

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

Creating the Finder Method

In CakePHP we like to keep our controller actions slim, and put most of our application's logic in the models. If you were to visit the **/bookmarks/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/BookmarksTable.php** add the following:

```
// 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(Query $query, array $options)
{
    $bookmarks = $this->find()
        ->select(['id', 'url', 'title', 'description']);

    if (empty($options['tags'])) {
        $bookmarks
            ->leftJoinWith('Tags')
            ->where(['Tags.title IS' => null]);
    } else {
        $bookmarks
            ->innerJoinWith('Tags')
            ->where(['Tags.title IN ' => $options['tags']]);
    }

    return $bookmarks->group(['Bookmarks.id']);
}
```

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 `innerJoinWith()`, `where()` and `group()` methods which allow us to find distinct bookmarks that have a matching tag. When no tags are provided we use a `leftJoinWith()` and modify the 'where' condition, finding bookmarks without tags.

Creating the View

Now if you visit the **/bookmarks/tagged** URL, CakePHP will show an error letting you know that you have not made a view file. Next, let's build the view file for our `tags()` action. In **src/Template/Bookmarks/tags.ctp** put the following content:

```
<h1>
    Bookmarks tagged with
    <?= $this->Text->toList(h($tags)) ?>
</h1>

<section>
<?php foreach ($bookmarks as $bookmark): ?>
    <article>
        <!-- Use the HtmlHelper to create a link -->
        <h4><?= $this->Html->link($bookmark->title, $bookmark->url) ?></h4>
        <small><?= h($bookmark->url) ?></small>

        <!-- Use the TextHelper to format text -->
        <?= $this->Text->autoParagraph(h($bookmark->description)) ?>
    </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 user data to prevent HTML injection issues.

The **tags.ctp** 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 `$tags` and `$bookmarks` variables in our view. 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 templates as local variables.

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

So far, we've created a basic application to manage bookmarks, tags and users. However, everyone can see everyone else's tags. In the next chapter, we'll implement authentication and restrict the visible bookmarks to only those that belong to the current user.

Now continue to *Bookmarker Tutorial Part 2* to continue building your application or dive into the documentation to learn more about what CakePHP can do for you.

Bookmarker Tutorial Part 2

After finishing *the first part of this tutorial* you should have a very basic bookmarking application. In this chapter we'll be adding authentication and restricting the bookmarks each user can see/modify to only the ones they own.

Adding Login

In CakePHP, authentication is handled by *Components*. Components can be thought of as ways to create reusable chunks of controller code related to a specific feature or concept. Components can also hook into the controller's event life-cycle and interact with your application that way. To get started, we'll add the *AuthComponent* to our application. We'll pretty much want every method to require authentication, so we'll add AuthComponent in our AppController:

```
// In src/Controller/AppController.php
namespace App\Controller;

use Cake\Controller\Controller;

class AppController extends Controller
{
    public function initialize()
    {
        $this->loadComponent('Flash');
        $this->loadComponent('Auth', [
            'authenticate' => [
                'Form' => [
                    'fields' => [
                        'username' => 'email',
                        'password' => 'password'
                    ]
                ]
            ],
            'loginAction' => [
                'controller' => 'Users',
                'action' => 'login'
            ],
            'unauthorizedRedirect' => $this->referer() // If unauthorized, return
↪them to page they were just on
        ]);

        // Allow the display action so our pages controller
        // continues to work.
        $this->Auth->allow(['display']);
```

```
    }
}
```

We've just told CakePHP that we want to load the `Flash` and `Auth` components. In addition, we've customized the configuration of AuthComponent, as our users table uses `email` as the username. Now, if you go to any URL you'll be kicked to **/users/login**, which will show an error page as we have not written that code yet. So let's create the login action:

```
// In src/Controller/UsersController.php
public function login()
{
    if ($this->request->is('post')) {
        $user = $this->Auth->identify();
        if ($user) {
            $this->Auth->setUser($user);
            return $this->redirect($this->Auth->redirectUrl());
        }
        $this->Flash->error('Your username or password is incorrect.');
    }
}
```

And in **src/Template/Users/login.ctp** add the following:

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

Note: The `control()` method is available since 3.4. For prior versions you can use the `input()` method instead.

Now that we have a simple login form, we should be able to log in with one of the users that has a hashed password.

Note: If none of your users have hashed passwords, comment the `loadComponent('Auth')` line. Then go and edit the user, saving a new password for them.

You should now be able to log in. If not, make sure you are using a user that has a hashed password.

Adding Logout

Now that people can log in, you'll probably want to provide a way to log out as well. Again, in the `UsersController`, add the following code:

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

public function logout()
{
    $this->Flash->success('You are now logged out.');
```

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

This code whitelists the `logout` action as a public action, and implements the logout method. Now you can visit `/users/logout` to log out. You should then be sent to the login page.

Enabling Registrations

If you aren't logged in and you try to visit **/users/add** you will be kicked to the login page. We should fix that as we want to allow people to sign up for our application. In the `UsersController` add the following:

```
public function initialize()
{
    parent::initialize();
    // Add the 'add' action to the allowed actions list.
    $this->Auth->allow(['logout', 'add']);
}
```

The above tells `AuthComponent` that the `add()` action does *not* require authentication or authorization. You may want to take the time to clean up the **Users/add.ctp** 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 so they will not work as `AuthComponent` will deny you access to those controller actions.

Restricting Bookmark Access

Now that users can log in, we'll want to limit the bookmarks they can see to the ones they made. We'll do this using an 'authorization' adapter. Since our requirements are pretty simple, we can write some simple code in our `BookmarksController`. But before we do that, we'll want to tell the AuthComponent how our application is going to authorize actions. In your `AppController` add the following:

```
public function isAuthorized($user)
{
    return false;
}
```

Also, add the following to the configuration for `Auth` in your `AppController`:

```
'authorize' => 'Controller',
```

Your `initialize()` method should now look like:

```
public function initialize()
{
    $this->loadComponent('Flash');
    $this->loadComponent('Auth', [
        'authorize'=> 'Controller',//added this line
        'authenticate' => [
            'Form' => [
                'fields' => [
                    'username' => 'email',
                    'password' => 'password'
                ]
            ]
        ],
```

```
        'loginAction' => [
            'controller' => 'Users',
            'action' => 'login'
        ],
        'unauthorizedRedirect' => $this->referer()
    ]);

    // Allow the display action so our pages controller
    // continues to work.
    $this->Auth->allow(['display']);
}
```

We'll default to denying access, and incrementally grant access where it makes sense. First, we'll add the authorization logic for bookmarks. In your `BookmarksController` add the following:

```
public function isAuthorized($user)
{
    $action = $this->request->getParam('action');

    // The add and index actions are always allowed.
    if (in_array($action, ['index', 'add', 'tags'])) {
        return true;
    }
    // All other actions require an id.
    if (!$this->request->getParam('pass.0')) {
        return false;
    }

    // Check that the bookmark belongs to the current user.
    $id = $this->request->getParam('pass.0');
    $bookmark = $this->Bookmarks->get($id);
    if ($bookmark->user_id == $user['id']) {
        return true;
    }
    return parent::isAuthorized($user);
}
```

Now if you try to view, edit or delete a bookmark that does not belong to you, you should be redirected back to the page you came from. If no error message is displayed, add the following to your layout:

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

You should now see the authorization error messages.

Fixing List view and Forms

While view and delete are working, edit, add and index have a few problems:

1. When adding a bookmark you can choose the user.
2. When editing a bookmark you can choose the user.
3. The list page shows bookmarks from other users.

Let's tackle the add form first. To begin with remove the `control('user_id')` from **src/Template/Bookmarks/add.ctp**. With that removed, we'll also update the `add()` action from **src/Controller/BookmarksController.php** to look like:

```
public function add()
{
    $bookmark = $this->Bookmarks->newEntity();
    if ($this->request->is('post')) {
        $bookmark = $this->Bookmarks->patchEntity($bookmark, $this->request->
↪getData());
        $bookmark->user_id = $this->Auth->user('id');
        if ($this->Bookmarks->save($bookmark)) {
            $this->Flash->success('The bookmark has been saved.');
            return $this->redirect(['action' => 'index']);
        }
        $this->Flash->error('The bookmark could not be saved. Please, try again.');
    }
    $tags = $this->Bookmarks->Tags->find('list');
    $this->set(compact('bookmark', 'tags'));
    $this->set('_serialize', ['bookmark']);
}
```

By setting the entity property with the session data, we remove any possibility of the user modifying which user a bookmark is for. We'll do the same for the edit form and action. Your `edit()` action from **src/Controller/BookmarksController.php** should look like:

```
public function edit($id = null)
{
    $bookmark = $this->Bookmarks->get($id, [
        'contain' => ['Tags']
    ]);
    if ($this->request->is(['patch', 'post', 'put'])) {
        $bookmark = $this->Bookmarks->patchEntity($bookmark, $this->request->
↪getData());
        $bookmark->user_id = $this->Auth->user('id');
        if ($this->Bookmarks->save($bookmark)) {
            $this->Flash->success('The bookmark has been saved.');
            return $this->redirect(['action' => 'index']);
        }
        $this->Flash->error('The bookmark could not be saved. Please, try again.');
    }
    $tags = $this->Bookmarks->Tags->find('list');
    $this->set(compact('bookmark', 'tags'));
    $this->set('_serialize', ['bookmark']);
}
```

List View

Now, we only need to show bookmarks for the currently logged in user. We can do that by updating the call to `paginate()`. Make your `index()` action from **src/Controller/BookmarksController.php** look like:

```
public function index()
{
    $this->paginate = [
        'conditions' => [
            'Bookmarks.user_id' => $this->Auth->user('id'),
        ]
    ];
    $this->set('bookmarks', $this->paginate($this->Bookmarks));
    $this->set('_serialize', ['bookmarks']);
```

```
}
```

We should also update the `tags()` action and the related finder method, but we'll leave that as an exercise you can complete on your own.

Improving the Tagging Experience

Right now, adding new tags is a difficult process, as the `TagsController` disallows all access. Instead of allowing access, 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/Bookmark.php** add the following:

```
use Cake\Collection\Collection;

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

This will let us access the `$bookmark->tag_string` computed property. We'll use this property in controls later on. Remember to add the `tag_string` property to the `_accessible` list in your entity, as we'll want to 'save' it later on.

In **src/Model/Entity/Bookmark.php** add the `tag_string` to `$_accessible` this way:

```
protected $_accessible = [
    'user_id' => true,
    'title' => true,
    'description' => true,
    'url' => true,
    'user' => true,
    'tags' => true,
    'tag_string' => true,
];
```

Updating the Views

With the entity updated we can add a new control for our tags. In **src/Template/Bookmarks/add.ctp** and **src/Template/Bookmarks/edit.ctp**, replace the existing `tags._ids` control with the following:

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

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/BookmarksTable.php**:

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

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

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

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

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.

Wrapping Up

We've expanded our bookmarking application to handle authentication and basic authorization/access control scenarios. We've also added some nice UX improvements by leveraging the FormHelper and ORM capabilities.

Thanks for taking the time to explore CakePHP. Next, you can complete the *Blog Tutorial*, learn more about the *Database Access & ORM*, or you can peruse the /topics.

Blog Tutorial

This tutorial will walk you through the creation of a simple blog application. We'll be installing CakePHP, creating a database, and creating enough application logic to list, add, edit, and delete blog articles.

Here's what you'll need:

1. A running web server. We're going to assume you're using Apache, though the instructions for using other servers should be very similar. We might have to play a little with the server configuration, but most folks can get CakePHP up and running without any configuration at all. Make sure you have PHP 5.6.0 or greater, and that the `mbstring` and `intl` extensions are enabled in PHP.
2. 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: CakePHP will be taking the reins from there. Since we're using MySQL, also make sure that you have `pdo_mysql` enabled in PHP.
3. Basic PHP knowledge.

Let's get started!

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, it's as easy as running the following:

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

Or, you can download `composer.phar` from the Composer website[45].

Then simply type the following line in your terminal from your installation directory to install the CakePHP application skeleton in the directory that you wish to use it with. For this example we will be using "blog" but feel free to change it to something else.:

```
php composer.phar create-project --prefer-dist cakephp/app blog
```

In case you've already got composer installed globally, you may instead type:

```
composer self-update && composer create-project --prefer-dist cakephp/app blog
```

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 something like the following:

```
/cake_install
    /bin
    /config
```

[45] https://getcomposer.org/download/

```
    /logs
    /plugins
    /src
    /tests
    /tmp
    /vendor
    /webroot
    .editorconfig
    .gitignore
    .htaccess
    .travis.yml
    composer.json
    index.php
    phpunit.xml.dist
    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.

Directory Permissions on tmp and logs

The `tmp` and `logs` directories need to have proper permissions to be writable by your webserver. If you used Composer for the install, this should have been done for you and confirmed with a "Permissions set on <folder>" message. If you instead got an error message or want to do it manually, the best way would be to find out what user your webserver runs as (`<?= `whoami`; ?>`) and change the ownership of these two directories to that user. The final command you run (in *nix) might look something like this:

```
chown -R www-data tmp
chown -R www-data logs
```

If for some reason CakePHP can't write to these directories, you'll be informed by a warning while not in production mode.

While not recommended, if you are unable to set the permissions to the same as your webserver, you can simply set write permissions on the folder by running a command such as:

```
chmod 777 -R tmp
chmod 777 -R logs
```

Creating the Blog Database

Next, let's set up the underlying MySQL database for our blog. If you haven't already done so, create an empty database for use in this tutorial, with a name of your choice, e.g. `cake_blog`. Right now, we'll just create a single table to store our articles. We'll also throw in a few articles to use for testing purposes. Execute the following SQL statements into your database:

```
/* First, create our articles table: */
CREATE TABLE articles (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(50),
    body TEXT,
    created DATETIME DEFAULT NULL,
    modified DATETIME DEFAULT NULL
);
```

```
/* Then insert some articles for testing: */
INSERT INTO articles (title,body,created)
    VALUES ('The title', 'This is the article body.', NOW());
INSERT INTO articles (title,body,created)
    VALUES ('A title once again', 'And the article body follows.', NOW());
INSERT INTO articles (title,body,created)
    VALUES ('Title strikes back', 'This is really exciting! Not.', NOW());
```

The choices on table and column names are not arbitrary. If you follow CakePHP's database naming conventions, and CakePHP's class naming conventions (both outlined in *CakePHP Conventions*), you'll be able to take advantage of a lot of free functionality and avoid configuration. CakePHP is flexible enough to accommodate even inconsistent legacy database schemas, but adhering to the conventions will save you time.

Check out *CakePHP Conventions* for more information, but it's suffice to say that naming our table 'articles' automatically hooks it to our Articles model, and having fields called 'modified' and 'created' will be automatically managed by CakePHP.

Database Configuration

Next, let's tell CakePHP where our database is and how to connect to it. For many, this will be the first and last time you will need to configure anything.

The configuration should be pretty straightforward: just replace the values in the `Datasources.default` array in the **config/app.php** file with those that apply to your setup. A sample completed configuration array might look something like the following:

```
return [
    // More configuration above.
    'Datasources' => [
        'default' => [
            'className' => 'Cake\Database\Connection',
            'driver' => 'Cake\Database\Driver\Mysql',
            'persistent' => false,
            'host' => 'localhost',
            'username' => 'cake_blog',
            'password' => 'AngelF00dC4k3~',
            'database' => 'cake_blog',
            'encoding' => 'utf8',
            'timezone' => 'UTC'
        ],
    ],
    // More configuration below.
];
```

Once you've saved your **config/app.php** file, you should be able to open your browser and see the CakePHP welcome page. It should also tell you that your database connection file was found, and that CakePHP can successfully connect to the database.

Note: A copy of CakePHP's default configuration file is found in **config/app.default.php**.

Optional Configuration

There are a few other items that can be configured. Most developers complete these laundry-list items, but they're not required for this tutorial. One is defining a custom string (or "salt") for use in security hashes.

The security salt is used for generating hashes. If you used Composer this too is taken care of for you during the install. Else you'd need to change the default salt value by editing **config/app.php**. It doesn't matter much what the new value is, as long as it's not guessable:

```
'Security' => [
    'salt' => 'something long and containing lots of different values.',
],
```

A Note on mod_rewrite

Occasionally new users will run into mod_rewrite issues. For example if the CakePHP welcome page looks a little funny (no images or CSS styles). This probably means mod_rewrite is not functioning on your system. Please refer to the *URL Rewriting* section to help resolve any issues you are having.

Now continue to *Blog Tutorial - Part 2* to start building your first CakePHP application.

Blog Tutorial - Part 2

Create an Article Model

Models are the bread and butter of CakePHP applications. By creating a CakePHP model that will interact with our database, we'll have the foundation in place needed to do our view, add, edit, and delete operations later.

CakePHP's model class files are split between `Table` and `Entity` objects. `Table` objects provide access to the collection of entities stored in a specific table and go 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:

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

namespace App\Model\Table;

use Cake\ORM\Table;

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

Naming conventions are very important in CakePHP. By naming our Table object `ArticlesTable`, CakePHP can automatically infer that this Table object will be used in the `ArticlesController`, and will be tied to a database table called `articles`.

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.

For more on models, such as callbacks, and validation, check out the *Database Access & ORM* chapter of the Manual.

Note: If you completed *Part 1 of the Blog Tutorial* and created the `articles` table in our Blog database you can leverage CakePHP's bake console and its code generation capabilities to create the `ArticlesTable` model:

```
bin/cake bake model Articles
```

For more on bake and its code generation features please visit *Code Generation with Bake*.

Create the Articles Controller

Next, we'll create a controller for our articles. The controller is where all interaction with articles will happen. In a nutshell, it's the place where you play with the business logic contained in the models and get work related to articles done. 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:

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

namespace App\Controller;

class ArticlesController extends AppController
{
}
```

Now, let's add an action to our controller. Actions often represent a single function or interface in an application. For example, when users request www.example.com/articles/index (which is also the same as www.example.com/articles/), they might expect to see a listing of articles. The code for that action would look like this:

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

namespace App\Controller;

class ArticlesController extends AppController
{

    public function index()
    {
        $articles = $this->Articles->find('all');
        $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.

> **Warning:** You may be tempted to name your controllers and actions a certain way to obtain a certain URL. Resist that temptation. Follow *CakePHP Conventions* (capitalization, plural names, etc.) and create readable, understandable action names. You can map URLs to your code using *Routing* covered later on.

The single instruction in the action uses `set()` to pass data from the controller to the view (which we'll create next). The line sets the view variable called 'articles' equal to the return value of the `find('all')` method of the

`ArticlesTable` object.

Note: If you completed *Part 1 of the Blog Tutorial* and created the `articles` table in your Blog database you can leverage CakePHP's bake console and its code generation capabilities to create the ArticlesController class:

```
bin/cake bake controller Articles
```

For more on bake and its code generation features please visit *Code Generation with Bake*.

To learn more about CakePHP's controllers, check out the *Controllers* chapter.

Creating Article Views

Now that we have our data flowing from our model, and our application logic is defined by our controller, let's create a view for the index action we created above.

CakePHP views are just presentation-flavored fragments that fit inside an application's layout. For most applications, they're HTML mixed with PHP, but they may end up as XML, CSV, or even binary data.

A layout is presentation code that is wrapped around a view. Multiple layouts can be defined, and you can switch between them, but for now, let's just use the default.

Remember in the last section how we assigned the 'articles' variable to the view using the `set()` method? That would hand down the query object collection to the view to be invoked with a `foreach` iteration.

CakePHP's template files are stored in **src/Template** inside a folder named after the controller they correspond to (we'll have to create a folder named 'Articles' in this case). To format this article data in a nice table, our view code might look something like this:

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

<h1>Blog articles</h1>
<table>
    <tr>
        <th>Id</th>
        <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><?= $article->id ?></td>
        <td>
            <?= $this->Html->link($article->title, ['action' => 'view', $article->
↪id]) ?>
        </td>
        <td>
            <?= $article->created->format(DATE_RFC850) ?>
        </td>
    </tr>
    <?php endforeach; ?>
</table>
```

Hopefully this should look somewhat simple.

You might have noticed the use of an object called `$this->Html`. This is an instance of the CakePHP *`Cake\View\Helper\HtmlHelper`* class. CakePHP comes with a set of view helpers that make things like linking, form output a snap. You can learn more about how to use them in *Helpers*, but what's important to note here is that the `link()` method will generate an HTML link with the given title (the first parameter) and URL (the second parameter).

When specifying URLs in CakePHP, it is recommended that you use the array format. This is explained in more detail in the section on Routes. Using the array format for URLs allows you to take advantage of CakePHP's reverse routing capabilities. You can also specify URLs relative to the base of the application in the form of `/controller/action/param1/param2` or use *named routes*.

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

If you happened to have clicked on one of the links we created in this view (that link a article's title to a URL `/articles/view/some\_id`), you were probably informed by CakePHP that the action hasn't yet been defined. If you were not so informed, either something has gone wrong, or you actually did define it already, in which case you are very sneaky. Otherwise, we'll create it in the `ArticlesController` now:

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

namespace App\Controller;

class ArticlesController extends AppController
{

    public function index()
    {
         $this->set('articles', $this->Articles->find('all'));
    }

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

The `set()` call should look familiar. Notice we're using `get()` rather than `find('all')` because we only really want a single article's information.

Notice that our view action takes a parameter: the ID of the article we'd like to see. This parameter is handed to the action through the requested URL. If a user requests `/articles/view/3`, then the value '3' is passed as `$id`.

We also do a bit of error checking to ensure a user is actually accessing a record. By using the `get()` function in the Articles table, we make sure the user has accessed a record that exists. In case the requested article is not present in the database, or the id is false the `get()` function will throw a `NotFoundException`.

Now let's create the view for our new 'view' action and place it in **src/Template/Articles/view.ctp**

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

<h1><?= h($article->title) ?></h1>
<p><?= h($article->body) ?></p>
<p><small>Created: <?= $article->created->format(DATE_RFC850) ?></small></p>
```

Verify that this is working by trying the links at `/articles/index` or manually requesting an article by accessing `/articles/view/{id}`, replacing `{id}` by an article 'id'.

Adding Articles

Reading from the database and showing us the articles is a great start, but let's allow for the adding of new articles.

First, start by creating an `add()` action in the `ArticlesController`:

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

namespace App\Controller;

use App\Controller\AppController;

class ArticlesController extends AppController
{

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

        $this->loadComponent('Flash'); // Include the FlashComponent
    }

    public function index()
    {
        $this->set('articles', $this->Articles->find('all'));
    }

    public function view($id)
    {
        $article = $this->Articles->get($id);
        $this->set(compact('article'));
    }

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

Note: You need to include the *Flash* component in any controller where you will use it. If necessary, include it in your `AppController`.

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 `ServerRequest` object which is accessible using `$this->request`. The

request object contains useful information regarding the request that was just received, and can be used to control the flow of your application. In this case, we use the *Cake\Http\ServerRequest::is()* method to check that the request is a HTTP POST request.

When a user uses a form to POST data to your application, that information 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.

We use FlashComponent's `success()` and `error()` methods to set a message to a session variable. These methods are provided using PHP's magic method features[46]. Flash messages will be displayed on the page after redirection. In the layout we have `<?= $this->Flash->render() ?>` which displays the message and clears the corresponding session variable. The controller's *Cake\Controller\Controller::redirect* function redirects to another URL. 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[47] to see the formats in which you can specify a URL for various CakePHP functions.

Calling the `save()` method will check for validation errors and abort the save if any occur. We'll discuss how those errors are handled in the following sections.

Data Validation

CakePHP goes a long way toward taking the monotony out of form input validation. Everyone hates coding up endless forms and their validation routines. CakePHP makes it easier and faster.

To take advantage of the validation features, you'll need to use CakePHP's *Form* helper in your views. The *Cake\View\Helper\FormHelper* is available by default to all views at `$this->Form`.

Here's our add view:

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

<h1>Add Article</h1>
<?php
    echo $this->Form->create($article);
    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">
```

If `create()` is called with no parameters supplied, it assumes you are building a form that submits via POST to the current controller's `add()` action (or `edit()` action when `id` is included in the form data).

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 automagic here: `control()` will output different form elements based on the model field specified.

The `$this->Form->end()` call ends the form. Outputting hidden inputs if CSRF/Form Tampering prevention is enabled.

[46] http://php.net/manual/en/language.oop5.overloading.php#object.call
[47] https://api.cakephp.org

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

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

You may be wondering: how do I tell CakePHP about my validation requirements? Validation rules are defined in the model. Let's look back at our Articles model and make a few adjustments:

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

namespace App\Model\Table;

use Cake\ORM\Table;
use Cake\Validation\Validator;

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

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

        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 body and title fields must not be empty, and are required for both create and update operations. CakePHP's validation engine is strong, with a number of pre-built rules (credit card numbers, email addresses, etc.) and 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.

Editing Articles

Post editing: here we go. You're a CakePHP pro by now, so you should have picked up a pattern. Make the action, then the view. Here's what the `edit()` action of the `ArticlesController` would look like:

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

public function edit($id = null)
{
    $article = $this->Articles->get($id);
    if ($this->request->is(['post', 'put'])) {
        $this->Articles->patchEntity($article, $this->request->getData());
        if ($this->Articles->save($article)) {
```

```
            $this->Flash->success(__('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 `$id` parameter, or the article does not exist, we throw a `NotFoundException` for the CakePHP ErrorHandler to take care of.

Next the action checks whether the request is either a POST or a PUT request. If it is, then we use the POST data to update our article entity by using the `patchEntity()` method. Finally we use the table object to save the entity back or kick back and show the user validation errors.

The edit view might look something like this:

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

<h1>Edit Article</h1>
<?php
    echo $this->Form->create($article);
    echo $this->Form->control('title');
    echo $this->Form->control('body', ['rows' => '3']);
    echo $this->Form->button(__('Save Article'));
    echo $this->Form->end();
?>
```

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

CakePHP will determine whether a `save()` generates an insert or an update statement based on the state of the entity.

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

```
<!-- File: src/Template/Articles/index.ctp  (edit links added) -->

<h1>Blog articles</h1>
<p><?= $this->Html->link("Add Article", ['action' => 'add']) ?></p>
<table>
    <tr>
        <th>Id</th>
        <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><?= $article->id ?></td>
        <td>
            <?= $this->Html->link($article->title, ['action' => 'view', $article->
↪id]) ?>
        </td>
        <td>
            <?= $article->created->format(DATE_RFC850) ?>
```

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

</table>
```

Deleting Articles

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

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

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

    $article = $this->Articles->get($id);
    if ($this->Articles->delete($article)) {
        $this->Flash->success(__('The article with id: {0} has been deleted.', h(
↪$id)));
        return $this->redirect(['action' => 'index']);
    }
}
```

This logic deletes the article specified by `$id`, and uses `$this->Flash->success()` to show the user a confirmation message after redirecting them on to `/articles`. If the user attempts to do a delete using a GET request, the `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.

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

```
<!-- File: src/Template/Articles/index.ctp (delete links added) -->

<h1>Blog articles</h1>
<p><?= $this->Html->link('Add Article', ['action' => 'add']) ?></p>
<table>
    <tr>
        <th>Id</th>
        <th>Title</th>
        <th>Created</th>
        <th>Actions</th>
    </tr>

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

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

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

</table>
```

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

Warning: Allowing content to be deleted using GET requests is dangerous, as web crawlers could accidentally delete all your content.

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

Routes

For some, CakePHP's default routing works well enough. Developers who are sensitive to user-friendliness and general search engine compatibility will appreciate the way that CakePHP's URLs map to specific actions. So we'll just make a quick change to routes in this tutorial.

For more information on advanced routing techniques, see *Connecting Routes*.

By default, CakePHP responds to a request for the root of your site (e.g., http://www.example.com) using its `PagesController`, rendering a view called "home". Instead, we'll replace this with our ArticlesController by creating a routing rule.

CakePHP's routing is found in **config/routes.php**. You'll want to comment out or remove the line that defines the default root route. It looks like this:

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

This line connects the URL '/' with the default CakePHP home page. We want it to connect with our own controller, so replace that line with this one:

```
$routes->connect('/', ['controller' => 'Articles', 'action' => 'index']);
```

This should connect users requesting '/' to the `index()` action of our `ArticlesController`.

Note: CakePHP also makes use of 'reverse routing'. If, with the above route defined, you pass `['controller' => 'Articles','action' => 'index']` to a function expecting an array, the resulting URL used will be

'/'. It's therefore a good idea to always use arrays for URLs as this means your routes define where a URL goes, and also ensures that links point to the same place.

Conclusion

Creating applications this way will win you peace, honor, love, and money beyond even your wildest fantasies. Simple, isn't it? Keep in mind that this tutorial was very basic. CakePHP has *many* more features to offer, and is flexible in ways we didn't wish to cover here for simplicity's sake. Use the rest of this manual as a guide for building more feature-rich applications.

Now that you've created a basic CakePHP application, you can either continue to *Blog Tutorial - Part 3*, or start your own project. You can also peruse the /topics or API[48] to learn more about CakePHP.

If you need help, there are many ways to get the help you need - please see the *Where to Get Help* page. Welcome to CakePHP!

Suggested Follow-up Reading

These are common tasks people learning CakePHP usually want to study next:

1. *Layouts*: Customizing your website layout
2. *Elements*: Including and reusing view snippets
3. *Code Generation with Bake*: Generating basic CRUD code
4. *Blog Tutorial - Authentication and Authorization*: User authentication and authorization tutorial

Blog Tutorial - Part 3

Create a Tree Category

Let's continue our blog application and imagine we want to categorize our articles. We want the categories to be ordered, and for this, we will use the *Tree behavior* to help us organize the categories.

But first, we need to modify our tables.

Migrations Plugin

We will use the migrations plugin[49] to create a table in our database. If you already have an articles table in your database, erase it.

Now open your application's **composer.json** file. Normally you would see that the migrations plugin is already under `require`. If not, add it by executing:

```
composer require cakephp/migrations:~1.0
```

The migrations plugin will now be in your application's **plugins** folder. Also, add `Plugin::load('Migrations');` to your application's **bootstrap.php** file.

Once the plugin is loaded, run the following command to create a migration file:

[48] https://api.cakephp.org
[49] https://github.com/cakephp/migrations

```
bin/cake bake migration CreateArticles title:string body:text category_id:integer␣
↪created modified
```

A migration file will be generated in the **/config/Migrations** folder with the following:

```
<?php

use Migrations\AbstractMigration;

class CreateArticles extends AbstractMigration
{
    public function change()
    {
        $table = $this->table('articles');
        $table->addColumn('title', 'string', [
            'default' => null,
            'limit' => 255,
            'null' => false,
        ]);
        $table->addColumn('body', 'text', [
            'default' => null,
            'null' => false,
        ]);
        $table->addColumn('category_id', 'integer', [
            'default' => null,
            'limit' => 11,
            'null' => false,
        ]);
        $table->addColumn('created', 'datetime', [
            'default' => null,
            'null' => false,
        ]);
        $table->addColumn('modified', 'datetime', [
            'default' => null,
            'null' => false,
        ]);
        $table->create();
    }
}
```

Run another command to create a `categories` table. If you need to specify a field length, you can do it within brackets in the field type, ie:

```
bin/cake bake migration CreateCategories parent_id:integer lft:integer[10] rght:
↪integer[10] name:string[100] description:string created modified
```

This will generate the following file in **config/Migrations**:

```
<?php

use Migrations\AbstractMigration;

class CreateCategories extends AbstractMigration
{
    public function change()
    {
        $table = $this->table('categories');
        $table->addColumn('parent_id', 'integer', [
```

```
            'default' => null,
            'limit' => 11,
            'null' => false,
        ]);
        $table->addColumn('lft', 'integer', [
            'default' => null,
            'limit' => 10,
            'null' => false,
        ]);
        $table->addColumn('rght', 'integer', [
            'default' => null,
            'limit' => 10,
            'null' => false,
        ]);
        $table->addColumn('name', 'string', [
            'default' => null,
            'limit' => 100,
            'null' => false,
        ]);
        $table->addColumn('description', 'string', [
            'default' => null,
            'limit' => 255,
            'null' => false,
        ]);
        $table->addColumn('created', 'datetime', [
            'default' => null,
            'null' => false,
        ]);
        $table->addColumn('modified', 'datetime', [
            'default' => null,
            'null' => false,
        ]);
        $table->create();
    }
}
```

Now that the migration files are created, you can edit them before creating your tables. We need to change the `'null' => false` for the `parent_id` field with `'null' => true` because a top-level category has a null `parent_id`.

Run the following command to create your tables:

```
bin/cake migrations migrate
```

Modifying the Tables

With our tables set up, we can now focus on categorizing our articles.

We suppose you already have the files (Tables, Controllers and Templates of Articles) from part 2. So we'll just add the references to categories.

We need to associate the Articles and Categories tables together. Open the **src/Model/Table/ArticlesTable.php** file and add the following:

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

namespace App\Model\Table;
```

```
use Cake\ORM\Table;

class ArticlesTable extends Table
{
    public function initialize(array $config)
    {
        $this->addBehavior('Timestamp');
        // Just add the belongsTo relation with CategoriesTable
        $this->belongsTo('Categories', [
            'foreignKey' => 'category_id',
        ]);
    }
}
```

Generate Skeleton Code for Categories

Create all files by launching bake commands:

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

Alternatively, you can bake all with just one line:

```
bin/cake bake all Categories
```

The bake tool has created all your files in a snap. You can give them a quick read if you want re-familiarize yourself with how CakePHP works.

Note: If you are on Windows remember to use \ instead of /.

You'll need to edit the following in **src/Template/Categories/add.ctp** and **src/Template/Categories/edit.ctp**:

```
echo $this->Form->control('parent_id', [
    'options' => $parentCategories,
    'empty' => 'No parent category'
]);
```

Attach TreeBehavior to CategoriesTable

The *TreeBehavior* helps you manage hierarchical Tree structures in database table. It uses the MPTT logic[50] to manage the data. MPTT tree structures are optimized for reads, which often makes them a good fit for read heavy applications like blogs.

If you open the **src/Model/Table/CategoriesTable.php** file, you'll see that the TreeBehavior has been attached to your CategoriesTable in the `initialize()` method. Bake adds this behavior to any Tables that contain `lft` and `rght` columns:

```
$this->addBehavior('Tree');
```

[50] http://www.sitepoint.com/hierarchical-data-database-2/

With the TreeBehavior attached you'll be able to access some features like reordering the categories. We'll see that in a moment.

But for now, you have to remove the following controls in your Categories add and edit template files:

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

In addition you should disable or remove the requirePresence from the validator for both the `lft` and `rght` columns in your CategoriesTable model:

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

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

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

These fields are automatically managed by the TreeBehavior when a category is saved.

Using your web browser, add some new categories using the `/yoursite/categories/add` controller action.

Reordering Categories with TreeBehavior

In your categories index template file, you can list the categories and re-order them.

Let's modify the index method in your **CategoriesController.php** and add `moveUp()` and `moveDown()` methods to be able to reorder the categories in the tree:

```
class CategoriesController extends AppController
{
    public function index()
    {
        $categories = $this->Categories->find()
            ->order(['lft' => 'ASC']);
        $this->set(compact('categories'));
        $this->set('_serialize', ['categories']);
    }

    public function moveUp($id = null)
    {
        $this->request->allowMethod(['post', 'put']);
        $category = $this->Categories->get($id);
        if ($this->Categories->moveUp($category)) {
            $this->Flash->success('The category has been moved Up.');
        } else {
            $this->Flash->error('The category could not be moved up. Please, try␣
↪again.');
```

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

    public function moveDown($id = null)
    {
        $this->request->allowMethod(['post', 'put']);
        $category = $this->Categories->get($id);
        if ($this->Categories->moveDown($category)) {
            $this->Flash->success('The category has been moved down.');
        } else {
            $this->Flash->error('The category could not be moved down. Please, try
↪again.');
        }
        return $this->redirect($this->referer(['action' => 'index']));
    }
}
```

In **src/Template/Categories/index.ctp** replace the existing content with:

```
<div class="actions large-2 medium-3 columns">
    <h3><?= __('Actions') ?></h3>
    <ul class="side-nav">
        <li><?= $this->Html->link(__('New Category'), ['action' => 'add']) ?></li>
    </ul>
</div>
<div class="categories index large-10 medium-9 columns">
    <table cellpadding="0" cellspacing="0">
    <thead>
        <tr>
            <th>Id</th>
            <th>Parent Id</th>
            <th>Lft</th>
            <th>Rght</th>
            <th>Name</th>
            <th>Description</th>
            <th>Created</th>
            <th class="actions"><?= __('Actions') ?></th>
        </tr>
    </thead>
    <tbody>
    <?php foreach ($categories as $category): ?>
        <tr>
            <td><?= $category->id ?></td>
            <td><?= $category->parent_id ?></td>
            <td><?= $category->lft ?></td>
            <td><?= $category->rght ?></td>
            <td><?= h($category->name) ?></td>
            <td><?= h($category->description) ?></td>
            <td><?= h($category->created) ?></td>
            <td class="actions">
                <?= $this->Html->link(__('View'), ['action' => 'view', $category->
↪id]) ?>
                <?= $this->Html->link(__('Edit'), ['action' => 'edit', $category->
↪id]) ?>
                <?= $this->Form->postLink(__('Delete'), ['action' => 'delete',
↪$category->id], ['confirm' => __('Are you sure you want to delete # {0}?',
↪$category->id)]) ?>
```

```
                <?= $this->Form->postLink(__('Move down'), ['action' => 'moveDown',
↪$category->id], ['confirm' => __('Are you sure you want to move down # {0}?',
↪$category->id)]) ?>
                <?= $this->Form->postLink(__('Move up'), ['action' => 'moveUp',
↪$category->id], ['confirm' => __('Are you sure you want to move up # {0}?',
↪$category->id)]) ?>
            </td>
        </tr>
    <?php endforeach; ?>
    </tbody>
    </table>
</div>
```

Modifying the ArticlesController

In our `ArticlesController`, we'll get the list of all the categories. This will allow us to choose a category for an Article when creating or editing it:

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

namespace App\Controller;

use Cake\Network\Exception\NotFoundException;

class ArticlesController extends AppController
{

    // ...

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

        // Just added the categories list to be able to choose
        // one category for an article
        $categories = $this->Articles->Categories->find('treeList');
        $this->set(compact('categories'));
    }
}
```

Modifying the Articles Templates

The article add file should look something like this:

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

<h1>Add Article</h1>
<?php
echo $this->Form->create($article);
// just added the categories control
echo $this->Form->control('category_id');
echo $this->Form->control('title');
echo $this->Form->control('body', ['rows' => '3']);
echo $this->Form->button(__('Save Article'));
echo $this->Form->end();
```

When you go to the address */yoursite/articles/add* you should see a list of categories to choose.

Blog Tutorial - Authentication and Authorization

Following our *Blog Tutorial* example, imagine we wanted to secure access to certain URLs, based on the logged-in user. We also have another requirement: to allow our blog to have multiple authors who can create, edit, and delete their own articles while disallowing other authors from making changes to articles they do not own.

Creating All User-Related Code

First, let's create a new table in our blog database to hold our users' data:

```
CREATE TABLE users (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50),
    password VARCHAR(255),
    role VARCHAR(20),
    created DATETIME DEFAULT NULL,
    modified DATETIME DEFAULT NULL
);
```

We have adhered to the CakePHP conventions in naming tables, but we're also taking advantage of another convention: By using the username and password columns in a users table, CakePHP will be able to auto-configure most things for us when implementing the user login.

Next step is to create our UsersTable class, responsible for finding, saving and validating any user data:

```
// src/Model/Table/UsersTable.php
namespace App\Model\Table;

use Cake\ORM\Table;
use Cake\Validation\Validator;

class UsersTable extends Table
{

    public function validationDefault(Validator $validator)
    {
        return $validator
            ->notEmpty('username', 'A username is required')
            ->notEmpty('password', 'A password is required')
            ->notEmpty('role', 'A role is required')
```

```
            ->add('role', 'inList', [
                'rule' => ['inList', ['admin', 'author']],
                'message' => 'Please enter a valid role'
            ]);
    }

}
```

Let's also create our UsersController. The following content corresponds to parts of a basic baked UsersController class using the code generation utilities bundled with CakePHP:

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

namespace App\Controller;

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

class UsersController extends AppController
{

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

     public function index()
     {
        $this->set('users', $this->Users->find('all'));
    }

    public function view($id)
    {
        $user = $this->Users->get($id);
        $this->set(compact('user'));
    }

    public function add()
    {
        $user = $this->Users->newEntity();
        if ($this->request->is('post')) {
            $user = $this->Users->patchEntity($user, $this->request->getData());
            if ($this->Users->save($user)) {
                $this->Flash->success(__('The user has been saved.'));
                return $this->redirect(['action' => 'add']);
            }
            $this->Flash->error(__('Unable to add the user.'));
        }
        $this->set('user', $user);
    }

}
```

In the same way we created the views for our articles by using the code generation tool, we can implement the user views. For the purpose of this tutorial, we will show just the add.ctp:

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

<div class="users form">
<?= $this->Form->create($user) ?>
    <fieldset>
        <legend><?= __('Add User') ?></legend>
        <?= $this->Form->control('username') ?>
        <?= $this->Form->control('password') ?>
        <?= $this->Form->control('role', [
            'options' => ['admin' => 'Admin', 'author' => 'Author']
        ]) ?>
   </fieldset>
<?= $this->Form->button(__('Submit')); ?>
<?= $this->Form->end() ?>
</div>
```

Authentication (Login and Logout)

We're now ready to add our authentication layer. In CakePHP this is handled by the `Cake\Controller\Component\AuthComponent`, a class responsible for requiring login for certain actions, handling user login and logout, and also authorizing logged-in users to the actions they are allowed to reach.

To add this component to your application open your **src/Controller/AppController.php** file and add the following lines:

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

namespace App\Controller;

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

class AppController extends Controller
{
    //...

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

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

```
    //...
}
```

There is not much to configure, as we used the conventions for the users table. We just set up the URLs that will be loaded after the login and logout actions is performed, in our case to `/articles/` and `/` respectively.

What we did in the `beforeFilter()` function was to tell the AuthComponent to not require a login for all `index()` and `view()` actions, in every controller. We want our visitors to be able to read and list the entries without registering in the site.

Now, we need to be able to register new users, save their username and password, and more importantly, hash their password so it is not stored as plain text in our database. Let's tell the AuthComponent to let un-authenticated users access the users add function and implement the login and logout action:

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

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

class UsersController extends AppController
{
    // Other methods..

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

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

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

Password hashing is not done yet, we need an Entity class for our User in order to handle its own specific logic. Create the **src/Model/Entity/User.php** entity file and add the following:

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

use Cake\Auth\DefaultPasswordHasher;
```

```
use Cake\ORM\Entity;

class User extends Entity
{

    // Make all fields mass assignable except for primary key field "id".
    protected $_accessible = [
        '*' => true,
        'id' => false
    ];

    // ...

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

    // ...
}
```

Now every time the password property is assigned to the user it will be hashed using the `DefaultPasswordHasher` class. We're just missing a template view file for the login function. Open up your **src/Template/Users/login.ctp** file and add the following lines:

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

<div class="users form">
<?= $this->Flash->render() ?>
<?= $this->Form->create() ?>
    <fieldset>
        <legend><?= __('Please enter your username and password') ?></legend>
        <?= $this->Form->control('username') ?>
        <?= $this->Form->control('password') ?>
    </fieldset>
<?= $this->Form->button(__('Login')); ?>
<?= $this->Form->end() ?>
</div>
```

You can now register a new user by accessing the `/users/add` URL and log in with the newly created credentials by going to `/users/login` URL. Also, try to access any other URL that was not explicitly allowed such as `/articles/add`, you will see that the application automatically redirects you to the login page.

And that's it! It looks too simple to be true. Let's go back a bit to explain what happened. The `beforeFilter()` function is telling the AuthComponent to not require a login for the `add()` action in addition to the `index()` and `view()` actions that were already allowed in the AppController's `beforeFilter()` function.

The `login()` action calls the `$this->Auth->identify()` function in the AuthComponent, and it works without any further config because we are following conventions as mentioned earlier. That is, having a Users table with a username and a password column, and use a form posted to a controller with the user data. This function returns whether the login was successful or not, and in the case it succeeds, then we redirect the user to the configured redirection URL that we used when adding the AuthComponent to our application.

The logout works by just accessing the `/users/logout` URL and will redirect the user to the configured logoutUrl formerly described. This URL is the result of the `AuthComponent::logout()` function on success.

Authorization (who's allowed to access what)

As stated before, we are converting this blog into a multi-user authoring tool, and in order to do this, we need to modify the articles table a bit to add the reference to the Users table:

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

Also, a small change in the ArticlesController is required to store the currently logged in user as a reference for the created article:

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

public function add()
{
    $article = $this->Articles->newEntity();
    if ($this->request->is('post')) {
        $article = $this->Articles->patchEntity($article, $this->request->getData());
        // Added this line
        $article->user_id = $this->Auth->user('id');
        // You could also do the following
        //$newData = ['user_id' => $this->Auth->user('id')];
        //$article = $this->Articles->patchEntity($article, $newData);
        if ($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);

    // Just added the categories list to be able to choose
    // one category for an article
    $categories = $this->Articles->Categories->find('treeList');
    $this->set(compact('categories'));
}
```

The `user()` function provided by the component returns any column from the currently logged in user. We used this method to add the data into the request info that is saved.

Let's secure our app to prevent some authors from editing or deleting the others' articles. Basic rules for our app are that admin users can access every URL, while normal users (the author role) can only access the permitted actions. Again, open the AppController class and add a few more options to the Auth config:

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

public function initialize()
{
    $this->loadComponent('Flash');
    $this->loadComponent('Auth', [
        'authorize' => ['Controller'], // Added this line
        'loginRedirect' => [
            'controller' => 'Articles',
            'action' => 'index'
        ],
        'logoutRedirect' => [
            'controller' => 'Pages',
            'action' => 'display',
            'home'
```

```
            ]
        ]);
    }

    public function isAuthorized($user)
    {
        // Admin can access every action
        if (isset($user['role']) && $user['role'] === 'admin') {
            return true;
        }

        // Default deny
        return false;
    }
```

We just created a simple authorization mechanism. Users with the `admin` role will be able to access any URL in the site when logged-in. All other users – those with the `author` role – will have the same access as users who aren't logged-in.

This is not exactly what we want. We need to supply more rules to our `isAuthorized()` method. However instead of doing it in AppController, we'll delegate supplying those extra rules to each individual controller. The rules we're going to add to ArticlesController should permit authors to create articles but prevent authors from editing articles they do not own. Add the following content to your **ArticlesController.php**:

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

public function isAuthorized($user)
{
    // All registered users can add articles
    if ($this->request->getParam('action') === 'add') {
        return true;
    }

    // The owner of an article can edit and delete it
    if (in_array($this->request->getParam('action'), ['edit', 'delete'])) {
        $articleId = (int)$this->request->getParam('pass.0');
        if ($this->Articles->isOwnedBy($articleId, $user['id'])) {
            return true;
        }
    }

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

We're now overriding the AppController's `isAuthorized()` call and internally checking if the parent class is already authorizing the user. If he isn't, then just allow him to access the add action, and conditionally access edit and delete. One final thing has not been implemented. To tell whether or not the user is authorized to edit the article, we're calling a `isOwnedBy()` function in the Articles table. Let's then implement that function:

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

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

This concludes our simple authentication and authorization tutorial. For securing the UsersController you can follow the same technique we did for ArticlesController. You could also be more creative and code something more general

in AppController based on your own rules.

Should you need more control, we suggest you read the complete Auth guide in the *Authentication* section where you will find more about configuring the component, creating custom Authorization classes, and much more.

Suggested Follow-up Reading

1. *Code Generation with Bake* Generating basic CRUD code
2. *Authentication*: User registration and login

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 have got an up to date PHP version:

```
php -v
```

You should at least have got installed PHP 5.6.0 (CLI) or higher. Your webserver's PHP version must also be of 5.6.0 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, it's as easy as running the following:

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

Or, you can download `composer.phar` from the Composer website[51].

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 cms
```

If you downloaded and ran the Composer Windows Installer[52], then type the following line in your terminal from your installation directory (ie. C:\wamp\www\dev\cakephp3):

```
composer self-update && composer create-project --prefer-dist cakephp/app cms
```

[51] https://getcomposer.org/download/
[52] https://getcomposer.org/Composer-Setup.exe

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 something like the following:

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

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 checkmarks 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 and create our first model*.

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 a name of your choice, e.g. `cake_cms`. You can execute the following SQL to create the necessary tables:

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

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

CREATE TABLE tags (
    id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(191),
    created DATETIME,
    modified DATETIME,
    UNIQUE KEY (title)
) CHARSET=utf8mb4;

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 bookmark_key(article_id) REFERENCES articles(id)
);

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

You may have noticed that the `articles_tags` table used a composite primary key. CakePHP supports composite primary keys almost everywhere, making it easier to build multi-tenanted applications.

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.php** file with those that apply to your setup. A sample com-

pleted configuration array might look something like the following:

```
<?php
return [
    // More configuration above.
    'Datasources' => [
        'default' => [
            'className' => 'Cake\Database\Connection',
            'driver' => 'Cake\Database\Driver\Mysql',
            'persistent' => false,
            'host' => 'localhost',
            'username' => 'cakephp',
            'password' => 'AngelF00dC4k3~',
            'database' => 'cake_cms',
            'encoding' => 'utf8mb4',
            'timezone' => 'UTC',
            'cacheMetadata' => true,
        ],
    ],
    // More configuration below.
];
```

Once you've saved your **config/app.php** file, you should see that 'CakePHP is able to connect to the database' section have a checkmark.

Note: A copy of CakePHP's default configuration file is found in **config/app.default.php**.

Creating our First Model

Models are the heart of a 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 build the foundations necessary to build 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
namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{
    public function initialize(array $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
namespace App\Model\Entity;

use Cake\ORM\Entity;

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

Our entity is quite slim right now, and we've only setup the `_accessible` property which controls how properties can be modified by *Mass Assignment*.

We can't do much with our models right now, so 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()
    {
        $this->loadComponent('Paginator');
        $articles = $this->Paginator->paginate($this->Articles->find());
        $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 **src/Template** 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: src/Template/Articles/index.ctp -->

<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 easy. 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 use 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 **src/Template/Articles/view.ctp**

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

<h1><?= h($article->title) ?></h1>
<p><?= h($article->body) ?></p>
<p><small>Created: <?= $article->created->format(DATE_RFC850) ?></small></p>
<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/slug-name`.

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`:

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

namespace App\Controller;

use App\Controller\AppController;

class ArticlesController extends AppController
{

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

        $this->loadComponent('Flash'); // Include the FlashComponent
    }

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

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

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

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[53]. 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[54] 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: src/Template/Articles/add.ctp -->

<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 **src/Template/Articles/index.ctp** view to include a new "Add Article" link. Before the `<table>`, add the following line:

[53] http://php.net/manual/en/language.oop5.overloading.php#object.call
[54] https://api.cakephp.org

```
<?= $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:

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

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

// Add the following method.

public function beforeSave($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 is temporary, and will be removed later
    // when we build authentication out.
    if (!$entity->user_id) {
        $entity->user_id = 1;
    }
}
```

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

<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: src/Template/Articles/index.ctp  (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
        ->notEmpty('title')
        ->minLength('title', 10)
        ->maxLength('title', 255)

        ->notEmpty('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

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: src/Template/Articles/index.ctp  (deleted 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 *View\Helper\FormHelper::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.

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

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 Password Hashing

If you were to create/update a user at this point in time, 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 separate the methods that operate on a collection of objects, and a single object into different classes. 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 convention based setter methods 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 Cake\Auth\DefaultPasswordHasher; // Add this line
use Cake\ORM\Entity;

class User extends Entity
{

    // Code from bake.
```

```
    // Add this method
    protected function _setPassword($value)
    {
        if (strlen($value)) {
            $hasher = new DefaultPasswordHasher();

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

Now, point your browser to **http://localhost:8765/users** to see a list of users. 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[55] by default. You can also use SHA-1 or MD5 if you're working with an existing database, but we recommend bcrypt for all new applications.

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)
{
    $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
{
```

[55] http://codahale.com/how-to-safely-store-a-password/

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

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

```
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');

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

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

Remember to add the new tags mutliple select control we added to the **add.ctp** template to the **src/Template/Articles/edit.ctp** 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** should look like:

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

Router::defaultRouteClass(DashedRoute::class);

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

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

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

Plugin::routes();
```

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:

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

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
    ]);

    // 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
    ]);

    // 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:

```
// 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(Query $query, array $options)
{
    $columns = [
        'Articles.id', 'Articles.user_id', 'Articles.title',
        'Articles.body', 'Articles.published', 'Articles.created',
        'Articles.slug',
    ];

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

```
    if (empty($options['tags'])) {
        // 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' => $options['tags']]);
    }

    return $query->group(['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 bookmarks 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 **src/Template/Articles/tags.ctp** put the following content:

```
<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) ?>
    </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.ctp** 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;

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

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 **src/Template/Articles/add.ctp** and **src/Template/Articles/edit.ctp**, replace the existing `tags._ids` control with the following:

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

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

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

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.

Next we'll be adding *authentication*.

CMS Tutorial - Authentication

Now that our CMS has users, we should enable them to login, and apply some basic access control to the article creation & editing experiences.

Adding Login

In CakePHP, authentication is handled by *Components*. Components can be thought of as ways to create reusable chunks of controller code related to a specific feature or concept. Components can hook into the controller's event life-cycle and interact with your application that way. To get started, we'll add the *AuthComponent* to our application. We'll want the create, update and delete methods to require authentication, so we'll add AuthComponent in our AppController:

```
// In src/Controller/AppController.php
namespace App\Controller;

use Cake\Controller\Controller;

class AppController extends Controller
{
    public function initialize()
    {
        // Existing code

        $this->loadComponent('Flash');
        $this->loadComponent('Auth', [
            'authenticate' => [
                'Form' => [
                    'fields' => [
                        'username' => 'email',
                        'password' => 'password'
                    ]
                ]
            ],
            'loginAction' => [
                'controller' => 'Users',
                'action' => 'login'
            ],
             // If unauthorized, return them to page they were just on
            'unauthorizedRedirect' => $this->referer()
        ]);

        // Allow the display action so our pages controller
        // continues to work. Also enable the read only actions.
        $this->Auth->allow(['display', 'view', 'index']);
    }
}
```

We've just told CakePHP that we want to load the `Flash` and `Auth` components. In addition, we've customized the configuration of AuthComponent, as our users table uses `email` as the username. Now, if you go any protected URL, such as `/articles/add`, you'll be redirected to **/users/login**, which will show an error page as we have not written that code yet. So let's create the login action:

```
// In src/Controller/UsersController.php
public function login()
{
    if ($this->request->is('post')) {
        $user = $this->Auth->identify();
        if ($user) {
            $this->Auth->setUser($user);
            return $this->redirect($this->Auth->redirectUrl());
        }
        $this->Flash->error('Your username or password is incorrect.');
    }
}
```

And in **src/Template/Users/login.ctp** add the following:

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

```
<?= $this->Form->control('password') ?>
<?= $this->Form->button('Login') ?>
<?= $this->Form->end() ?>
```

Note: The `control()` method is available since 3.4. For prior versions you can use the `input()` method instead.

Now that we have a simple login form, we should be able to log in with one of the users that has a hashed password.

Note: If none of your users have hashed passwords, comment the `loadComponent('Auth')` block and `$this->Auth->allow()` calls. Then go and edit the user, saving a new password for them. After saving a new password for the user, make sure to uncomment the lines we just temporarily commented!

Try it out! Before logging in, visit `/articles/add`. 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`.

Adding Logout

Now that people can log in, you'll probably want to provide a way to log out as well. Again, in the `UsersController`, add the following code:

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

public function logout()
{
    $this->Flash->success('You are now logged out.');
    return $this->redirect($this->Auth->logout());
}
```

This code adds the `logout` action to the list of actions that do not require authentication and implements the logout method. Now you can visit `/users/logout` to log out. You should then be sent to the login page.

Enabling Registrations

If you aren't logged in and you try to visit **/users/add** 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` add the following:

```
public function initialize()
{
    parent::initialize();
    // Add the 'add' action to the allowed actions list.
    $this->Auth->allow(['logout', 'add']);
}
```

The above tells `AuthComponent` 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.ctp** 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.

Restricting Article Access

Now that users can log in, we'll want to limit users to only edit articles that they created. We'll do this using an 'authorization' adapter. Since our requirements are pretty simple, we can write some simple code in our `ArticlesController`. But before we do that, we'll want to tell the `AuthComponent` how our application is going to authorize actions. Update your `AppController` adding the following:

```
public function isAuthorized($user)
{
    // By default deny access.
    return false;
}
```

Next we'll tell `AuthComponent` that we want to use controller hook methods for authorization. Your `AppController::initialize()` method should now look like:

```
public function initialize()
{
    // Existing code

    $this->loadComponent('Flash');
    $this->loadComponent('Auth', [
        // Added this line
        'authorize'=> 'Controller',
        'authenticate' => [
            'Form' => [
                'fields' => [
                    'username' => 'email',
                    'password' => 'password'
                ]
            ]
        ],
        'loginAction' => [
            'controller' => 'Users',
            'action' => 'login'
        ],
         // If unauthorized, return them to page they were just on
        'unauthorizedRedirect' => $this->referer()
    ]);

    // Allow the display action so our pages controller
    // continues to work. Also enable the read only actions.
    $this->Auth->allow(['display', 'view', 'index']);
}
```

We'll default to denying access, and incrementally grant access where it makes sense. First, we'll add the authorization logic for articles. In your `ArticlesController` add the following:

```
public function isAuthorized($user)
{
    $action = $this->request->getParam('action');
    // The add and tags actions are always allowed to logged in users.
    if (in_array($action, ['add', 'tags'])) {
        return true;
    }

    // All other actions require a slug.
    $slug = $this->request->getParam('pass.0');
```

```
    if (!$slug) {
        return false;
    }

    // Check that the article belongs to the current user.
    $article = $this->Articles->findBySlug($slug)->first();

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

Now if you try to edit or delete an article that does not belong to you, you should be redirected back to the page you came from. If no error message is displayed, add the following to your layout:

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

While the above is fairly simplistic it illustrates how you could build more complex logic that combines the current user and request data to build flexible authorization logic.

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

        // Added: Set the user_id from the session.
        $article->user_id = $this->Auth->user('id');

        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);
}
```

Remember to remove the `user_id` control from **src/Templates/Articles/add.ctp** as well. 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();

    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.'));
    }

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

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

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 **src/Templates/Articles/edit.ctp** 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[56], 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.

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

[56] http://en.wikipedia.org/wiki/Continuous_integration

- /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[57].
- 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. E.g. "asdf asdf *Controller* asdf" or "asdf asdf Kontroller (*Controller*) asfd" as appropriate.
- Do not submit partial translations.

[57] http://en.wikipedia.org/wiki/Register_(linguistics)

- Do not edit a section with a pending change.
- Do not use HTML entities[58] 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[59]. 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.

Paragraphs

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

[58] http://en.wikipedia.org/wiki/List_of_XML_and_HTML_character_entity_references
[59] http://en.wikipedia.org/wiki/ReStructuredText

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

The resulting link would look like this: External Link to php.net[60]

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>``.

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[61] 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.

[60] http://php.net
[61] http://pypi.python.org/pypi/sphinxcontrib-phpdomain

.. **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*

.. **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 2.6.3: This awesome feature was added on version 2.6.3

Deprecated since version 2.6.3: This old feature was deprecated on version 2.6.3

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[62].

[62] https://github.com/cakephp/cakephp/issues

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[63] 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. The #cakephp IRC channel on Freenode[64] has many developers available to help answer your questions. Also have a look at Stack Overflow[65].

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 5.6.0 or greater
- PHPUnit 5.7.0 or greater

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

[63] https://github.com/cakephp/cakephp/search?q=it+is+broken&ref=cmdform&type=Issues
[64] https://webchat.freenode.net
[65] https://stackoverflow.com/questions/tagged/cakephp

```
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[66] book.

Get a clone of the CakePHP source code from GitHub:

- If you don't have a GitHub[67] account, create one.
- Fork the CakePHP repository[68] 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
```

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. This makes merging your changes in later much simpler, as Github does not let you edit the target 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. e.g. ticket-1234, 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 good clear and concise commit messages.

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

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 `3.2.10`, the branch accepting new features will be `3.next`.
- If your change is a breaks existing functionality, or API's then you'll have to choose then next major release. For example, if the current release is `3.2.2` then the next time existing behavior can be broken will be in `4.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[69] will become the owner of any contributed code. Contributors should follow the CakePHP Community Guidelines[70].

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

[69] http://cakefoundation.org/pages/about
[70] http://community.cakephp.org/guidelines

Coding Standards

CakePHP developers will use the PSR-2 coding style guide[71] 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[72] to check that your code follows required standards.

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.

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:

[71] http://www.php-fig.org/psr/psr-2/
[72] https://github.com/cakephp/cakephp-codesniffer

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

Use this instead:

```
$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. 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;
}

// 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 (.ctp 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;
    } else {
        $dnsInfo = BD::parseDNS($dns);
    }

    if (!($dnsInfo) || !($dnsInfo['phpType'])) {
        return $this->addError();
    }

    return true;
}
```

There are spaces on both side of the equals sign.

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

[73] http://www.php-fig.org/psr/psr-2/

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[74] tags:

- @author[75]
- @copyright[76]
- @deprecated[77] Using the `@version <vector> <description>` format, where `version` and `description` are mandatory.
- @example[78]
- @ignore[79]
- @internal[80]
- @link[81]
- @see[82]
- @since[83]
- @version[84]

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
```

[74] http://phpdoc.org
[75] http://phpdoc.org/docs/latest/references/phpdoc/tags/author.html
[76] http://phpdoc.org/docs/latest/references/phpdoc/tags/copyright.html
[77] http://phpdoc.org/docs/latest/references/phpdoc/tags/deprecated.html
[78] http://phpdoc.org/docs/latest/references/phpdoc/tags/example.html
[79] http://phpdoc.org/docs/latest/references/phpdoc/tags/ignore.html
[80] http://phpdoc.org/docs/latest/references/phpdoc/tags/internal.html
[81] http://phpdoc.org/docs/latest/references/phpdoc/tags/link.html
[82] http://phpdoc.org/docs/latest/references/phpdoc/tags/see.html
[83] http://phpdoc.org/docs/latest/references/phpdoc/tags/since.html
[84] http://phpdoc.org/docs/latest/references/phpdoc/tags/version.html

```
 */
function foo()
{
}
```

Comment blocks, with the exception of the first block in a file, should always be preceded by a newline.

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, e.g. 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[85] function.

PHP Tags

Always use long tags (`<?php ?>`) instead of short tags (`<? ?>`). The short echo should be used in template files (**.ctp**) 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()
{
}
```

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:

[85] http://php.net/require_once

```
$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**[86]) 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
```

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:

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

```
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);
```

Careful when using empty()/isset()

While `empty()` is an easy to use function, it can mask errors and cause unintended effects when `'0'` and `0` are given. When variables or properties are already defined, the usage of `empty()` is not recommended. When working with variables, it is better to rely on type-coercion to boolean instead of `empty()`:

```
function manipulate($var)
{
    // Not recommended, $var is already defined in the scope
    if (empty($var)) {
        // ...
    }

    // Use boolean type coercion
    if (!$var) {
        // ...
    }
    if ($var) {
        // ...
    }
}
```

When dealing with defined properties you should favour `null` checks over `empty()`/`isset()` checks:

```
class Thing
{
    private $property; // Defined

    public function readProperty()
    {
        // Not recommended as the property is defined in the class
        if (!isset($this->property)) {
            // ...
        }
        // Recommended
        if ($this->property === null) {

        }
    }
}
```

When working with arrays, it is better to merge in defaults over using `empty()` checks. By merging in defaults, you can ensure that required keys are defined:

```
function doWork(array $array)
{
    // Merge defaults to remove need for empty checks.
    $array += [
        'key' => null,
    ];
```

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

    // Recommended
    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[87], 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: CakePHP started following semantic versioning in 2.0.0. These rules do not apply to 1.x.

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.

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.

[87] http://semver.org/

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 [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

[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.

CHAPTER 6

Installation

CakePHP is simple and easy to install. The minimum requirements are a web server and a copy of CakePHP, that's it! While this chapter focuses primarily on setting up on Apache (because it's simple to install and setup), CakePHP will run on a variety of web servers such as nginx, LightHTTPD, or Microsoft IIS.

Requirements

- HTTP Server. For example: Apache. Having mod_rewrite is preferred, but by no means required.
- PHP 5.6.0 or greater (including PHP 7.1).
- mbstring PHP extension
- intl PHP extension

Note: In both XAMPP and WAMP, the mbstring extension is working by default.

In XAMPP, intl extension is included but you have to uncomment `extension=php_intl.dll` 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 (5.1.10 or greater)
- PostgreSQL
- Microsoft SQL Server (2008 or higher)
- SQLite 3

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 5.6.0 (CLI) or higher. Your webserver's PHP version must also be of 5.6.0 or higher, and should be the same version your command line interface (CLI) uses.

Installing Composer

CakePHP uses Composer[88], 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[89] 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[90]. Further instructions for Composer's Windows installer can be found within the README here[91].

Create a CakePHP Project

Now that you've downloaded and installed Composer, create a new CakePHP application into my_app_name folder. For this just run the following composer command:

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

Or if Composer is installed globally:

```
composer self-update && composer create-project --prefer-dist cakephp/app 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 **src/Template/Pages/home.ctp**.

Although composer is the recommended installation method, there are pre-installed downloads available on Github[92]. 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.

[88] http://getcomposer.org
[89] https://getcomposer.org/download/
[90] https://github.com/composer/windows-setup/releases/
[91] https://github.com/composer/windows-setup
[92] 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": "3.4.*"
}
```

Each time you run `php composer.phar update` you will receive bugfix releases for this minor version. You can instead change this to `~3.4` to also receive the latest stable releases of the `3.x` branch.

If you want to stay up to date with the latest unreleased changes in CakePHP, designate **dev-master** as the package version in your application's **composer.json**:

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

Be aware that this is not recommended, as your application can break when the next major version is released. Additionally, composer does not cache development branches, so it slows down consecutive composer installs/updates.

Installation using Oven

Another quick way to install CakePHP is Oven[93]. It is a simple PHP script which checks the necessary system requirements, installs the CakePHP application skeleton, and sets up the development environment.

After the installation completes, your CakePHP application is ready to go!

Note: IMPORTANT: This is not a deployment script. It is aimed to help developers install CakePHP for the first time and set up a development environment quickly. Production environments should consider several other factors, like file permissions, virtualhost configuration, etc.

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 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
```

[93] https://github.com/CakeDC/oven

```
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.

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/
    src/
    tests/
    tmp/
    vendor/
    webroot/ (this directory is set as DocumentRoot)
    .gitignore
    .htaccess
    .travis.yml
    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 http://wiki.apache.org/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
    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 Indexes FollowSymLinks MultiViews
    AllowOverride All
    Order Allow,Deny
    Allow from all
</Directory>
```

   On macOS, another solution is to use the tool virtualhostx[94] 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/

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

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:

```
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.

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[95] to install the URL Rewrite Module 2.0[96] or download it directly (32-bit[97] / 64-bit[98]).
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)"
```

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

```
                    stopProcessing="true">
                      <match url="^(img|css|files|js|favicon.ico)(.*)$" />
                      <action type="Rewrite" url="webroot/{R:1}{R:2}"
                        appendQueryString="false" />
                    </rule>
                    <rule name="Rewrite requested file/folder to index.php"
                      stopProcessing="true">
                      <match url="^(.*)$" ignoreCase="false" />
                      <action type="Rewrite" url="index.php"
                        appendQueryString="true" />
                    </rule>
                </rules>
            </rewrite>
        </system.webServer>
</configuration>
```

Once the web.config file is created with the correct IIS-friendly rewrite rules, CakePHP's links, CSS, JavaScript, and rerouting should work correctly.

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 `config()` methods to make configuration simple and transparent.

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::config('default', new PhpConfig());
Configure::load('app', 'default', false);
Configure::load('other_config', 'default');
```

You can also use additional configuration files to provide environment specific overrides. Each file loaded after **app.php** can redefine previously declared values allowing you to customize configuration for development or staging environments.

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.

> **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 /. E.g., */basedir* is a valid App.base. Otherwise, the AuthComponent will not work properly.

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 shells) 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 (e.g. 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.

Security.salt A random string used in hashing. This value is also used as the HMAC salt when doing symetric 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.

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.

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' => [
                APP . 'Template' . DS,
                APP . 'Template2' . DS
            ],
            'locales' => [
                APP . 'Locale' . DS
            ]
        ]
    ]
];
```

Paths should end with a directory separator, or they will not work properly.

Inflection Configuration

See the *Inflection Configuration* docs for more information.

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[99]. 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[100] to allow easy local development using environment variables. You will see a `config/.env.default` 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.default` 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.

Changed in version 3.5.0: dotenv library support was added to the application skeleton.

[99] http://12factor.net/
[100] https://github.com/josegonzalez/php-dotenv

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. This is especially handy for JSON interactions where debugging information can cause parsing problems.

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.

Changed in version 3.5.0: The `$default` parameter was added in 3.5.0

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'];
```

New in version 3.1.7: `Configure::readOrFail()` was added in 3.1.7

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.

Reading and writing configuration files

static `Cake\Core\Configure::config`(*$name*, *$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[101] 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()`*:

[101] http://php.net/parse_ini_file

```
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()`:

```
// 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.

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 confiugration engines[102]. The built in configuration engines are:

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

By default your application will use `PhpConfig`.

Bootstrapping CakePHP

If you have any additional configuration needs, you should add them to your application's **config/bootstrap.php** file. This file is included before each request, and CLI command.

This file is ideal for a number of common bootstrapping tasks:

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

- Defining convenience functions.
- Declaring constants.
- Defining cache configuration.
- Defining logging configuration.
- Loading custom inflections.
- Loading configuration files.

It might be tempting to place formatting functions there in order to use them in your controllers. As you'll see in the *Controllers* and *Views* sections there are better ways you add custom logic to your application.

Application::bootstrap()

In addition to the **config/bootstrap.php** file which should be used to configure low-level concerns of your application, you can also use the `Application::bootstrap()` hook method to load/initialize plugins, and attach global event listeners:

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

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

class Application extends BaseApplication
{
    public function bootstrap()
    {
        // Call the parent to `require_once` config/bootstrap.php
        parent::bootstrap();

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

Loading plugins/events in `Application::bootstrap()` makes *Controller Integration Testing* easier as events and routes will be re-processed on each test method.

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\Network\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);
        }
    });
}
```

CHAPTER 8

Routing

class `Cake\Routing\`**`Router`**

Routing provides you tools that map URLs to controller actions. By defining routes, you can separate how your application is implemented from how its URL's 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 **routes.php** file:

```
use Cake\Routing\Router;

// Using the scoped route builder.
Router::scope('/', function ($routes) {
    $routes->connect('/', ['controller' => 'Articles', 'action' => 'index']);
});

// Using the static method.
Router::connect('/', ['controller' => 'Articles', 'action' => 'index']);
```

`Router` provides two interfaces for connecting routes. The static method is a backwards compatible interface, while the scoped builders offer more terse syntax when building multiple routes, and better performance.

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:

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

// Prior to 3.5 use the 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(
    '/login',
    ['controller' => 'Users', 'action' => 'login'],
    ['_name' => 'login']
);

use Cake\Routing\Router;

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

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:

```
Router::scope('/blog', ['plugin' => 'Blog'], function ($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

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

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\Route\DashedRoute;

Router::scope('/', function ($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',
    ['default' => 'defaultValue'],
    ['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 is an associative array. The keys of the array should be named after the route elements the URL template represents. The values in the array are the default values for those keys. Let's look at some basic examples before we start using the third parameter of `connect()`:

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

This route is found in the routes.php file distributed with CakePHP. It matches any URL starting with `/pages/` and hands it to the `display()` action of the `PagesController`. A request to `/pages/products` would be mapped to `PagesController->display('products')`.

In addition to the greedy star `/*` there is also the `/**` trailing star syntax. 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`.

You can use the second parameter of `connect()` to provide any routing parameters that are composed of the default values of the route:

```
$routes->connect(
    '/government',
    ['controller' => 'Pages', 'action' => 'display', 5]
);
```

This example shows how you can use the second parameter of `connect()` to define default parameters. If you built a site 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 create URL segments that don't match your controller or model names. Let's say that instead of accessing our regular URL at `/users/some_action/5`, we'd like to be able to access it by `/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.

New in version 3.5.0: The HTTP verb helper methods were added in 3.5.0

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]+']);

// Prior to 3.5 use the options array
$routes->connect(
    '/:controller/:id',
    ['action' => 'view'],
    ['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.
$routes->scope('/', function ($routes) {
    $routes->setRouteClass(DashedRoute::class);
    $routes->connect('/:controller/:id', ['action' => 'view'])
        ->setPatterns(['id' => '[0-9]+']);

    // Prior to 3.5 use options array
    $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.

If you need lowercased and underscored URLs while migrating from a CakePHP 2.x application, you can instead use the `InflectedRoute` class.

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:

```
// Prior to 3.5 use the options array instead of setPatterns()
$routes->connect(
    '/:userShortcut',
    ['controller' => 'Teachers', 'action' => 'profile', 1],
)->setPatterns(['userShortcut' => '(?i:principal)']);
```

One more example, and you'll be a routing pro:

```
// Prior to 3.5 use the options array instead of setPatterns()
$routes->connect(
    '/:controller/:year/:month/:day',
    ['action' => 'index']
)->setPatterns([
    'year' => '[12][0-9]{3}',
    'month' => '0[1-9]|1[012]',
    'day' => '0[1-9]|[12][0-9]|3[01]'
]);
```

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 (grouping) 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()`.

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 *FULL_BASE_URL* constant will be prepended to generated URLs.
- `#` Allows you to set URL hash fragments.
- `_ssl` Set to `true` to convert the generated URL to https or `false` to force http.
- `_method` Define the HTTP verb/method to use. Useful when working with *Creating RESTful Routes*.
- `_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
->setExtenions(['json'])

// Set lang to be a persistent parameter
->setPersist(['lang']);
```

New in version 3.5.0: Fluent builder methods were added in 3.5.0

Passing Parameters to Action

When connecting routes using *Route Elements* you may want to have routed elements be passed arguments instead. The `pass` option whitelists 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
Router::scope('/', function ($routes) {
    $routes->connect(
        '/blog/:id-:slug', // E.g. /blog/3-CakePHP_Rocks
        ['controller' => 'Blogs', 'action' => 'view']
    )
    // Define the route elements in the route template
    // to pass as function arguments. Order matters since this
    // will simply map ":id" to $articleId in your action
    ->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.ctp
// 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 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 specifiy 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 (3.5.0+)
$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:

```
Router::scope('/api', ['_namePrefix' => 'api:'], function ($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()
Router::plugin('Contacts', ['_namePrefix' => 'contacts:'], function ($routes) {
    // Connect routes.
});

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

You can also use the `_namePrefix` option inside nested scopes and it works as you'd expect:

```
Router::plugin('Contacts', ['_namePrefix' => 'contacts:'], function ($routes) {
    $routes->scope('/api', ['_namePrefix' => 'api:'], function ($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.

New in version 3.1: The `_namePrefix` option was added in 3.1

Prefix Routing

static `Cake\Routing\Router::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;

Router::prefix('admin', function ($routes) {
    // All routes here will be prefixed with `/admin`
    // And have the prefix => admin route element added.
    $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 **src/Template/Admin/Users/edit.ctp**

You can map the URL /admin to your `index()` action of pages controller using following route:

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

```
Router::prefix('admin', ['param' => 'value'], function ($routes) {
    // Routes connected here are prefixed with '/admin' and
    // have the 'param' routing key set.
    $routes->connect('/:controller');
});
```

You can define prefixes inside plugin scopes as well:

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

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

The above would create a route template like `/manager/admin/:controller`. 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. Here's how to build this link using the HTML helper:

```
// Go into a prefixed route.
echo $this->Html->link(
    'Manage articles',
    ['prefix' => 'manager', 'controller' => 'Articles', 'action' => 'add']
);

// Leave a prefix
echo $this->Html->link(
    'View Post',
    ['prefix' => false, 'controller' => 'Articles', 'action' => 'view', 5]
);
```

Note: You should connect prefix routes *before* you connect fallback routes.

Plugin Routing

static `Cake\Routing\Router::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:

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

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

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

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

Matching Specific HTTP Methods

Routes can match specific HTTP methods using the HTTP verb helper methods:

```
Router::scope('/', function($routes) {
    // This route only matches on POST requests.
    $routes->post(
        '/reviews/start',
        ['controller' => 'Reviews', 'action' => 'start']
    );

    // Match multiple verbs
    // Prior to 3.5 use $options['_method'] to set method
    $routes->connect(
        '/reviews/start',
        [
            'controller' => 'Reviews',
            'action' => 'start',
        ]
    )->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:

```
Router::scope('/', function($routes) {
    // This route only matches on http://images.example.com
    // Prior to 3.5 use the _host option
    $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('images.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',
]);
```

New in version 3.4.0: The `_host` option was added in 3.4.0

Routing File Extensions

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

To handle different file extensions with your routes, you can define extensions on a global, as well as on a scoped level. Defining global extensions can be achieved via the routers static *`Router::extensions()`* method:

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

This will affect **all** routes that are being connected **afterwards**, no matter their scope.

In order to restrict extensions to specific scopes, you can define them using the `Cake\Routing\RouteBuilder::setExtensions()` method:

```
Router::scope('/', function ($routes) {
    // Prior to 3.5.0 use `extensions()`
    $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. Similar to the global *`Router::extensions()`* method, any routes connected prior to the call will not inherit the extensions.

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, 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:

```
Router::scope('/page', function ($routes) {
    // Prior to 3.5.0 use `extensions()`
    $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']
);
```

File extensions are used by *Request Handling* to do automatic view switching based on content types.

Connecting 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.

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;
```

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

Once registered, scoped middleware can be applied to specific scopes:

```
$routes->scope('/cms', function ($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:

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

```
$routes->scope('/blog', function ($routes) {
    $routes->applyMiddleware('auth');
    // Connect the authenticated actions for the blog here.
});
$routes->scope('/blog', function ($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');
```

New in version 3.5.0: Scoped middleware & middleware groups were added in 3.5.0

Creating RESTful Routes

Router makes it easy to 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...

Router::scope('/', function ($routes) {
    // Prior to 3.5.0 use `extensions()`
    $routes->setExtensions(['json']);
    $routes->resources('Recipes');
});
```

The first line sets up a number of default routes for easy REST access where method specifies the desired result format (e.g. xml, json, 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)

CakePHP's Router class uses a number of different indicators to detect the HTTP method being used. Here they are in order of preference:

1. The _method POST variable
2. The X_HTTP_METHOD_OVERRIDE
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:

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

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

New in version 3.3: The `prefix` option was added to `resources()` in 3.3.

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`.

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' => 'DELETE',
            '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.

Custom Route Classes for Resource Routes

You can provide `connectOptions` key in the `$options` array for `resources()` to provide custom setting used by `connect()`:

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

URL Inflection for Resource Routes

By default, multi-worded controllers' URL fragments are the underscored form of the controller's name. E.g., `BlogPostsController`'s URL fragment would be **/blog_posts**.

You can specify an alternative inflection type using the `inflect` option:

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

The above will generate URLs styled like: **/blog-posts**.

Note: As of CakePHP 3.1 the official app skeleton uses `DashedRoute` as its default route class. Using the `'inflect' => 'dasherize'` option when connecting resource routes is recommended for URL consistency.

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:

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

New in version 3.5.0: The `path` option was added in 3.5.0

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\Router::url` (*$url = null*, *$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. By using *routing arrays* to define your URLs, you can later configure routes and the generated URLs will automatically update.

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:

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

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.

When using array URLs, you can define both query string parameters and document fragments using special keys:

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

// Will generate a URL like.
/articles/index?page=1#top
```

Router will also convert any unknown parameters in a routing array to querystring parameters. The `?` is offered for backwards compatibility with older versions of CakePHP.

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 `FULL_BASE_URL` constant will be prepended to generated URLs.
- `_ssl` Set to `true` to convert the generated URL to https or `false` to force http.
- `_name` Name of route. If you have setup named routes, you can use this key to specify it.

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:

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

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:

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

This would redirect `/articles/*` to `http://google.com` with a HTTP status of 302.

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 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 `false` 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']
);

// Or by setting the routeClass in your scope.
$routes->scope('/', function ($routes) {
    //Prior to 3.5.0 use `routeClass()`
```

```
    $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\Router::`**`defaultRouteClass`**(*$routeClass = null*)

If you want to use an alternate route class for all your routes besides the default `Route`, you can do so by calling `Router::defaultRouteClass()` before setting up any routes and avoid having to specify the `routeClass` option for each route. For example using:

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

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

will cause all routes connected after this to use the `InflectedRoute` route class. Calling the method without an argument will return current default route class.

Fallbacks Method

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

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:

- `$params` The URL params being processed.
- `$request` The current request.

The URL filter function should *always* return the params even if unmodified.

URL filters allow you to implement features like persistent parameters:

```
Router::addUrlFilter(function ($params, $request) {
    if ($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 ($params, $request) {
    if (empty($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']);
```

Handling Named Parameters in URLs

Although named parameters were removed in CakePHP 3.0, applications may have published URLs containing them. You can continue to accept URLs containing named parameters.

In your controller's `beforeFilter()` method you can call `parseNamedParams()` to extract any named parameters from the passed arguments:

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

This will populate `$this->request->getParam('named')` with any named parameters found in the passed arguments. Any passed argument that was interpreted as a named parameter, will be removed from the list of passed arguments.

Dispatcher Filters

Deprecated since version 3.3.0: As of 3.3.0 Dispatcher Filters are deprecated. You should use /controllers/middleware instead now.

There are several reasons to want a piece of code to be run before any controller code is executed or right before the response is sent to the client, such as response caching, header tuning, special authentication or just to provide access to a mission-critical API response in lesser time than a complete request dispatching cycle would take.

CakePHP provides a clean interface for attaching filters to the dispatch cycle. It is similar to a middleware layer, but re-uses the existing event subsystem used in other parts of CakePHP. Since they do not work exactly like traditional middleware, we refer to them as *Dispatcher Filters*.

Built-in Filters

CakePHP comes with several dispatcher filters built-in. They handle common features that all applications are likely to need. The built-in filters are:

- `AssetFilter` checks whether the request is referring to a theme or plugin asset file, such as a CSS, JavaScript or image file stored in either a plugin's webroot folder or the corresponding one for a Theme. It will serve the file accordingly if found, stopping the rest of the dispatching cycle:

  ```
  // Use options to set cacheTime for your static assets
  // If not set, this defaults to +1 hour
  DispatcherFactory::add('Asset', ['cacheTime' => '+24 hours']);
  ```

- `RoutingFilter` applies application routing rules to the request URL. Populates `$request->getParam()` with the results of routing.
- `ControllerFactory` uses `$request->getParam()` to locate the controller that will handle the current request.
- `LocaleSelector` enables automatic language switching from the `Accept-Language` header sent by the browser.

Using Filters

Filters are usually enabled in your application's **bootstrap.php** file, but you could load them any time before the request is dispatched. Adding and removing filters is done through `Cake\Routing\DispatcherFactory`. By default, the CakePHP application template comes with a couple filter classes already enabled for all requests; let's take a look at how they are added:

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

```
// Plugin syntax is also possible
DispatcherFactory::add('PluginName.DispatcherName');

// Use options to set priority
DispatcherFactory::add('Asset', ['priority' => 1]);
```

Dispatcher filters with higher `priority` (lower numbers) - will be executed first. Priority defaults to `10`.

While using the string name is convenient, you can also pass instances into `add()`:

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

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

Configuring Filter Order

When adding filters, you can control the order they are invoked in using event handler priorities. While filters can define a default priority using the `$_priority` property, you can set a specific priority when attaching the filter:

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

The higher the priority the later this filter will be invoked.

Conditionally Applying Filters

If you don't want to run a filter on every request, you can use conditions to only apply it some of the time. You can apply conditions using the `for` and `when` options. The `for` option lets you match on URL substrings, while the `when` option allows you to run a callable:

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

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

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

Building a Filter

To create a filter, define a class in **src/Routing/Filter**. In this example, we'll be making a filter that adds a tracking cookie for the first landing page. First, create the file. Its contents should look like:

```
namespace App\Routing\Filter;

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

```
class TrackingCookieFilter extends DispatcherFilter
{

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

Save this file into **src/Routing/Filter/TrackingCookieFilter.php**. As you can see, like other classes in CakePHP, dispatcher filters have a few conventions:

- Class names end in `Filter`.
- Classes are in the `Routing\Filter` namespace. For example, `App\Routing\Filter`.
- Generally filters extend `Cake\Routing\DispatcherFilter`.

`DispatcherFilter` exposes two methods that can be overridden in subclasses, they are `beforeDispatch()` and `afterDispatch()`. These methods are executed before or after any controller is executed respectively. Both methods receive a `Cake\Event\Event` object containing the `ServerRequest` and `Response` objects (*`Cake\Http\ServerRequest`* and *`Cake\Http\Response`* instances) inside the `$data` property.

While our filter was pretty simple, there are a few other interesting things we can do in filter methods. By returning an `Response` object, you can short-circuit the dispatch process and prevent the controller from being called. When returning a response, you should also remember to call `$event->stopPropagation()` so other filters are not called.

Note: When a beforeDispatch method returns a response, the controller, and afterDispatch event will not be invoked.

Let's now create another filter for altering response headers in any public page, in our case it would be anything served from the `PagesController`:

```
namespace App\Routing\Filter;

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

class HttpCacheFilter extends DispatcherFilter
{

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

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

```
    }
}


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

This filter will send a expiration header to 1 day in the future for all responses produced by the pages controller. You could of course do the same in the controller, this is just an example of what could be done with filters. For instance, instead of altering the response, you could cache it using `Cake\Cache\Cache` and serve the response from the `beforeDispatch()` callback.

While powerful, dispatcher filters have the potential to make your application more difficult to maintain. Filters are an extremely powerful tool when used wisely and adding response handlers for each URL in your app is not a good use for them. Keep in mind that not everything needs to be a filter; *Controllers* and *Components* are usually a more accurate choice for adding any request handling code to your app.

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. 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.

As of 3.4.0, CakePHP's request object implements the PSR-7 ServerRequestInterface[106] making it easier to use libraries from outside of CakePHP.

Request Parameters

The request exposes the routing parameters through the `getParam()` method:

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

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

All *Route Elements* are accessed through this interface.

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

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*)

Query string parameters can be read using the `getQuery()` method:

```
// URL is /posts/index?page=1&sort=title
$page = $this->request->getQuery('page');

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

New in version 3.4.0: `getQueryParams()` and `getQuery()` were added in 3.4.0

Request Body Data

`Cake\Http\ServerRequest::getData`(*$name*, *$default = null*)

All POST data can be accessed using *`Cake\Http\ServerRequest::getData()`*. Any form data that contains a `data` prefix will have that data prefix removed. For example:

```
// An input with a name attribute equal to 'MyModel[title]' is accessible at
$title = $this->request->getData('MyModel.title');
```

Any keys that do not exist will return `null`:

```
$foo = $this->request->getData('Value.that.does.not.exist');
// $foo == null
```

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::env`(*$key*, *$value* = *null*)

`ServerRequest::env()` is a wrapper for `env()` global function and acts as a getter/setter for enviromnent variables without having to modify globals `$_SERVER` and `$_ENV`:

```
// Get the host
$host = $this->request->env('HTTP_HOST');

// Set a value, generally helpful in testing.
$this->request->env('REQUEST_METHOD', 'POST');
```

To access all the environment variables in a request use `getServerParams()`:

```
$env = $this->request->getServerParams();
```

New in version 3.4.0: `getServerParams()` was added in 3.4.0

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 *`Http\ServerRequest::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, *`Http\ServerRequest::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->here();

// Holds /subdir
$base = $request->getAttribute('base');

// Holds /subdir/
$base = $request->getAttribute('base');

// Prior to 3.4.0
$webroot = $request->webroot;
$base = $request->base;
```

Checking Request Conditions

`Cake\Http\ServerRequest::is`(*$type*, *$args...*)

The request object provides an easy 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 four different types of detectors that you can create:

- Environment value comparison - Compares a value fetched from *env()* for equality with the provided value.
- 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']
);

// Add an option detector
$this->request->addDetector('internalIp', [
```

```
    'env' => 'CLIENT_IP',
    'options' => ['192.168.0.101', '192.168.0.100']
]);

// 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. As of 3.3.0
$this->request->addDetector(
    'controller',
    function ($request, $name) {
        return $request->getParam('controller') === $name;
    }
);
```

`Request` also includes methods like *Cake\Http\ServerRequest::domain()*, *Cake\Http\ServerRequest::subdomains()* and *Cake\Http\ServerRequest::host()* to help applications with subdomains, have a slightly easier life.

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('requested')` Check to see whether the request has a query param 'requested' with value 1.
- `is('json')` Check to see whether the request has 'json' extension and accept 'application/json' mimetype.
- `is('xml')` Check to see whether the request has 'xml' extension and accept 'application/xml' or 'text/xml' mimetype.

New in version 3.3.0: Detectors can take additional parameters as of 3.3.0.

Session Data

To access the session for a given request use the `session()` method:

```
$userName = $this->request->session()->read('Auth.User.name');
```

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();

// Prior to 3.4.0
echo $request->method();
```

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');

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

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 = false*)

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();
```

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');
```

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 (starting with 3.5.0)
$cookies = $this->request->getCookieCollection()
```

See the *Cake\Http\Cookie\CookieCollection* documentation for how to work with cookie collection.

New in version 3.5.0: `ServerRequest::getCookieCollection()` was added in 3.5.0

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. Like *Cake\Http\ServerRequest*, *Cake\Http\Response* consolidates a number of methods previously found on `Controller`, *RequestHandlerComponent* and `Dispatcher`. The old methods are deprecated in favour of using *Cake\Http\Response*.

`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 `type()` as well:

```
// Add a vCard type
$this->response->type(['vcf' => 'text/v-card']);

// Set the response Content-Type to vcard.
$this->response = $this->response->withType('vcf');

// Prior to 3.4.0
$this->response->type('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`(*$path*, *$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;
}

// Prior to 3.4.0
$file = $this->Attachments->getFile($id);
$this->response->file($file['path']);
// Return the response to prevent controller from trying to render
// a view.
return $this->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\Reponse::$_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']
);

// Prior to 3.4.0
$this->response->file(
    $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;
    $response->body($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;
}
```

Callbacks can also return the body as a string:

```
$path = '/some/file.png';
$this->response->body(function () use ($path) {
    return file_get_contents($path);
});
```

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');

// Prior to 3.4.0 - Set a header
$this->response->header('Location', 'http://example.com');
```

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']));
```

New in version 3.4.3: `withStringBody()` was added in 3.4.3

`Cake\Http\Response::withBody(`*$body*`)`

To set the response body, use the `withBody()` method, which is provided by the `Zend\Diactoros\MessageTrait`:

```
$response = $response->withBody($stream);

// Prior to 3.4.0 - Set the body
$this->response->body('My Body');
```

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 `Zend\Diactoros\Stream` streams:

```
// To stream from a file
use Zend\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);

// Prior to 3.4.0 you can use the following to create streaming responses.
$file = fopen('/some/file.png', 'r');
$this->response->body(function () use ($file) {
    rewind($file);
    fpassthru($file);
    fclose($file);
});
```

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');

// Prior to 3.4.0
$this->response->charset('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 *Request Handling* in your controller:

```
public function index()
{
    $articles = $this->Articles->find('all');
    $response = $this->response->withEtag($this->Articles->generateHash($articles));
    if ($response->checkNotModified($this->request)) {
        return $response;
    }
    $this->response = $response;
    // ...
}
```

Note: 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 *Request Handling* in your controller:

```
public function view()
{
    $article = $this->Articles->find()->first();
    $response = $this->response->withModified($article->modified);
    if ($response->checkNotModified($this->request)) {
        return $response;
    }
    $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:

```
// Add a cookie as an array using the immutable API (3.4.0+)
$this->response = $this->response->withCookie('remember_me', [
    'value' => 'yes',
    'path' => '/',
    'httpOnly' => true,
    'secure' => false,
    'expire' => strtotime('+1 year')
]);

// Before 3.4.0
$this->response->cookie('remember', [
    'value' => 'yes',
    'path' => '/',
    'httpOnly' => true,
    'secure' => false,
    'expire' => strtotime('+1 year')
]);
```

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:

```
// As of 3.5.0
$this->response = $this->response->withExpiredCookie('remember_me');
```

Setting Cross Origin Request Headers (CORS)

As of 3.2 you can use the `cors()` method to define HTTP Access Control[107] related headers with a fluent interface:

```
$this->response->cors($this->request)
    ->allowOrigin(['*.cakephp.org'])
    ->allowMethods(['GET', 'POST'])
    ->allowHeaders(['X-CSRF-Token'])
    ->allowCredentials()
    ->exposeHeaders(['Link'])
    ->maxAge(300)
    ->build();
```

[107] https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS

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.

New in version 3.2: The `CorsBuilder` was added in 3.2

Common Mistakes with Immutable Responses

As of CakePHP 3.4.0, 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!');
```

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.

You should use the `withCookie()` method to add cookies to `Response` objects as it is simpler to use:

```
$response = $this->response->withCookie($cookie);
```

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
$cookie = $cookies->get('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();
```

New in version 3.5.0: `CookieCollection` and `Cookie` were added in 3.5.0.

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()
    {
        // Always enable the CSRF component.
        $this->loadComponent('Csrf');
    }

}
```

In addition to the `initialize()` method, the older `$components` property will also allow you to declare which components should be loaded. While normal object-oriented inheritance rules apply, the components and helpers used by a controller are treated specially. In these cases, `AppController` property values are merged with child controller class arrays. The values in the child class will always override those in `AppController`.

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 **src/Template/Recipes/view.ctp**, **src/Template/Recipes/share.ctp**, and **src/Template/Recipes/search.ctp**. 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);
```

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()
    ->helpers(['MyCustom'])
    ->theme('Modern')
    ->className('Modern.Admin');
```

The above shows how you can load custom helpers, set the theme and use a custom view class.

New in version 3.1: ViewBuilder was added in 3.1

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 **src/Template/Recipes/search.ctp** will be rendered:

```
namespace App\Controller;

class RecipesController extends AppController
{
// ...
    public function search()
    {
        // Render the view in src/Template/Recipes/search.ctp
        $this->render();
    }
// ...
}
```

Although CakePHP will automatically call it after every action's logic (unless you've set `$this->autoRender` to `false`), 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 **src/Template** folder. This allows direct rendering of elements, very useful in AJAX calls:

```
// Render the element in src/Template/Element/ajaxreturn.ctp
$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 **src/Template/Posts/custom_file.ctp** instead of **src/Template/Posts/my_action.ctp**.

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 my_action()
    {
        $this->render('Users.UserDetails/custom_file');
    }
}
```

This would render **plugins/Users/src/Template/UserDetails/custom_file.ctp**

Redirecting to Other Pages

`Cake\Controller\Controller::`**`redirect`**(*string|array $url*, *integer $status*)

The flow control method you'll use most often is `Controller::redirect()`. This method takes its first parameter in the form of a CakePHP-relative URL. When a user has successfully placed an order, you might wish to redirect him to a receipt screen.

```
public function place_order()
{
    // Logic for finalizing order goes here
    if ($success) {
        return $this->redirect(
            ['controller' => 'Orders', 'action' => 'thanks']
        );
    }
    return $this->redirect(
        ['controller' => 'Orders', 'action' => 'confirm']
    );
}
```

The method will return the response instance with appropriate headers set. You should return the response instance from your action to prevent view rendering and let the dispatcher handle actual redirection.

You can also use a relative or absolute URL as the $url argument:

```
return $this->redirect('/orders/thanks');
return $this->redirect('http://www.example.com');
```

You can also pass data to the action:

```
return $this->redirect(['action' => 'edit', $id]);
```

The second parameter of `Controller::redirect()` allows you to define an HTTP status code to accompany the redirect. You may want to use 301 (moved permanently) or 303 (see other), depending on the nature of the redirect.

If you need to redirect to the referer page you can use:

```
return $this->redirect($this->referer());
```

An example using query strings and hash would look like:

```
return $this->redirect([
    'controller' => 'Orders',
    'action' => 'confirm',
    '?' => [
        'product' => 'pizza',
        'quantity' => 5
    ],
    '#' => 'top'
]);
```

The generated URL would be:

```
http://www.example.com/orders/confirm?product=pizza&quantity=5#top
```

Redirecting to Another Action on the Same Controller

`Cake\Controller\Controller::`**`setAction`**(*$action*, *$args...*)

If you need to forward the current action to a different action on the *same* controller, you can use `Controller::setAction()` to update the request object, modify the view template that will be rendered and forward execution to the named action:

```
// From a delete action, you can render the updated
// list page.
$this->setAction('index');
```

Loading Additional Models

`Cake\Controller\Controller::`**`loadModel`**(*string $modelClass*, *string $type*)

The `loadModel()` function comes handy when you need to use a model table/collection that is not the controller's default one:

```
// In a controller method.
$this->loadModel('Articles');
$recentArticles = $this->Articles->find('all', [
    'limit' => 5,
    'order' => 'Articles.created DESC'
]);
```

If you are using a table provider other than the built-in ORM you can link that table system into CakePHP's controllers by connecting its factory method:

```
// In a controller method.
$this->modelFactory(
    'ElasticIndex',
    ['ElasticIndexes', 'factory']
);
```

After registering a table factory, you can use `loadModel` to load instances:

```
// In a controller method.
$this->loadModel('Locations', 'ElasticIndex');
```

Note: The built-in ORM's TableRegistry is connected by default as the 'Table' provider.

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 an easy way to customize how `paginate()` behaves:

```
class ArticlesController extends AppController
{
    public $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()
{
    parent::initialize();
    $this->loadComponent('Csrf');
```

```
    $this->loadComponent('Comments', Configure::read('Comments'));
}
```

property Cake\Controller\Controller::$**components**

The $components property on your controllers allows you to configure components. Configured components and their dependencies will be created by CakePHP for you. Read the *Configuring Components* section for more information. As mentioned earlier the $components property will be merged with the property defined in each of you controller's parent classes.

Configuring Helpers to Load

property Cake\Controller\Controller::$**helpers**

Let's look at how to tell a CakePHP Controller that you plan to use additional MVC classes:

```
class RecipesController extends AppController
{
    public $helpers = ['Form'];
}
```

Each of these variables are merged with their inherited values, therefore it is not necessary (for example) to redeclare the FormHelper, or anything that is declared in your AppController.

Deprecated since version 3.0: Loading Helpers from the controller is provided for backwards compatibility reasons. You should see *Configuring Helpers* for how to load helpers.

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**(*Event $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`(*Event $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 *`Controller\Controller::render()`* manually before the end of a given action.

`Cake\Controller\Controller::afterFilter`(*Event $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:

```
public function beforeFilter(Event $event)
{
    parent::beforeFilter($event);
}
```

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 **src/Template/Pages/home.ctp**. If you make the view file **src/Template/Pages/about_us.ctp** 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:

Authentication

class `AuthComponent`(*ComponentCollection $collection*, *array $config = []*)

Identifying, authenticating, and authorizing users is a common part of almost every web application. In CakePHP AuthComponent provides a pluggable way to do these tasks. AuthComponent allows you to combine authentication objects and authorization objects to create flexible ways of identifying and checking user authorization.

Suggested Reading Before Continuing

Configuring authentication requires several steps including defining a users table, creating a model, controller & views, etc.

This is all covered step by step in the *Blog Tutorial*.

If you are looking for existing authentication and/or authorization solutions for CakePHP, have a look at the Authentication and Authorization[108] section of the Awesome CakePHP list.

Authentication

Authentication is the process of identifying users by provided credentials and ensuring that users are who they say they are. Generally, this is done through a username and password, that are checked against a known list of users. In CakePHP, there are several built-in ways of authenticating users stored in your application.

- `FormAuthenticate` allows you to authenticate users based on form POST data. Usually, this is a login form that users enter information into.
- `BasicAuthenticate` allows you to authenticate users using Basic HTTP authentication.
- `DigestAuthenticate` allows you to authenticate users using Digest HTTP authentication.

By default `AuthComponent` uses `FormAuthenticate`.

Choosing an Authentication Type

Generally, you'll want to offer form based authentication. It is the easiest for users using a web-browser to use. If you are building an API or webservice, you may want to consider basic authentication or digest authentication. The key differences between digest and basic authentication are mostly related to how passwords are handled. In basic authentication, the username and password are transmitted as plain-text to the server. This makes basic authentication un-suitable for applications without SSL, as you would end up exposing sensitive passwords. Digest authentication uses a digest hash of the username, password, and a few other details. This makes digest authentication more appropriate for applications without SSL encryption.

You can also use authentication systems like OpenID as well; however, OpenID is not part of CakePHP core.

Configuring Authentication Handlers

You configure authentication handlers using the `authenticate` config. You can configure one or many handlers for authentication. Using multiple handlers allows you to support different ways of logging users in. When logging users in, authentication handlers are checked in the order they are declared. Once one handler is able to identify the user, no other handlers will be checked. Conversely, you can halt all authentication by throwing an exception. You will need to catch any thrown exceptions and handle them as needed.

You can configure authentication handlers in your controller's `beforeFilter()` or `initialize()` methods. You can pass configuration information into each authentication object using an array:

```
// Simple setup
$this->Auth->config('authenticate', ['Form']);

// Pass settings in
$this->Auth->config('authenticate', [
    'Basic' => ['userModel' => 'Members'],
```

[108] https://github.com/FriendsOfCake/awesome-cakephp/blob/master/README.md#authentication-and-authorization

```
    'Form' => ['userModel' => 'Members']
]);
```

In the second example, you'll notice that we had to declare the `userModel` key twice. To help you keep your code DRY, you can use the `all` key. This special key allows you to set settings that are passed to every attached object. The all key is also exposed as `AuthComponent::ALL`:

```
// Pass settings in using 'all'
$this->Auth->config('authenticate', [
    AuthComponent::ALL => ['userModel' => 'Members'],
    'Basic',
    'Form'
]);
```

In the above example, both `Form` and `Basic` will get the settings defined for the 'all' key. Any settings passed to a specific authentication object will override the matching key in the 'all' key. The core authentication objects support the following configuration keys.

- `fields` The fields to use to identify a user by. You can use keys `username` and `password` to specify your username and password fields respectively.
- `userModel` The model name of the users table; defaults to Users.
- `finder` The finder method to use to fetch a user record. Defaults to 'all'.
- `passwordHasher` Password hasher class; Defaults to `Default`.
- The `scope` and `contain` options have been deprecated as of 3.1. Use a custom finder instead to modify the query to fetch a user record.
- The `userFields` option has been deprecated as of 3.1. Use `select()` in your custom finder.

To configure different fields for user in your `initialize()` method:

```
public function initialize()
{
    parent::initialize();
    $this->loadComponent('Auth', [
        'authenticate' => [
            'Form' => [
                'fields' => ['username' => 'email', 'password' => 'passwd']
            ]
        ]
    ]);
}
```

Do not put other `Auth` configuration keys, such as `authError`, `loginAction`, etc., within the `authenticate` or `Form` element. They should be at the same level as the authenticate key. The setup above with other Auth configuration should look like:

```
public function initialize()
{
    parent::initialize();
    $this->loadComponent('Auth', [
        'loginAction' => [
            'controller' => 'Users',
            'action' => 'login',
            'plugin' => 'Users'
        ],
        'authError' => 'Did you really think you are allowed to see that?',
```

```
        'authenticate' => [
            'Form' => [
                'fields' => ['username' => 'email']
            ]
        ],
        'storage' => 'Session'
    ]);
}
```

In addition to the common configuration, Basic authentication supports the following keys:

- `realm` The realm being authenticated. Defaults to `env('SERVER_NAME')`.

In addition to the common configuration Digest authentication supports the following keys:

- `realm` The realm authentication is for. Defaults to the servername.
- `nonce` A nonce used for authentication. Defaults to `uniqid()`.
- `qop` Defaults to auth; no other values are supported at this time.
- `opaque` A string that must be returned unchanged by clients. Defaults to `md5($config['realm'])`.

Note: To find the user record, the database is queried only using the username. The password check is done in PHP. This is necessary because hashing algorithms like bcrypt (which is used by default) generate a new hash each time, even for the same string and you can't just do simple string comparison in SQL to check if the password matches.

Customizing Find Query

You can customize the query used to fetch the user record using the `finder` option in authenticate class config:

```
public function initialize()
{
    parent::initialize();
    $this->loadComponent('Auth', [
        'authenticate' => [
            'Form' => [
                'finder' => 'auth'
            ]
        ],
    ]);
}
```

This will require your `UsersTable` to have finder method `findAuth()`. In the example shown below the query is modified to fetch only required fields and add a condition. You must ensure that you select the fields you need to authenticate a user, such as `username` and `password`:

```
public function findAuth(\Cake\ORM\Query $query, array $options)
{
    $query
        ->select(['id', 'username', 'password'])
        ->where(['Users.active' => 1]);

    return $query;
}
```

Note: `finder` option is available since 3.1. Prior to that you can use `scope` and `contain` options to modify a query.

Identifying Users and Logging Them In

`AuthComponent::identify()`

You need to manually call `$this->Auth->identify()` to identify the user using credentials provided in request. Then use `$this->Auth->setUser()` to log the user in, i.e., save user info to session.

When authenticating users, attached authentication objects are checked in the order they are attached. Once one of the objects can identify the user, no other objects are checked. A sample login function for working with a login form could look like:

```
public function login()
{
    if ($this->request->is('post')) {
        $user = $this->Auth->identify();
        if ($user) {
            $this->Auth->setUser($user);
            return $this->redirect($this->Auth->redirectUrl());
        } else {
            $this->Flash->error(__('Username or password is incorrect'));
        }
    }
}
```

The above code will attempt to first identify a user by using the POST data. If successful we set the user info to the session so that it persists across requests and then redirect to either the last page they were visiting or a URL specified in the `loginRedirect` config. If the login is unsuccessful, a flash message is set.

Warning: `$this->Auth->setUser($data)` will log the user in with whatever data is passed to the method. It won't actually check the credentials against an authentication class.

Redirecting Users After Login

`AuthComponent::redirectUrl()`

After logging a user in, you'll generally want to redirect them back to where they came from. Pass a URL in to set the destination a user should be redirected to after logging in.

If no parameter is passed, the returned URL will use the following rules:

- Returns the normalized URL from the `redirect` query string value if it is present and for the same domain the current app is running on. Before 3.4.0, the `Auth.redirect` session value was used.
- If there is no query string/session value and there is a config `loginRedirect`, the `loginRedirect` value is returned.
- If there is no redirect value and no `loginRedirect`, `/` is returned.

Creating Stateless Authentication Systems

Basic and digest are stateless authentication schemes and don't require an initial POST or a form. If using only basic/digest authenticators you don't require a login action in your controller. Stateless authentication will re-verify the user's credentials on each request, this creates a small amount of additional overhead, but allows clients to login without using cookies and makes AuthComponent more suitable for building APIs.

For stateless authenticators, the `storage` config should be set to `Memory` so that AuthComponent does not use a session to store user record. You may also want to set config `unauthorizedRedirect` to `false` so that AuthComponent throws a `ForbiddenException` instead of the default behavior of redirecting to referrer.

Authentication objects can implement a `getUser()` method that can be used to support user login systems that don't rely on cookies. A typical getUser method looks at the request/environment and uses the information there to confirm the identity of the user. HTTP Basic authentication for example uses `$_SERVER['PHP_AUTH_USER']` and `$_SERVER['PHP_AUTH_PW']` for the username and password fields.

Note: In case authentication does not work like expected, check if queries are executed at all (see `BaseAuthenticate::_query($username)`). In case no queries are executed check if `$_SERVER['PHP_AUTH_USER']` and `$_SERVER['PHP_AUTH_PW']` do get populated by the webserver. If you are using Apache with FastCGI-PHP you might need to add this line to your **.htaccess** file in webroot:

```
RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization},L]
```

On each request, these values, `PHP_AUTH_USER` and `PHP_AUTH_PW`, are used to re-identify the user and ensure they are the valid user. As with authentication object's `authenticate()` method, the `getUser()` method should return an array of user information on the success or `false` on failure.

```
public function getUser(ServerRequest $request)
{
    $username = env('PHP_AUTH_USER');
    $pass = env('PHP_AUTH_PW');

    if (empty($username) || empty($pass)) {
        return false;
    }
    return $this->_findUser($username, $pass);
}
```

The above is how you could implement the getUser method for HTTP basic authentication. The `_findUser()` method is part of `BaseAuthenticate` and identifies a user based on a username and password.

Using Basic Authentication

Basic authentication allows you to create a stateless authentication that can be used in intranet applications or for simple API scenarios. Basic authentication credentials will be rechecked on each request.

> **Warning:** Basic authentication transmits credentials in plain-text. You should use HTTPS when using Basic authentication.

To use basic authentication, you'll need to configure AuthComponent:

```
$this->loadComponent('Auth', [
    'authenticate' => [
        'Basic' => [
            'fields' => ['username' => 'username', 'password' => 'api_key'],
            'userModel' => 'Users'
        ],
    ],
    'storage' => 'Memory',
    'unauthorizedRedirect' => false
]);
```

Here we're using username + API key as our fields and use the Users model.

Creating API Keys for Basic Authentication

Because basic HTTP sends credentials in plain-text, it is unwise to have users send their login password. Instead, an opaque API key is generally used. You can generate these API tokens randomly using libraries from CakePHP:

```
namespace App\Model\Table;

use Cake\Auth\DefaultPasswordHasher;
use Cake\Utility\Text;
use Cake\Event\Event;
use Cake\ORM\Table;

class UsersTable extends Table
{
    public function beforeSave(Event $event)
    {
        $entity = $event->getData('entity');

        if ($entity->isNew()) {
            $hasher = new DefaultPasswordHasher();

            // Generate an API 'token'
            $entity->api_key_plain = sha1(Text::uuid());

            // Bcrypt the token so BasicAuthenticate can check
            // it during login.
            $entity->api_key = $hasher->hash($entity->api_key_plain);
        }
        return true;
    }
}
```

The above generates a random hash for each user as they are saved. The above code assumes you have two columns `api_key` - to store the hashed API key, and `api_key_plain` - to the plaintext version of the API key, so we can display it to the user later on. Using a key instead of a password means that even over plain HTTP, your users can use an opaque token instead of their original password. It is also wise to include logic allowing API keys to be regenerated at a user's request.

Using Digest Authentication

Digest authentication offers an improved security model over basic authentication, as the user's credentials are never sent in the request header. Instead, a hash is sent.

To use digest authentication, you'll need to configure AuthComponent:

```
$this->loadComponent('Auth', [
    'authenticate' => [
        'Digest' => [
            'fields' => ['username' => 'username', 'password' => 'digest_hash'],
            'userModel' => 'Users'
        ],
    ],
    'storage' => 'Memory',
    'unauthorizedRedirect' => false
]);
```

Here we're using username + digest_hash as our fields and use the Users model.

Hashing Passwords For Digest Authentication

Because Digest authentication requires a password hashed in the format defined by the RFC, in order to correctly hash a password for use with Digest authentication you should use the special password hashing function on `DigestAuthenticate`. If you are going to be combining digest authentication with any other authentication strategies, it's also recommended that you store the digest password in a separate column, from the normal password hash:

```
namespace App\Model\Table;

use Cake\Auth\DigestAuthenticate;
use Cake\Event\Event;
use Cake\ORM\Table;

class UsersTable extends Table
{
    public function beforeSave(Event $event)
    {
        $entity = $event->getData('entity');

        // Make a password for digest auth.
        $entity->digest_hash = DigestAuthenticate::password(
            $entity->username,
            $entity->plain_password,
            env('SERVER_NAME')
        );
        return true;
    }
}
```

Passwords for digest authentication need a bit more information than other password hashes, based on the RFC for digest authentication.

Note: The third parameter of DigestAuthenticate::password() must match the 'realm' config value defined when DigestAuthentication was configured in AuthComponent::$authenticate. This defaults to `env('SCRIPT_NAME')`. You may wish to use a static string if you want consistent hashes in multiple environments.

Creating Custom Authentication Objects

Because authentication objects are pluggable, you can create custom authentication objects in your application or plugins. If for example, you wanted to create an OpenID authentication object. In **src/Auth/OpenidAuthenticate.php** you could put the following:

```
namespace App\Auth;

use Cake\Auth\BaseAuthenticate;
use Cake\Http\ServerRequest;
use Cake\Http\Response;

class OpenidAuthenticate extends BaseAuthenticate
{
    public function authenticate(ServerRequest $request, Response $response)
    {
        // Do things for OpenID here.
        // Return an array of user if they could authenticate the user,
        // return false if not.
    }
}
```

Authentication objects should return `false` if they cannot identify the user and an array of user information if they can. It's not required that you extend `BaseAuthenticate`, only that your authentication object implements `Cake\Event\EventListenerInterface`. The `BaseAuthenticate` class provides a number of helpful methods that are commonly used. You can also implement a `getUser()` method if your authentication object needs to support stateless or cookie-less authentication. See the sections on basic and digest authentication below for more information.

`AuthComponent` triggers two events, `Auth.afterIdentify` and `Auth.logout`, after a user has been identified and before a user is logged out respectively. You can set callback functions for these events by returning a mapping array from `implementedEvents()` method of your authenticate class:

```
public function implementedEvents()
{
    return [
        'Auth.afterIdentify' => 'afterIdentify',
        'Auth.logout' => 'logout'
    ];
}
```

Using Custom Authentication Objects

Once you've created your custom authentication object, you can use them by including them in AuthComponents authenticate array:

```
$this->Auth->config('authenticate', [
    'Openid', // app authentication object.
    'AuthBag.Openid', // plugin authentication object.
]);
```

Note: Note that when using simple notation there's no 'Authenticate' word when initiating the authentication object. Instead, if using namespaces, you'll need to set the full namespace of the class, including the 'Authenticate' word.

Handling Unauthenticated Requests

When an unauthenticated user tries to access a protected page first the `unauthenticated()` method of the last authenticator in the chain is called. The authenticate object can handle sending response or redirection by returning a response object to indicate no further action is necessary. Due to this, the order in which you specify the authentication provider in `authenticate` config matters.

If authenticator returns null, AuthComponent redirects user to the login action. If it's an AJAX request and config `ajaxLogin` is specified that element is rendered else a 403 HTTP status code is returned.

Displaying Auth Related Flash Messages

In order to display the session error messages that Auth generates, you need to add the following code to your layout. Add the following two lines to the **src/Template/Layout/default.ctp** file in the body section:

```
// Only this is necessary after 3.4.0
echo $this->Flash->render();

// Prior to 3.4.0 this will be required as well.
echo $this->Flash->render('auth');
```

You can customize the error messages and flash settings AuthComponent uses. Using `flash` config you can configure the parameters AuthComponent uses for setting flash messages. The available keys are

- `key` - The key to use, defaults to 'default'. Prior to 3.4.0, the key defaulted to 'auth'.
- `element` - The element name to use for rendering, defaults to null.
- `params` - The array of additional params to use, defaults to `[]`.

In addition to the flash message settings you can customize other error messages AuthComponent uses. In your controller's beforeFilter, or component settings you can use `authError` to customize the error used for when authorization fails:

```
$this->Auth->config('authError', "Woopsie, you are not authorized to access this area.
↪");
```

Sometimes, you want to display the authorization error only after the user has already logged-in. You can suppress this message by setting its value to boolean `false`.

In your controller's beforeFilter() or component settings:

```
if (!$this->Auth->user()) {
    $this->Auth->config('authError', false);
}
```

Hashing Passwords

You are responsible for hashing the passwords before they are persisted to the database, the easiest way is to use a setter function in your User entity:

```
namespace App\Model\Entity;

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

```
class User extends Entity
{

    // ...

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

    // ...
}
```

AuthComponent is configured by default to use the `DefaultPasswordHasher` when validating user credentials so no additional configuration is required in order to authenticate users.

`DefaultPasswordHasher` uses the bcrypt hashing algorithm internally, which is one of the stronger password hashing solutions used in the industry. While it is recommended that you use this password hasher class, the case may be that you are managing a database of users whose password was hashed differently.

Creating Custom Password Hasher Classes

In order to use a different password hasher, you need to create the class in **src/Auth/LegacyPasswordHasher.php** and implement the `hash()` and `check()` methods. This class needs to extend the `AbstractPasswordHasher` class:

```
namespace App\Auth;

use Cake\Auth\AbstractPasswordHasher;

class LegacyPasswordHasher extends AbstractPasswordHasher
{

    public function hash($password)
    {
        return sha1($password);
    }

    public function check($password, $hashedPassword)
    {
        return sha1($password) === $hashedPassword;
    }
}
```

Then you are required to configure the AuthComponent to use your own password hasher:

```
public function initialize()
{
    parent::initialize();
    $this->loadComponent('Auth', [
        'authenticate' => [
            'Form' => [
                'passwordHasher' => [
                    'className' => 'Legacy',
```

```
                ]
            ]
        ]
    ]);
}
```

Supporting legacy systems is a good idea, but it is even better to keep your database with the latest security advancements. The following section will explain how to migrate from one hashing algorithm to CakePHP's default.

Changing Hashing Algorithms

CakePHP provides a clean way to migrate your users' passwords from one algorithm to another, this is achieved through the `FallbackPasswordHasher` class. Assuming you are migrating your app from CakePHP 2.x which uses `sha1` password hashes, you can configure the AuthComponent as follows:

```
public function initialize()
{
    parent::initialize();
    $this->loadComponent('Auth', [
        'authenticate' => [
            'Form' => [
                'passwordHasher' => [
                    'className' => 'Fallback',
                    'hashers' => [
                        'Default',
                        'Weak' => ['hashType' => 'sha1']
                    ]
                ]
            ]
        ]
    ]);
}
```

The first name appearing in the `hashers` key indicates which of the classes is the preferred one, but it will fallback to the others in the list if the check was unsuccessful.

When using the `WeakPasswordHasher` you will need to set the `Security.salt` configure the value to ensure passwords are salted.

In order to update old users' passwords on the fly, you can change the login function accordingly:

```
public function login()
{
    if ($this->request->is('post')) {
        $user = $this->Auth->identify();
        if ($user) {
            $this->Auth->setUser($user);
            if ($this->Auth->authenticationProvider()->needsPasswordRehash()) {
                $user = $this->Users->get($this->Auth->user('id'));
                $user->password = $this->request->getData('password');
                $this->Users->save($user);
            }
            return $this->redirect($this->Auth->redirectUrl());
        }
        ...
    }
}
```

As you can see we are just setting the plain password again so the setter function in the entity will hash the password as shown in the previous example and then save the entity.

Manually Logging Users In

`AuthComponent::setUser(array $user)`

Sometimes the need arises where you need to manually log a user in, such as just after they registered for your application. You can do this by calling `$this->Auth->setUser()` with the user data you want to 'login':

```
public function register()
{
    $user = $this->Users->newEntity($this->request->getData());
    if ($this->Users->save($user)) {
        $this->Auth->setUser($user->toArray());
        return $this->redirect([
            'controller' => 'Users',
            'action' => 'home'
        ]);
    }
}
```

Warning: Be sure to manually add the new User id to the array passed to the `setUser()` method. Otherwise, you won't have the user id available.

Accessing the Logged In User

`AuthComponent::user($key = null)`

Once a user is logged in, you will often need some particular information about the current user. You can access the currently logged in user using `AuthComponent::user()`:

```
// From inside a controller or other component.
$this->Auth->user('id');
```

If the current user is not logged in or the key doesn't exist, null will be returned.

Logging Users Out

`AuthComponent::logout()`

Eventually, you'll want a quick way to de-authenticate someone and redirect them to where they need to go. This method is also useful if you want to provide a 'Log me out' link inside a members' area of your application:

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

Logging out users that logged in with Digest or Basic auth is difficult to accomplish for all clients. Most browsers will retain credentials for the duration they are still open. Some clients can be forced to logout by sending a 401 status code. Changing the authentication realm is another solution that works for some clients.

Deciding When to run Authentication

In some cases you may want to use `$this->Auth->user()` in the `beforeFilter(Event $event)` method. This is achievable by using the `checkAuthIn` config key. The following changes which event for which initial authentication checks should be done:

```
//Set up AuthComponent to authenticate in initialize()
$this->Auth->config('checkAuthIn', 'Controller.initialize');
```

Default value for `checkAuthIn` is `'Controller.startup'` - but by using `'Controller.initialize'` initial authentication is done before `beforeFilter()` method.

Authorization

Authorization is the process of ensuring that an identified/authenticated user is allowed to access the resources they are requesting. If enabled `AuthComponent` can automatically check authorization handlers and ensure that logged in users are allowed to access the resources they are requesting. There are several built-in authorization handlers and you can create custom ones for your application or as part of a plugin.

- `ControllerAuthorize` Calls `isAuthorized()` on the active controller, and uses the return of that to authorize a user. This is often the most simple way to authorize users.

Note: The `ActionsAuthorize` & `CrudAuthorize` adapter available in CakePHP 2.x have now been moved to a separate plugin cakephp/acl[109].

Configuring Authorization Handlers

You configure authorization handlers using the `authorize` config key. You can configure one or many handlers for authorization. Using multiple handlers allows you to support different ways of checking authorization. When authorization handlers are checked, they will be called in the order they are declared. Handlers should return `false`, if they are unable to check authorization, or the check has failed. Handlers should return `true` if they were able to check authorization successfully. Handlers will be called in sequence until one passes. If all checks fail, the user will be redirected to the page they came from. Additionally, you can halt all authorization by throwing an exception. You will need to catch any thrown exceptions and handle them.

You can configure authorization handlers in your controller's `beforeFilter()` or `initialize()` methods. You can pass configuration information into each authorization object, using an array:

```
// Basic setup
$this->Auth->config('authorize', ['Controller']);

// Pass settings in
$this->Auth->config('authorize', [
    'Actions' => ['actionPath' => 'controllers/'],
    'Controller'
]);
```

Much like `authenticate`, `authorize`, helps you keep your code DRY, by using the `all` key. This special key allows you to set settings that are passed to every attached object. The `all` key is also exposed as `AuthComponent::ALL`:

[109] https://github.com/cakephp/acl

```
// Pass settings in using 'all'
$this->Auth->config('authorize', [
    AuthComponent::ALL => ['actionPath' => 'controllers/'],
    'Actions',
    'Controller'
]);
```

In the above example, both the `Actions` and `Controller` will get the settings defined for the 'all' key. Any settings passed to a specific authorization object will override the matching key in the 'all' key.

If an authenticated user tries to go to a URL he's not authorized to access, he's redirected back to the referrer. If you do not want such redirection (mostly needed when using stateless authentication adapter) you can set config option `unauthorizedRedirect` to `false`. This causes AuthComponent to throw a `ForbiddenException` instead of redirecting.

Creating Custom Authorize Objects

Because authorize objects are pluggable, you can create custom authorize objects in your application or plugins. If for example, you wanted to create an LDAP authorize object. In **src/Auth/LdapAuthorize.php** you could put the following:

```
namespace App\Auth;

use Cake\Auth\BaseAuthorize;
use Cake\Http\ServerRequest;

class LdapAuthorize extends BaseAuthorize
{
    public function authorize($user, ServerRequest $request)
    {
        // Do things for ldap here.
    }
}
```

Authorize objects should return `false` if the user is denied access, or if the object is unable to perform a check. If the object is able to verify the user's access, `true` should be returned. It's not required that you extend `BaseAuthorize`, only that your authorize object implements an `authorize()` method. The `BaseAuthorize` class provides a number of helpful methods that are commonly used.

Using Custom Authorize Objects

Once you've created your custom authorize object, you can use them by including them in your AuthComponent's authorize array:

```
$this->Auth->config('authorize', [
    'Ldap', // app authorize object.
    'AuthBag.Combo', // plugin authorize object.
]);
```

Using No Authorization

If you'd like to not use any of the built-in authorization objects and want to handle things entirely outside of AuthComponent, you can set `$this->Auth->config('authorize',false);`. By default AuthComponent starts off

with `authorize` set to `false`. If you don't use an authorization scheme, make sure to check authorization yourself in your controller's beforeFilter or with another component.

Making Actions Public

`AuthComponent::allow`(*$actions = null*)

There are often times controller actions that you wish to remain entirely public or that don't require users to be logged in. AuthComponent is pessimistic and defaults to denying access. You can mark actions as public actions by using `AuthComponent::allow()`. By marking actions as public, AuthComponent will not check for a logged in user nor will authorize objects to be checked:

```
// Allow all actions
$this->Auth->allow();

// Allow only the index action.
$this->Auth->allow('index');

// Allow only the view and index actions.
$this->Auth->allow(['view', 'index']);
```

By calling it empty you allow all actions to be public. For a single action, you can provide the action name as a string. Otherwise, use an array.

Note: You should not add the "login" action of your `UsersController` to allow list. Doing so would cause problems with the normal functioning of `AuthComponent`.

Making Actions Require Authorization

`AuthComponent::deny`(*$actions = null*)

By default all actions require authorization. However, after making actions public you want to revoke the public access. You can do so using `AuthComponent::deny()`:

```
// Deny all actions.
$this->Auth->deny();

// Deny one action
$this->Auth->deny('add');

// Deny a group of actions.
$this->Auth->deny(['add', 'edit']);
```

By calling it empty you deny all actions. For a single action, you can provide the action name as a string. Otherwise, use an array.

Using ControllerAuthorize

ControllerAuthorize allows you to handle authorization checks in a controller callback. This is ideal when you have very simple authorization or you need to use a combination of models and components to do your authorization and don't want to create a custom authorize object.

The callback is always called `isAuthorized()` and it should return a boolean as to whether or not the user is allowed to access resources in the request. The callback is passed the active user so it can be checked:

```
class AppController extends Controller
{
    public function initialize()
    {
        parent::initialize();
        $this->loadComponent('Auth', [
            'authorize' => 'Controller',
        ]);
    }

    public function isAuthorized($user = null)
    {
        // Any registered user can access public functions
        if (!$this->request->getParam('prefix')) {
            return true;
        }

        // Only admins can access admin functions
        if ($this->request->getParam('prefix') === 'admin') {
            return (bool)($user['role'] === 'admin');
        }

        // Default deny
        return false;
    }
}
```

The above callback would provide a very simple authorization system where only users with role = admin could access actions that were in the admin prefix.

Configuration options

The following settings can all be defined either in your controller's `initialize()` method or using `$this->Auth->config()` in your `beforeFilter()`:

ajaxLogin The name of an optional view element to render when an AJAX request is made with an invalid or expired session.

allowedActions Controller actions for which user validation is not required.

authenticate Set to an array of Authentication objects you want to use when logging users in. There are several core authentication objects; see the section on *Suggested Reading Before Continuing*.

authError Error to display when user attempts to access an object or action to which they do not have access.

You can suppress authError message from being displayed by setting this value to boolean `false`.

authorize Set to an array of Authorization objects you want to use when authorizing users on each request; see the section on *Authorization*.

flash Settings to use when Auth needs to do a flash message with `FlashComponent::set()`. Available keys are:

- `element` - The element to use; defaults to 'default'.
- `key` - The key to use; defaults to 'auth'.
- `params` - The array of additional params to use; defaults to [].

loginAction A URL (defined as a string or array) to the controller action that handles logins. Defaults to `/users/login`.

loginRedirect The URL (defined as a string or array) to the controller action users should be redirected to after logging in. This value will be ignored if the user has an `Auth.redirect` value in their session.

logoutRedirect The default action to redirect to after the user is logged out. While AuthComponent does not handle post-logout redirection, a redirect URL will be returned from *`AuthComponent::logout()`*. Defaults to `loginAction`.

unauthorizedRedirect Controls handling of unauthorized access. By default unauthorized user is redirected to the referrer URL or `loginAction` or '/'. If set to `false`, a ForbiddenException exception is thrown instead of redirecting.

storage Storage class to use for persisting user record. When using stateless authenticator you should set this to `Memory`. Defaults to `Session`. You can pass config options to storage class using array format. For e.g. to use a custom session key you can set `storage` to `['className' => 'Session','key' => 'Auth.Admin']`.

checkAuthIn Name of the event in which initial auth checks should be done. Defaults to `Controller.startup`. You can set it to `Controller.initialize` if you want the check to be done before controller's `beforeFilter()` method is run.

You can get current configuration values by calling `$this->Auth->config()`:: only the configuration option:

```
$this->Auth->config('loginAction');

$this->redirect($this->Auth->config('loginAction'));
```

This is useful if you want to redirect a user to the `login` route for example. Without a parameter, the full configuration will be returned.

Testing Actions Protected By AuthComponent

See the *Testing Actions That Require Authentication* section for tips on how to test controller actions that are protected by `AuthComponent`.

Cookie

class `Cake\Controller\Component\`**`CookieComponent`**(*ComponentRegistry $collection*, *array $config = []*)

The CookieComponent is a wrapper around the native PHP `setcookie()` method. It makes it easier to manipulate cookies, and automatically encrypt cookie data. Cookies added through CookieComponent will only be sent if the controller action completes.

Deprecated since version 3.5.0: You should use encrypted-cookie-middleware instead of `CookieComponent`.

Configuring Cookies

Cookies can be configured either globally or per top-level name. The global configuration data will be merged with the top-level configuration. So only need to override the parts that are different. To configure the global settings use the `config()` method:

```
$this->Cookie->config('path', '/');
$this->Cookie->config([
    'expires' => '+10 days',
    'httpOnly' => true
]);
```

To configure a specific key use the `configKey()` method:

```
$this->Cookie->configKey('User', 'path', '/');
$this->Cookie->configKey('User', [
    'expires' => '+10 days',
    'httpOnly' => true
]);
```

There are a number of configurable values for cookies:

expires How long the cookies should last for. Defaults to 1 month.

path The path on the server in which the cookie will be available on. If path is set to '/foo/', the cookie will only be available within the /foo/ directory and all sub-directories such as /foo/bar/ of domain. The default value is app's base path.

domain The domain that the cookie is available. To make the cookie available on all subdomains of example.com set domain to '.example.com'.

secure Indicates that the cookie should only be transmitted over a secure HTTPS connection. When set to `true`, the cookie will only be set if a secure connection exists.

key Encryption key used when encrypted cookies are enabled. Defaults to Security.salt.

httpOnly Set to `true` to make HTTP only cookies. Cookies that are HTTP only are not accessible in JavaScript. Defaults to `false`.

encryption Type of encryption to use. Defaults to 'aes'. Can also be 'rijndael' for backwards compatibility.

Using the Component

The CookieComponent offers a number of methods for working with Cookies.

`Cake\Controller\Component\CookieComponent::write`(*mixed $key*, *mixed $value = null*)

The write() method is the heart of the cookie component. $key is the cookie variable name you want, and the $value is the information to be stored:

```
$this->Cookie->write('name', 'Larry');
```

You can also group your variables by using dot notation in the key parameter:

```
$this->Cookie->write('User.name', 'Larry');
$this->Cookie->write('User.role', 'Lead');
```

If you want to write more than one value to the cookie at a time, you can pass an array:

```
$this->Cookie->write('User',
    ['name' => 'Larry', 'role' => 'Lead']
);
```

All values in the cookie are encrypted with AES by default. If you want to store the values as plain text, be sure to configure the key space:

```
$this->Cookie->configKey('User', 'encryption', false);
```

Cake\Controller\Component\CookieComponent::**read**(*mixed $key = null*)

This method is used to read the value of a cookie variable with the name specified by $key.

```
// Outputs "Larry"
echo $this->Cookie->read('name');

// You can also use the dot notation for read
echo $this->Cookie->read('User.name');

// To get the variables which you had grouped
// using the dot notation as an array use the following
$this->Cookie->read('User');

// This outputs something like ['name' => 'Larry', 'role' => 'Lead']
```

> **Warning:** CookieComponent cannot interact with bare strings values that contain `,`. The component will attempt to interpret these values as arrays, leading to incorrect results. Instead you should use `$request->getCookie()`.

Cake\Controller\Component\CookieComponent::**check**(*$key*)

Parameters

- **$key** (*string*) – The key to check.

Used to check whether a key/path exists and has a non-null value.

Cake\Controller\Component\CookieComponent::**delete**(*mixed $key*)

Deletes a cookie variable of the name in $key. Works with dot notation:

```
// Delete a variable
$this->Cookie->delete('bar');

// Delete the cookie variable bar, but not everything under foo
$this->Cookie->delete('foo.bar');
```

Cross Site Request Forgery

By enabling the CSRF Component you get protection against attacks. CSRF[110] or Cross Site Request Forgery is a common vulnerability in web applications. It allows an attacker to capture and replay a previous request, and sometimes submit data requests using image tags or resources on other domains.

The CsrfComponent works by setting a cookie to the user's browser. When forms are created with the *Cake\View\Helper\FormHelper*, a hidden field is added containing the CSRF token. During the `Controller.startup` event, if the request is a POST, PUT, DELETE, PATCH request the component will compare the request data & cookie value. If either is missing or the two values mismatch the component will throw a *Cake\Network\Exception\InvalidCsrfTokenException*.

Note: You should always verify the HTTP method being used before executing side-effects. You should *check the HTTP method* or use *Cake\Http\ServerRequest::allowMethod()* to ensure the correct HTTP method is

[110] http://en.wikipedia.org/wiki/Cross-site_request_forgery

used.

New in version 3.1: The exception type changed from *Cake\Network\Exception\ForbiddenException* to *Cake\Network\Exception\InvalidCsrfTokenException*.

Deprecated since version 3.5.0: You should use csrf-middleware instead of `CsrfComponent`.

Using the CsrfComponent

Simply by adding the `CsrfComponent` to your components array, you can benefit from the CSRF protection it provides:

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

Settings can be passed into the component through your component's settings. 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. Accepts `strtotime` values as of 3.1
- `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`.
- `field` The form field to check. Defaults to `_csrfToken`. Changing this will also require configuring FormHelper.

When enabled, you can access the current CSRF token on the request object:

```
$token = $this->request->getParam('_csrfToken');
```

Integration with FormHelper

The CsrfComponent 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 the CsrfComponent 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.

Disabling the CSRF Component for Specific Actions

While not recommended, you may want to disable the CsrfComponent on certain requests. You can do this using the controller's event dispatcher, during the `beforeFilter()` method:

```
public function beforeFilter(Event $event)
{
    $this->eventManager()->off($this->Csrf);
}
```

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 **src/Template/Element/Flash** directory. By convention, camelcased methods will map to the lowercased and underscored element name:

```
// Uses src/Template/Element/Flash/success.ctp
$this->Flash->success('This was successful');

// Uses src/Template/Element/Flash/great_success.ctp
$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');
```

New in version 3.1: Flash messages now stack. Successive calls to `set()` or `__call()` with the same key will append the messages in the `$_SESSION`. If you want to keep the old behavior (one message even after consecutive calls), set the `clear` parameter 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.

New in version 3.1: A new key `clear` was added. This key 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',
```

```
    'params' => [
        'name' => $user->name,
        'email' => $user->email
    ]
]);

// In your View
<?= $this->Flash->render('positive') ?>

<!-- In src/Template/Element/Flash/success.ctp -->
<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.ctp** element under **plugins/PluginName/src/Template/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

New in version 3.3.3.

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.

Security

class `SecurityComponent` (*ComponentCollection $collection*, *array $config = []*)

The Security Component creates an easy way to integrate tighter security in your application. It provides methods for various tasks like:

- Restricting which HTTP methods your application accepts.
- Form tampering protection
- Requiring that SSL be used.
- Limiting cross controller communication.

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 beforeFilter.

By using the Security Component you automatically get form tampering protection. Hidden token fields will automatically be inserted into forms and checked by the Security component.

If you are using Security component's form protection features and other components that process form data in their `startup()` callbacks, be sure to place Security Component before those components in your `initialize()` method.

Note: When using the Security Component you **must** use the FormHelper to create your forms. In addition, you must **not** override any of the fields' "name" attributes. The Security Component looks for certain indicators that are created and managed by the FormHelper (especially those created in *View\Helper\FormHelper::create()* and *View\Helper\FormHelper::end()*). Dynamically altering the fields that are submitted in a POST request (e.g. disabling, deleting or creating new fields via JavaScript) is likely to cause the request to be send to the blackhole callback.

You should always verify the HTTP method being used before executing side-effects. You should *check the HTTP method* or use *Cake\Http\ServerRequest::allowMethod()* to ensure the correct HTTP method is used.

Handling Blackhole Callbacks

`SecurityComponent`**::blackHole**(*object $controller*, *string $error = ''*, *SecurityException $exception = null*)

If an action is restricted by the Security Component it is 'black-holed' as an invalid request which will result in a 400 error by default. You can configure this behavior by setting the `blackHoleCallback` configuration option to a callback function in the controller.

By configuring a callback method you can customize how the blackhole process works:

```
public function beforeFilter(Event $event)
{
    $this->Security->setConfig('blackHoleCallback', 'blackhole');
}

public function blackhole($type)
{
    // Handle errors.
}
```

Note: use `$this->Security->config()` for CakePHP versions prior to 3.4

The `$type` parameter can have the following values:

- 'auth' Indicates a form validation error, or a controller/action mismatch error.
- 'secure' Indicates an SSL method restriction failure.

New in version cakephp/cakephp: 3.2.6

As of v3.2.6 an additional parameter is included in the blackHole callback, an instance of the `Cake\Controller\Exception\SecurityException` is included as a second parameter.

Restrict Actions to SSL

`SecurityComponent::requireSecure()`
: Sets the actions that require a SSL-secured request. Takes any number of arguments. Can be called with no arguments to force all actions to require a SSL-secured.

`SecurityComponent::requireAuth()`
: Sets the actions that require a valid Security Component generated token. Takes any number of arguments. Can be called with no arguments to force all actions to require a valid authentication.

Restricting Cross Controller Communication

allowedControllers A list of controllers which can send requests to this controller. This can be used to control cross controller requests.

allowedActions A list of actions which are allowed to send requests to this controller's actions. This can be used to control cross controller requests.

These configuration options allow you to restrict cross controller communication. Set them with the `setConfig()` method, or `config()` if you are using a CakePHP version below 3.4.

Form Tampering Prevention

By default the `SecurityComponent` prevents users from tampering with forms in specific ways. The `SecurityComponent` will prevent the following things:

- 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. When a form is submitted, the `SecurityComponent` will use the POST data to build the same structure and compare the hash.

Note: The SecurityComponent will **not** prevent select options from being added/changed. Nor will it prevent radio options from being added/changed.

---

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.

validatePost Set to `false` to completely skip the validation of POST requests, essentially turning off form validation.

The above configuration options can be set with `setConfig()` or `config()` for CakePHP versions below 3.4.

Usage

Using the security component is generally done in the controllers `beforeFilter()`. You would specify the security restrictions you want and the Security Component will enforce them on its startup:

```
namespace App\Controller;

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

class WidgetsController extends AppController
{

    public function initialize()
    {
        parent::initialize();
        $this->loadComponent('Security');
    }

    public function beforeFilter(Event $event)
    {
        if ($this->request->getParam('admin')) {
            $this->Security->requireSecure();
        }
    }
}
```

The above example would force all actions that had admin routing to require secure SSL requests:

```
namespace App\Controller;

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

class WidgetsController extends AppController
{

    public function initialize()
    {
        parent::initialize();
        $this->loadComponent('Security', ['blackHoleCallback' => 'forceSSL']);
    }

    public function beforeFilter(Event $event)
    {
        if ($this->request->getParam('admin')) {
            $this->Security->requireSecure();
        }
    }

    public function forceSSL()
    {
        return $this->redirect('https://' . env('SERVER_NAME') . $this->request->
↪here());
    }
}
```

This example would force all actions that had admin routing to require secure SSL requests. When the request is black holed, it will call the nominated `forceSSL()` callback which will redirect non-secure requests to secure requests automatically.

CSRF Protection

CSRF or Cross Site Request Forgery is a common vulnerability in web applications. It allows an attacker to capture and replay a previous request, and sometimes submit data requests using image tags or resources on other domains. To enable CSRF protection features use the *Cross Site Request Forgery*.

Disabling Security Component for Specific Actions

There may be cases where you want to disable all security checks for an action (ex. AJAX requests). You may "unlock" these actions by listing them in `$this->Security->unlockedActions` in your `beforeFilter()`. The `unlockedActions` property will **not** affect other features of `SecurityComponent`:

```
namespace App\Controller;

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

class WidgetController extends AppController
{

    public function initialize()
    {
        parent::initialize();
        $this->loadComponent('Security');
    }

    public function beforeFilter(Event $event)
    {
         $this->Security->setConfig('unlockedActions', ['edit']);
    }
}
```

Note: use `$this->Security->config()` for CakePHP versions prior to 3.4.0

This example would disable all security checks for the edit action.

Pagination

class `Cake\Controller\Component\`**`PaginatorComponent`**

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 quick, easy way to paginate data.

Pagination in CakePHP is offered by a component in the controller, to make building paginated queries easier. In the View *`View\Helper\PaginatorHelper`* is used to make the generation of pagination links & buttons simple.

Using Controller::paginate()

In the controller, we start by defining the default query conditions pagination will use in the `$paginate` controller variable. These conditions, serve as the basis for your pagination queries. They are augmented by the `sort`, `direction` `limit`, and `page` parameters passed in from the URL. It is important to note that the `order` key must be defined in an array structure like below:

```
class ArticlesController extends AppController
{

    public $paginate = [
        'limit' => 25,
        'order' => [
            'Articles.title' => 'asc'
        ]
    ];

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

You can also include any of the options supported by *ORM\Table::find()*, such as `fields`:

```
class ArticlesController extends AppController
{

    public $paginate = [
        'fields' => ['Articles.id', 'Articles.created'],
        'limit' => 25,
        'order' => [
            'Articles.title' => 'asc'
        ]
    ];

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

While you can pass most of the query options from the paginate property it is often cleaner and simpler to bundle up your pagination options into a *Custom Finder Methods*. You can define the finder pagination uses by setting the `finder` option:

```
class ArticlesController extends AppController
{

    public $paginate = [
        'finder' => 'published',
    ];
}
```

Because custom finder methods can also take in options, this is how you pass in options into a custom finder method within the paginate property:

```
class ArticlesController extends AppController
{

    // find articles by tag
    public function tags()
    {
        $tags = $this->request->getParam('pass');

        $customFinderOptions = [
            'tags' => $tags
        ];
        // the custom finder method is called findTagged inside ArticlesTable.php
        // it should look like this:
        // public function findTagged(Query $query, array $options) {
        // hence you use tagged as the key
        $this->paginate = [
            'finder' => [
                'tagged' => $customFinderOptions
            ]
        ];
        $articles = $this->paginate($this->Articles);
        $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, you just name the keys of the array after the model you wish to configure:

```
class ArticlesController extends AppController
{

    public $paginate = [
        'Articles' => [],
        'Authors' => [],
    ];
}
```

The values of the `Articles` and `Authors` keys could contain all the properties that a model/key less `$paginate` array could.

Once the `$paginate` property has been defined, we can use the *`Controller\Controller::paginate()`* method to create the pagination data, and add the `PaginatorHelper` if it hasn't already been added. The controller's paginate method will return the result set of the paginated query, and set pagination metadata to the request. You can access the pagination metadata at `$this->request->getParam('paging')`. A more complete example of using `paginate()` would be:

```
class ArticlesController extends AppController
{

    public function index()
    {
        $this->set('articles', $this->paginate());
    }
}
```

By default the `paginate()` method will use the default model for a controller. You can also pass the resulting query of a find method:

```
public function index()
{
    $query = $this->Articles->find('popular')->where(['author_id' => 1]);
    $this->set('articles', $this->paginate($query));
}
```

If you want to paginate a different model you can provide a query for it, the table object itself, or its name:

```
// Using a query
$comments = $this->paginate($commentsTable->find());

// Using the model name.
$comments = $this->paginate('Comments');

// Using a table object.
$comments = $this->paginate($commentTable);
```

Using the Paginator Directly

If you need to paginate data from another component you may want to use the PaginatorComponent directly. It features a similar API to the controller method:

```
$articles = $this->Paginator->paginate($articleTable->find(), $config);

// Or
$articles = $this->Paginator->paginate($articleTable, $config);
```

The first parameter should be the query object from a find on table object you wish to paginate results from. Optionally, you can pass the table object and let the query be constructed for you. The second parameter should be the array of settings to use for pagination. This array should have the same structure as the `$paginate` property on a controller. When paginating a `Query` object, the `finder` option will be ignored. It is assumed that you are passing in the query you want paginated.

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
public $paginate = [
    'Articles' => ['scope' => 'article'],
    'Tags' => ['scope' => 'tag']
];

// 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 `PaginatorComponent` 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.

New in version 3.3.0: Multiple Pagination was added in 3.3.0

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 whitelist of fields that can be sorted using the `sortWhitelist` 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:

```
public $paginate = [
    'sortWhitelist' => [
        'id', 'title', 'Users.username', 'created'
    ]
];
```

Any requests that attempt to sort on fields not in the whitelist 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`:

```
public $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.

Joining Additional Associations

Additional associations can be loaded to the paginated table by using the `contain` parameter:

```
public function index()
{
    $this->paginate = [
        'contain' => ['Authors', 'Comments']
    ];

    $this->set('articles', $this->paginate($this->Articles));
}
```

Out of Range Page Requests

The PaginatorComponent 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\Network\Exception\NotFoundException;

public function index()
{
    try {
        $this->paginate();
    } catch (NotFoundException $e) {
        // Do something here like redirecting to first or last page.
        // $this->request->getParam('paging') will give you required info.
    }
}
```

Pagination in the View

Check the *View\Helper\PaginatorHelper* documentation for how to create links for pagination navigation.

Request Handling

class RequestHandlerComponent (*ComponentCollection $collection*, *array $config = []*)

The Request Handler component is used in CakePHP to obtain additional information about the HTTP requests that are made to your application. You can use it to see what content types clients prefer, automatcally parse request input, define how content types map to view classes or template paths.

By default RequestHandler will automatically detect AJAX requests based on the `X-Requested-With` HTTP header that many JavaScript libraries use. When used in conjunction with *Cake\Routing\Router::extensions()*, RequestHandler will automatically switch the layout and template files to those that match non-HTML media types. Furthermore, if a helper with the same name as the requested extension exists, it will be added to the Controllers Helper array. Lastly, if XML/JSON data is POST'ed to your Controllers, it will be parsed into an array which is assigned to `$this->request->getData()`, and can then be accessed as you would standard POST data. In order to make use of RequestHandler it must be included in your `initialize()` method:

```
class WidgetsController extends AppController
{
    public function initialize()
    {
        parent::initialize();
        $this->loadComponent('RequestHandler');
    }

    // Rest of controller
}
```

Obtaining Request Information

Request Handler has several methods that provide information about the client and its request.

`RequestHandlerComponent::`**accepts** (*$type = null*)

$type can be a string, or an array, or null. If a string, accepts will return `true` if the client accepts the content

type. If an array is specified, accepts return `true` if any one of the content types is accepted by the client. If null returns an array of the content-types that the client accepts. For example:

```
class ArticlesController extends AppController
{

    public function initialize()
    {
        parent::initialize();
        $this->loadComponent('RequestHandler');
    }

    public function beforeFilter(Event $event)
    {
        if ($this->RequestHandler->accepts('html')) {
            // Execute code only if client accepts an HTML (text/html)
            // response.
        } elseif ($this->RequestHandler->accepts('xml')) {
            // Execute XML-only code
        }
        if ($this->RequestHandler->accepts(['xml', 'rss', 'atom'])) {
            // Executes if the client accepts any of the above: XML, RSS
            // or Atom.
        }
    }
}
```

Other request 'type' detection methods include:

`RequestHandlerComponent::isXml()`
Returns `true` if the current request accepts XML as a response.

`RequestHandlerComponent::isRss()`
Returns `true` if the current request accepts RSS as a response.

`RequestHandlerComponent::isAtom()`
Returns `true` if the current call accepts an Atom response, false otherwise.

`RequestHandlerComponent::isMobile()`
Returns `true` if user agent string matches a mobile web browser, or if the client accepts WAP content. The supported Mobile User Agent strings are:

- Android
- AvantGo
- BlackBerry
- DoCoMo
- Fennec
- iPad
- iPhone
- iPod
- J2ME
- MIDP
- NetFront

- Nokia
- Opera Mini
- Opera Mobi
- PalmOS
- PalmSource
- portalmmm
- Plucker
- ReqwirelessWeb
- SonyEricsson
- Symbian
- UP.Browser
- webOS
- Windows CE
- Windows Phone OS
- Xiino

`RequestHandlerComponent::isWap()`
Returns `true` if the client accepts WAP content.

All of the above request detection methods can be used in a similar fashion to filter functionality intended for specific content types. For example when responding to AJAX requests, you often will want to disable browser caching, and change the debug level. However, you want to allow caching for non-AJAX requests. The following would accomplish that:

```
if ($this->request->is('ajax')) {
    $this->response->disableCache();
}
// Continue Controller action
```

Automatically Decoding Request Data

Add a request data decoder. The handler should contain a callback, and any additional arguments for the callback. The callback should return an array of data contained in the request input. For example adding a CSV handler could look like:

```
class ArticlesController extends AppController
{
    public function initialize()
    {
        parent::initialize();
        $parser = function ($data) {
            $rows = str_getcsv($data, "\n");
            foreach ($rows as &$row) {
                $row = str_getcsv($row, ',');
            }
            return $rows;
        };
        $this->loadComponent('RequestHandler', [
```

```
            'inputTypeMap' => [
                'csv' => [$parser]
            ]
        ]);
    }
}
```

You can use any callable[111] for the handling function. You can also pass additional arguments to the callback, this is useful for callbacks like `json_decode`:

```
$this->RequestHandler->addInputType('json', ['json_decode', true]);

// After 3.1.0 you should use
$this->RequestHandler->config('inputTypeMap.json', ['json_decode', true]);
```

The above will make `$this->request->getData()` an array of the JSON input data, without the additional `true` you'd get a set of `stdClass` objects.

Deprecated since version 3.1.0: As of 3.1.0 the `addInputType()` method is deprecated. You should use `config()` to add input types at runtime.

Checking Content-Type Preferences

`RequestHandlerComponent::`**`prefers`**(*$type = null*)

Determines which content-types the client prefers. If no parameter is given the most likely content type is returned. If $type is an array the first type the client accepts will be returned. Preference is determined primarily by the file extension parsed by Router if one has been provided, and secondly by the list of content-types in `HTTP\_ACCEPT`:

```
$this->RequestHandler->prefers('json');
```

Responding To Requests

`RequestHandlerComponent::`**`renderAs`**(*$controller*, *$type*)

Change the render mode of a controller to the specified type. Will also append the appropriate helper to the controller's helper array if available and not already in the array:

```
// Force the controller to render an xml response.
$this->RequestHandler->renderAs($this, 'xml');
```

This method will also attempt to add a helper that matches your current content type. For example if you render as `rss`, the `RssHelper` will be added.

`RequestHandlerComponent::`**`respondAs`**(*$type*, *$options*)

Sets the response header based on content-type map names. This method lets you set a number of response properties at once:

```
$this->RequestHandler->respondAs('xml', [
    // Force download
    'attachment' => true,
    'charset' => 'UTF-8'
]);
```

[111] http://php.net/callback

`RequestHandlerComponent::responseType()`

Returns the current response type Content-type header or null if one has yet to be set.

Taking Advantage of HTTP Cache Validation

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 this way response times.

Enabling the RequestHandlerComponent in your controller automatically activates a check done before rendering the view. This check compares the response object against the original request to determine whether the response was not modified since the last time the client asked for it.

If response is evaluated as not modified, then the view rendering process is stopped, saving processing time, saving bandwidth and no content is returned to the client. The response status code is then set to `304 Not Modified`.

You can opt-out this automatic checking by setting the `checkHttpCache` setting to `false`:

```
public function initialize()
{
    parent::initialize();
    $this->loadComponent('RequestHandler', [
        'checkHttpCache' => false
    ]);
}
```

Using Custom ViewClasses

When using JsonView/XmlView you might want to override the default serialization with a custom View class, or add View classes for other types.

You can map existing and new types to your custom classes. You can also set this automatically by using the `viewClassMap` setting:

```
public function initialize()
{
    parent::initialize();
    $this->loadComponent('RequestHandler', [
        'viewClassMap' => [
            'json' => 'ApiKit.MyJson',
            'xml' => 'ApiKit.MyXml',
            'csv' => 'ApiKit.Csv'
        ]
    ]);
}
```

Deprecated since version 3.1.0: As of 3.1.0 the `viewClassMap()` method is deprecated. You should use `config()` to change the viewClassMap at runtime.

Configuring Components

Many of the core components require configuration. Some examples of components requiring configuration are *Authentication* and *Cookie*. 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()
    {
        parent::initialize();
        $this->loadComponent('Auth', [
            'authorize' => 'Controller',
            'loginAction' => ['controller' => 'Users', 'action' => 'login']
        ]);
        $this->loadComponent('Cookie', ['expires' => '1 day']);
    }

}
```

You can configure components at runtime using the `config()` method. Often, this is done in your controller's `beforeFilter()` method. The above could also be expressed as:

```
public function beforeFilter(Event $event)
{
    $this->Auth->config('authorize', ['controller']);
    $this->Auth->config('loginAction', ['controller' => 'Users', 'action' => 'login
↪']);

    $this->Cookie->config('name', 'CookieMonster');
}
```

Like helpers, components implement a `config()` method that is used to get and set any configuration data for a component:

```
// Read config data.
$this->Auth->config('loginAction');

// Set config
$this->Csrf->config('cookieName', 'token');
```

As with helpers, components will automatically merge their `$_defaultConfig` property with constructor configuration to create the `$_config` property which is accessible with `config()`.

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->Auth` or another common Component reference with a custom implementation:

```
// src/Controller/PostsController.php
class PostsController extends AppController
{
    public function initialize()
    {
        $this->loadComponent('Auth', [
            'className' => 'MyAuth'
        ]);
    }
}

// src/Controller/Component/MyAuthComponent.php
use Cake\Controller\Component\AuthComponent;
```

```
class MyAuthComponent extends AuthComponent
{
    // Add your code to override the core AuthComponent
}
```

The above would *alias* `MyAuthComponent` to `$this->Auth` 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()
    {
        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.

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->Csrf
public function initialize()
{
    parent::initialize();
    $this->loadComponent('Math');
    $this->loadComponent('Csrf');
}
```

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()
{
    parent::initialize();
    $this->loadComponent('Math', [
        'precision' => 2,
        'randomGenerator' => 'srand'
    ]);
    $this->loadComponent('Csrf');
}
```

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. In this case you can include other components in your component the exact same way you include them in controllers - using the `$components` var:

```
// src/Controller/Component/CustomComponent.php
namespace App\Controller\Component;

use Cake\Controller\Component;

class CustomComponent extends Component
{
    // The other component your component uses
    public $components = ['Existing'];

    // Execute any other additional setup for your component.
    public function initialize(array $config)
    {
        $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()
    {
        // ...
    }
}
```

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->_registry->getController();
```

You can access the controller in any callback method from the event object:

```
$controller = $event->getSubject();
```

Component Callbacks

Components also offer a few request life-cycle callbacks that allow them to augment the request cycle.

beforeFilter(*Event $event*)
Is called before the controller's beforeFilter method, but *after* the controller's initialize() method.

startup(*Event $event*)
Is called after the controller's beforeFilter method but before the controller executes the current action handler.

beforeRender(*Event $event*)
Is called after the controller executes the requested action's logic, but before the controller renders views and layout.

shutdown(*Event $event*)
Is called before output is sent to the browser.

beforeRedirect(*Event $event*, *$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.

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 PDF's 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
namespace App\View;

use Cake\View\View;

class AppView extends View
{
```

```
    public function initialize()
    {
        // Always enable the MyUtils Helper
        $this->loadHelper('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 have a default extension of **.ctp** (CakePHP Template) and utilize the alternative PHP syntax[112] 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 echo $variable; ?>
```

Using Short Tag support:

```
<?= $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 below 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`:

```
<ul>
<?php foreach ($todo as $item): ?>
  <li><?= $item ?></li>
<?php endforeach; ?>
</ul>
```

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: ?>
```

[112] http://php.net/manual/en/control-structures.alternative-syntax.php

```
    <h3>Hi unknown user</h3>
<?php endif; ?>
```

If you'd prefer using a templating language like Twig[113], a subclass of View will bridge your templating language and CakePHP.

Template files are stored in **src/Template/**, 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 **src/Template/Products/view.ctp**.

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:

- **views**: 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 a *Using 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); ?>
```

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'.

[113] http://twig.sensiolabs.org

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:

```
<!-- src/Template/Common/view.ctp -->
<h1><?= $this->fetch('title') ?></h1>
<?= $this->fetch('content') ?>

<div class="actions">
    <h3>Related actions</h3>
    <ul>
    <?= $this->fetch('sidebar') ?>
    </ul>
</div>
```

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:

```
<!-- src/Template/Posts/view.ctp -->
<?php
$this->extend('/Common/view');

$this->assign('title', $post);

$this->start('sidebar');
?>
<li>
<?php
echo $this->Html->link('edit', [
    'action' => 'edit',
    $post->id
]); ?>
</li>
<?php $this->end(); ?>

// The remaining content will be available as the 'content' block
// In the parent view.
<?= h($post->body) ?>
```

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.ctp** 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.

You can get the list of all populated blocks using the `blocks()` method:

```
$list = $this->blocks();
```

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');
$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', '');
```

New in version 3.2: View::reset() was added in 3.2

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');
```

Note: You should avoid using `content` as a block name. This is used by CakePHP internally for extended views, and view content in the layout.

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 src/Template/Layout/default.ctp
<?php if ($this->fetch('menu')): ?>
<div class="menu">
    <h3>Menu options</h3>
    <?= $this->fetch('menu') ?>
</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><?= $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 **src/Template/Layout/default.ctp**. 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 **src/Template/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');
```

You can create as many layouts as you wish: just place them in the **src/Template/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');

    // Before 3.4
    $this->viewBuilder()->layout('admin');

    // Before 3.1
    $this->layout = 'admin';
}

// 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');

        // or the following before 3.4
        $this->viewBuilder()->layout('default_small_ad');

        // or the following before 3.1
        $this->layout = '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 view_active()
    {
        $this->viewBuilder()->layout('Contacts.contact');
        // or the following before 3.1
        $this->layout = '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 **src/Template/Element/** folder, and have the .ctp 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 **src/Template/Element/helpbox.ctp** file can use the `$helptext` variable:

```
// Inside src/Template/Element/helpbox.ctp
echo $helptext; // Outputs "Oh, this text is very helpful."
```

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 (e.g., **plugins/Contacts/Template/Element/sidebar/helpbox.ctp**), use the following:

```
echo $this->element('Contacts.sidebar/helpbox');
```

Requesting Elements from the App

If you are within a plugin's template file and want to render an element residing in your main application rather than this or another plugin, use the following:

```
echo $this->element('some_global_element', [], ['plugin' => false]);
// or...
echo $this->element('some_global_element', ['localVar' => $someData], ['plugin' =>␣
↪false]);
```

Routing prefix and Elements

New in version 3.0.1.

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 **src/Template/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()->className('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[114], 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.

Creating a Cell

To create a cell, define a class in **src/View/Cell** and a template in **src/Template/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 **src/Template/Cell/Inbox/display.ctp**. 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 typed out.

[114] https://github.com/apotonick/cells

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()
    {
        $this->loadModel('Messages');
        $unread = $this->Messages->find('unread');
        $this->set('unread_count', $unread->count());
    }

}
```

Because Cells use the `ModelAwareTrait` and `ViewVarsTrait`, they behave very much like a controller would. We can use the `loadModel()` and `set()` methods just like we would in a controller. In our template file, add the following:

```
<!-- src/Template/Cell/Inbox/display.ctp -->
<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, e.g. **display.ctp**.

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 file-name and line number for any fatal errors raised. To obtain a meanful 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->template = '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 the `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\Paginator;

class FavoritesCell extends Cell
{
    public function display($user)
    {
        $this->loadModel('Messages');

        // Create a paginator
        $paginator = new Paginator();

        // Paginate the model
        $results = $paginator->paginate(
            $this->Messages,
            $this->request->getQueryParams(),
            [
                // Use a parameterized custom finder.
                'finder' => ['favorites' => [$user]],

                // Use scoped query string parameters.
```

```
                'scope' => 'favorites',
            ]
        );
        $this->set('favorites', $results);
    }
}
```

The above cell would paginate the `Messages` model using *scoped pagination parameters*.

New in version 3.5.0: `Cake\Datasource\Paginator` was added in 3.5.0.

Themes

You can take advantage of themes, making it easy to switch the look and feel of your page quickly. Themes in CakePHP are simply plugins that focus on providing template files. 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*.

To use themes, set the theme name in your controller's action or `beforeRender()` callback:

```
class ExamplesController extends AppController
{
    // For CakePHP before 3.1
    public $theme = 'Modern';

    public function beforeRender(\Cake\Event\Event $event)
    {
        $this->viewBuilder()->theme('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/src/Template**. It's important to remember that CakePHP expects CamelCase plugin/theme names. Beyond that, the folder structure within the **plugins/Modern/src/Template** folder is exactly the same as **src/Template/**.

For example, the view file for an edit action of a Posts controller would reside at **plugins/Modern/src/Template/Posts/edit.ctp**. Layout files would reside in **plugins/Modern/src/Template/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/src/Template/Plugin/Cms/Tags/edit.ctp** 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 **src/Template/** 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 easy packaging and distribution of themes. Whilst in development, requests for theme assets will be handled by `Cake\Routing\Dispatcher`. 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` let you create JSON and XML responses, and integrate with the `Cake\Controller\Component\RequestHandlerComponent`.

By enabling `RequestHandlerComponent` in your application, and enabling support for the `json` and or `xml` extensions, you can automatically leverage the new view classes. `JsonView` and `XmlView` will be referred to as data views for the rest of this page.

There are two ways you can generate data views. The first is by using the `_serialize` key, and the second is by creating normal template files.

Enabling Data Views in Your Application

Before you can use the data view classes, you'll first need to load the `Cake\Controller\Component\RequestHandlerComponent` in your controller:

```
public function initialize()
{
    ...
    $this->loadComponent('RequestHandler');
}
```

This can be done in your *AppController* and will enable automatic view class switching on content types. You can also set the component up with the `viewClassMap` setting, to map types to your custom classes and/or map other data types.

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` key is a special view variable that indicates which other 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;

class ArticlesController extends AppController
{
    public function initialize()
    {
        parent::initialize();
        $this->loadComponent('RequestHandler');
    }

    public function index()
    {
        // Set the view vars that have to be serialized.
        $this->set('articles', $this->paginate());
        // Specify which view vars JsonView should serialize.
        $this->set('_serialize', ['articles']);
    }
}
```

You can also define `_serialize` as an array of view variables to combine:

```
namespace App\Controller;

class ArticlesController extends AppController
{
    public function initialize()
    {
        parent::initialize();
        $this->loadComponent('RequestHandler');
    }

    public function index()
    {
        // Some code that created $articles and $comments

        // Set the view vars that have to be serialized.
        $this->set(compact('articles', 'comments'));

        // Specify which view vars JsonView should serialize.
        $this->set('_serialize', ['articles', 'comments']);
    }
}
```

Defining `_serialize` as an array has the added 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.

New in version 3.1.0: In this version, the variable `_serialize` is automaticaly set to `true` to serialize all view variables instead of explicitly specifying them.

Using a Data View with Template Files

You should use template files if you need to do some manipulation of your view content before creating the final output. For example if we had articles, that had 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 - src/Template/Articles/json/index.ctp
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.

Note: As of 3.1.0 AppController, in the application skeleton automatically adds `'_serialize' => true` to all XML/JSON requests. You will need to remove this code from the beforeRender callback or set `'_serialize' => false` in your controller's action if you want to use view files.

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` view variable.

The XmlView class supports the `_xmlOptions` variable that allows you to customize the options used to generate XML, e.g. `tags` vs `attributes`.

Creating JSON Views

class `JsonView`

The JsonView class supports the `_jsonOptions` variable that allows you to customize the bit-mask used to generate JSON. See the json_encode[115] 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->set('_jsonOptions', JSON_FORCE_OBJECT);
$this->set('_serialize', ['errors']);
```

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

[115] http://php.net/json_encode

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`.

Example Usage

While the *RequestHandlerComponent* can automatically set the view based on the request content-type or extension, you could also handle view mappings in your controller:

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

use App\Controller\AppController;
use Cake\Network\Exception\NotFoundException;

class VideosController extends AppController
{
    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
        $this->viewBuilder()->className($formats[$format]);

        // Set Force Download
        $this->response->download('report-' . date('YmdHis') . '.' . $format);

        // Get data
        $videos = $this->Videos->find('latest');

        // Set Data View
        $this->set(compact('videos'));
        $this->set('_serialize', ['videos']);
    }
}
```

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 = []*)

New in version 3.3.6.

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 informations.

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']
);

// 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}}>{{custom}}{{separator}}</span></
↪li>'
]
```

You can easily customize them using the `templates()` method from the `StringTemplateTrait`:

```
$this->Breadcrumbs->templates([
    '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->templates([
    '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);
```

New in version 3.4.0: The `reset()` method was added in 3.4.0

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 **src/Template/Element/Flash** directory. You'll notice that CakePHP's App template comes with three flash elements: **success.ctp**, **default.ctp**, and **error.ctp**.

Rendering Flash Messages

To render a flash message, you can simply use FlashHelper's `render()` method:

```
<?= $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 View: Will use great_success.ctp instead of succcess.ctp
<?= $this->Flash->render('flash', [
    'element' => 'great_success'
]);
```

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.

New in version 3.1: The *FlashComponent* now stacks messages. If you set multiple flash messages, when you call `render()`, each message will be rendered in its own elements, in the order they were set.

For more information about the available array options, please refer to the *FlashComponent* section.

Routing Prefix and Flash Messages

New in version 3.0.1.

If you have a Routing prefix configured, you can now have your Flash elements stored in **src/Template/{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 **src/Template/Element/Flash/error.ctp** 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 = []*)

- `$context` - The context for which the form is being defined. Can be an ORM entity, ORM resultset, array of metadata or `false/null` (to make a model-less form).
- `$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[116]; this context class allows FormHelper to work with results from the built-in ORM.
- An array containing the `'schema'` key, will map to ArrayContext[117] which allows you to create simple data structures to build forms against.
- `null` and `false` will map to NullContext[118]; 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.

[116] https://api.cakephp.org/3.x/class-Cake.View.Form.EntityContext.html
[117] https://api.cakephp.org/3.x/class-Cake.View.Form.ArrayContext.html
[118] https://api.cakephp.org/3.x/class-Cake.View.Form.NullContext.html

All context classes also have access to the request data, making it simpler to build forms.

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.ctp:
// 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.

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.

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 the query string

New in version 3.4.0.

A FormHelper's values sources define where its rendered elements, such as input-tags, receive their values from.

By default FormHelper draws its values from the 'context'. The default contexts, such as `EntityContext`, will fetch data from the current entity, or from `$request->getData()`.

If however, you are building a form that needs to read from the query string, you can use `valueSource()` to change where `FormHelper` reads data input data from:

```
// Prioritize query string over context:
echo $this->Form->create($article, [
    'valueSources' => ['query', 'context']
]);

// Same effect:
echo $this->Form
    ->setValueSources(['query', 'context'])
    ->create($articles);

// Only read data from the query string
echo $this->Form->create($article);
$this->Form->setValueSources('query');

// Same effect:
echo $this->Form->create($article, ['valueSources' => 'query']);
```

The supported sources are `context`, `data` and `query`. You can use one or more sources. Any widgets generated by `FormHelper` will gather their values from the sources, in the order you setup.

The value sources will be reset to the default (`['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.

E.g.

```
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 `login()` action of the current controller, you would supply an `$options` array, like the following:

```
echo $this->Form->create($article, ['url' => ['action' => 'login']]);
```

Output:

```
<form method="post" action="/users/login">
```

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' => 'http://www.google.com/search',
    'type' => 'get'
]);
```

Output:

```
<form method="get" action="http://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 validation sets, and you will want FormHelper to mark fields required based on a the specific validation rules 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 the 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.

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($request, $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 = []*)

[119] https://api.cakephp.org/3.x/class-Cake.View.Form.ContextInterface.html

- `$fieldName` - A field name in the form `'Modelname.fieldname'`.
- `$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 to 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 (e.g. `select`, `button`, `textarea`).

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}}/>'
```

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 day, month, and year selects

datetime, timestamp day, month, year, hour, minute, and meridian selects

time hour, minute, and meridian selects

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 *View\Helper\FormHelper::submit()* or set `'novalidate' => true` in options for *View\Helper\FormHelper::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: Day, Month, Year, Hour, Minute
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_dt', [
    'label' => 'Date of birth',
    'minYear' => date('Y') - 70,
    'maxYear' => 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'));
```

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'));
```

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'));
```

Note: You should not use `FormHelper::control()` to generate submit buttons. Use *`View\Helper\FormHelper::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 makes it easy to save data 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`.

  E.g.

  ```
  echo $this->Form->control('field', ['type' => 'file']);
  echo $this->Form->control('email', ['type' => 'email']);
  ```

  Output:

[120] https://en.wikipedia.org/wiki/Camel_case#Variations_and_synonyms

```
<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`.

  E.g.

```
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.

  E.g.

```
echo $this->Form->control('name', ['label' => false]);
```

  Output:

```
<div class="input">
    <input name="name" type="text" value="" id="name" />
</div>
```

  Set this 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.

  E.g.

```
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`.

  E.g.

  ```
  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.

  E.g.

  ```
  $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.
- `$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.

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 *`View\Helper\FormHelper::label()`* and *`View\Helper\FormHelper::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). An explicit default value will override any default values defined in the schema.

  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.

Changed in version 3.3.0: As of 3.3.0, FormHelper will automatically use any default values defined in your database schema. You can disable this behavior by setting the `schemaDefault` option to `false`.

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.

E.g.

```
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.

E.g.

```
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.

E.g.

```
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`.

  E.g.

  ```
  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.

  E.g.

  ```
  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', [
    '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
</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',
]);
```

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.

E.g.

```
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.

E.g.

```
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'`.

E.g.

```
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. 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.

E.g.

```
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>
```

Creating Select Pickers

`Cake\View\Helper\FormHelper::select` (*string $fieldName*, *array $options*, *array $attributes*)

- `$fieldName` - A field name in the form `'Modelname.fieldname'`. This will provide the `name` attribute of the `select` element.
- `$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.

E.g.

```
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.

E.g.

```
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.

  E.g.

  ```
  // 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.

  E.g.

  ```
  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">
        <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.

  E.g.

```
$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>
</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.

Upon submission, file fields provide an expanded data array to the script receiving the form data.

For the example above, the values in the submitted data array would be organized as follows, if CakePHP was installed on a Windows server (the key `'tmp\_name'` will contain a different path in a Unix environment):

```
$this->request->data['submittedfile']

// would contain the following array:
[
    'name' => 'conference_schedule.pdf',
```

```
    'type' => 'application/pdf',
    'tmp_name' => 'C:/WINDOWS/TEMP/php1EE.tmp',
    'error' => 0, // On Windows this can be a string.
    'size' => 41737,
];
```

This array is generated by PHP itself, so for more detail on the way PHP handles data passed via file fields read the PHP manual section on file uploads[121].

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

The date and time related methods share a number of common traits and options and hence are grouped together into this subsection.

Common Options for Date & Time Controls

These options are common for the date and time related controls:

- `'empty'` - If `true` an extra, empty, `option` HTML element is added inside `select` at the top of the list. If a string, that string is displayed as the empty element. Defaults to `true`.
- `'default'` | `value` - Use either of the two to set the default value to be shown by the field. A value in `$this->request->getData()` matching the field name will override this value. If no default is provided `time()` will be used.
- `'year','month','day','hour','minute','second','meridian'` - These options allow you to control which control elements are generated or not. By setting any of these options to `false` you can disable the generation of that specific that select picker (if by default it would be rendered in the used method). In addition each option allows you to pass HTML attributes to that specific `select` element.

Options for Date-Related Controls

These options are concerning the date-related methods - i.e. `year()`, `month()`, `day()`, `dateTime()` and `date()`:

- `'monthNames'` - If `false`, 2 digit numbers will be used instead of text for displaying months in the select picker. If set to an array (e.g. `['01' => 'Jan','02' => 'Feb',...]`), the given array will be used.
- `'minYear'` - The lowest value to use in the year select picker.
- `'maxYear'` - The maximum value to use in the year select picker.
- `'orderYear'` - The order of year values in the year select picker. Possible values are `'asc'` and `'desc'`. Defaults to `'desc'`.

[121] http://php.net/features.file-upload

Options for Time-Related Controls

These options are concerning the time-related methods - `hour()`, `minute()`, `second()`, `dateTime()` and `time()`:

- `'interval'` - The interval in minutes between the values which are displayed in the `option` elements of the minutes select picker. Defaults to 1.
- `'round'` - Set to `up` or `down` if you want to force rounding minutes in either direction when the value doesn't fit neatly into an interval. Defaults to `null`.
- `timeFormat` - Applies to `dateTime()` and `time()`. The time format to use in the select picker; either `12` or `24`. When this option is set to anything else than `24` the format will be automatically set to `12` and the `meridian` select picker will be displayed automatically to the right of the seconds select picker. Defaults to 24.
- `format` - Applies to `hour()`. The time format to use; either `12` or `24`. In case it's set to `12` the `meridian` select picker won't be automatically displayed. It's up to you to either add it or provide means to infer from the form context the right period of the day. Defaults to 24.
- `second` - Applies to `dateTime()` and `time()`. Set to `true` to enable the seconds drop down. Defaults to `false`.

Creating DateTime 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*, or specific datetime options (see above), as well as any valid HTML attributes.

Creates a set of `select` elements for date and time.

To control the order of controls, and any elements/content between the controls you can override the `dateWidget` template. By default the `dateWidget` template is:

```
{{year}}{{month}}{{day}}{{hour}}{{minute}}{{second}}{{meridian}}
```

Calling the method without additional options will generate, by default, 5 select pickers, for: year (4 digits), month (full English name), day (num), hour (num), minutes (num).

For example

```
<?= $this->form->dateTime('registered') ?>
```

Output:

```
<select name="registered[year]">
    <option value="" selected="selected"></option>
    <option value="2022">2022</option>
    ...
    <option value="2012">2012</option>
</select>
<select name="registered[month]">
    <option value="" selected="selected"></option>
    <option value="01">January</option>
    ...
    <option value="12">December</option>
```

```
</select>
<select name="registered[day]">
    <option value="" selected="selected"></option>
    <option value="01">1</option>
    ...
    <option value="31">31</option>
</select>
<select name="registered[hour]">
    <option value="" selected="selected"></option>
    <option value="00">0</option>
    ...
    <option value="23">23</option>
</select>
<select name="registered[minute]">
    <option value="" selected="selected"></option>
    <option value="00">00</option>
    ...
    <option value="59">59</option>
</select>
```

To create datetime controls with custom classes/attributes on a specific select box, you can provide them as arrays of options for each component, within the `$options` argument.

For example:

```
echo $this->Form->dateTime('released', [
    'year' => [
        'class' => 'year-classname',
    ],
    'month' => [
        'class' => 'month-class',
        'data-type' => 'month',
    ],
]);
```

Which would create the following two select pickers:

```
<select name="released[year]" class="year-class">
    <option value="" selected="selected"></option>
    <option value="00">0</option>
    <option value="01">1</option>
    <!-- .. snipped for brevity .. -->
</select>
<select name="released[month]" class="month-class" data-type="month">
    <option value="" selected="selected"></option>
    <option value="01">January</option>
    <!-- .. snipped for brevity .. -->
</select>
```

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*, of the *Common Options for Date & Time Controls*, any applicable *Options for Time-Related Controls*, as well as any valid HTML attributes.

Creates, by default, three select pickers populated with values for: year (4 digits), month (full English name) and day (numeric), respectively.

You can further control the generated `select` elements by providing additional options.

For example:

```
// Assuming current year is 2017; this disables day picker, removes empty
// option on year picker, limits lowest year, adds HTML attributes on year,
// adds a string 'empty' option on month, changes month to numeric
<?php
    echo $this->Form->date('registered', [
        'minYear' => 2018,
        'monthNames' => false,
        'empty' => [
            'year' => false,
            'month' => 'Choose month...'
        ],
        'day' => false,
        'year' => [
            'class' => 'cool-years',
            'title' => 'Registration Year'
        ]
    ]);
?>
```

Output:

```
<select class= "cool-years" name="registered[year]" title="Registration Year">
    <option value="2022">2022</option>
    <option value="2021">2021</option>
    ...
    <option value="2018">2018</option>
</select>
<select name="registered[month]">
    <option value="" selected="selected">Choose month...</option>
    <option value="01">1</option>
    ...
    <option value="12">12</option>
</select>
```

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*, of the *Common Options for Date & Time Controls*, any applicable *Options for Time-Related Controls*, as well as any valid HTML attributes.

Creates, by default, two `select` elements (`hour` and `minute`) populated with values for 24 hours and 60 minutes, respectively. Additionally, HTML attributes may be supplied in `$options` for each specific component. If

`$options['empty']` is `false`, the select picker will not include an empty default option.

For example, to create a time range with minutes selectable in 15 minute increments, and to apply classes to the select boxes, you could do the following:

```
echo $this->Form->time('released', [
    'interval' => 15,
    'hour' => [
        'class' => 'foo-class',
    ],
    'minute' => [
        'class' => 'bar-class',
    ],
]);
```

Which would create the following two select pickers:

```
<select name="released[hour]" class="foo-class">
    <option value="" selected="selected"></option>
    <option value="00">0</option>
    <option value="01">1</option>
    <!-- .. snipped for brevity .. -->
    <option value="22">22</option>
    <option value="23">23</option>
</select>
<select name="released[minute]" class="bar-class">
    <option value="" selected="selected"></option>
    <option value="00">00</option>
    <option value="15">15</option>
    <option value="30">30</option>
    <option value="45">45</option>
</select>
```

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*, of the *Common Options for Date & Time Controls*, any applicable *Options for Date-Related Controls*, as well as any valid HTML attributes.

Creates a `select` element populated with the years from `minYear` to `maxYear` (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', [
    'minYear' => 2000,
    'maxYear' => date('Y')
]);
```

If it was 2009, you would get the following:

```
<select name="purchased[year]">
    <option value=""></option>
    <option value="2009">2009</option>
    <option value="2008">2008</option>
    <option value="2007">2007</option>
    <option value="2006">2006</option>
    <option value="2005">2005</option>
    <option value="2004">2004</option>
    <option value="2003">2003</option>
    <option value="2002">2002</option>
    <option value="2001">2001</option>
    <option value="2000">2000</option>
</select>
```

Creating Month Controls

`Cake\View\Helper\FormHelper::month` (*string $fieldName*, *array $attributes*)

- `$fieldName` - A field name that will be used as a prefix for the HTML `name` attribute of the `select` element.
- `$attributes` - An optional array including any of the *Common Options For Specific Controls*, of the *Common Options for Date & Time Controls*, any applicable *Options for Date-Related Controls*, as well as any valid HTML attributes.

Creates a `select` element populated with month names.

For example:

```
echo $this->Form->month('mob');
```

Will output:

```
<select name="mob[month]">
    <option value=""></option>
    <option value="01">January</option>
    <option value="02">February</option>
    <option value="03">March</option>
    <option value="04">April</option>
    <option value="05">May</option>
    <option value="06">June</option>
    <option value="07">July</option>
    <option value="08">August</option>
    <option value="09">September</option>
    <option value="10">October</option>
    <option value="11">November</option>
    <option value="12">December</option>
</select>
```

You can pass in, your own array of months to be used by setting the `'monthNames'` attribute, or have months displayed as numbers by passing `false`.

E.g.

```
echo $this->Form->month('mob', ['monthNames' => false]);
```

Note: The default months can be localized with CakePHP *Internationalization & Localization* features.

Creating Day Controls

`Cake\View\Helper\FormHelper::day` (*string $fieldName*, *array $attributes*)

- `$fieldName` - A field name that will be used as a prefix for the HTML `name` attribute of the `select` element.
- `$attributes` - An optional array including any of the *Common Options For Specific Controls*, of the *Common Options for Date & Time Controls*, any applicable *Options for Date-Related Controls*, as well as any valid HTML attributes.

Creates a `select` element populated with the (numerical) days of the month.

To create an empty `option` element with a prompt text of your choosing (e.g. the first option is 'Day'), you can supply the text in the `'empty'` parameter.

For example:

```
echo $this->Form->day('created', ['empty' => 'Day']);
```

Will output:

```
<select name="created[day]">
    <option value="" selected="selected">Day</option>
    <option value="01">1</option>
    <option value="02">2</option>
    <option value="03">3</option>
    ...
    <option value="31">31</option>
</select>
```

Creating Hour Controls

`Cake\View\Helper\FormHelper::hour` (*string $fieldName*, *array $attributes*)

- `$fieldName` - A field name that will be used as a prefix for the HTML `name` attribute of the `select` element.
- `$attributes` - An optional array including any of the *Common Options For Specific Controls*, of the *Common Options for Date & Time Controls*, any applicable *Options for Time-Related Controls*, as well as any valid HTML attributes.

Creates a `select` element populated with the hours of the day.

You can create either 12 or 24 hour pickers using the `'format'` option:

```
echo $this->Form->hour('created', [
    'format' => 12
]);
echo $this->Form->hour('created', [
    'format' => 24
]);
```

Creating Minute Controls

`Cake\View\Helper\FormHelper::minute` (*string $fieldName*, *array $attributes*)

- `$fieldName` - A field name that will be used as a prefix for the HTML `name` attribute of the `select` element.

- `$attributes` - An optional array including any of the *Common Options For Specific Controls*, of the *Common Options for Date & Time Controls*, any applicable *Options for Time-Related Controls*, as well as any valid HTML attributes.

Creates a `select` element populated with values for the minutes of the hour. You can create a select picker that only contains specific values by using the `'interval'` option.

For example, if you wanted 10 minutes increments you would do the following:

```
// In your view template file
echo $this->Form->minute('arrival', [
    'interval' => 10
]);
```

This would output:

```
<select name="arrival[minute]">
    <option value="" selected="selected"></option>
    <option value="00">00</option>
    <option value="10">10</option>
    <option value="20">20</option>
    <option value="30">30</option>
    <option value="40">40</option>
    <option value="50">50</option>
</select>
```

Creating Meridian Controls

`Cake\View\Helper\FormHelper::meridian`(*string $fieldName*, *array $attributes*)

- `$fieldName` - A field name that will be used as a prefix for the HTML `name` attribute of the `select` element.
- `$attributes` - An optional array including any of the *Common Options For Specific Controls* as well as any valid HTML attributes.

Creates a `select` element populated with 'am' and 'pm'. This is useful when the hour format is set to `12` instead of `24`, as it allows to specify the period of the day to which the hour belongs.

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. String or an 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.

E.g.

```
echo $this->Form->label('User.name');
echo $this->Form->label('User.name', 'Your username');
```

Output:

```
<label for="user-name">Name</label>
<label for="user-name">Your username</label>
```

When you set `$options` to a string it will be used as a class name:

```
echo $this->Form->label('User.name', null, ['id' => 'user-label']);
echo $this->Form->label('User.name', 'Your username', 'highlight');
```

Output:

```
<label for="user-name" id="user-label">Name</label>
<label for="user-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
        ->requirePresence('ticket', 'create')
        ->notEmpty('ticket');
}

// And inside Templates/Tickets/add.ctp 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 *View\Helper\FormHelper::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');
}
```

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

- `$options['type']` - You can set this to one of the following three possible values:
  1. `'submit'` - Similarly to the `$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.
- `$options['escape']` - Boolean. If set to `true` it will HTML encode the value provided inside `$title`. Defaults to `false`.

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 of use of the `'escape'` option:

```
// Will render escaped HTML.
echo $this->Form->button('<em>Submit Form</em>', [
    'type' => 'submit',
    'escape' => true
]);
```

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.

E.g.

```
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.ctp
<?= $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.

Options for POST Link

- `'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'`.
- `'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);
// Prior to 3.4
$this->Form->templates($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 at the FormHelper API documentation[122].

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('User.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>'
]);
```

[122] https://api.cakephp.org/3.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.

E.g.

```
// 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>
```

New in version 3.1: The templateVars option was added in 3.1.0

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' => '{{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 $fields*, *$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]);
// Or prior to 3.4.0:
echo $this->Form->allInputs(['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)
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');

// Multiple select element for belongsToMany
echo $this->Form->control('tags._ids', [
    'type' => 'select',
    'multiple' => true,
    'options' => $tagList,
]);

// 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' => [
        'Authors',
        'Authors.Profiles',
```

```
        'Tags',
        'Comments'
    ]
]);
```

Adding Custom Widgets

CakePHP makes it easy to add custom control widgets in your application, 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\Widget\WidgetInterface;

class AutocompleteWidget implements WidgetInterface
{

    protected $_templates;

    public function __construct($templates)
    {
        $this->_templates = $templates;
    }

    public function render(array $data, ContextInterface $context)
    {
        $data += [
            'name' => '',
        ];
        return $this->_templates->format('autocomplete', [
            'name' => $data['name'],
            'attrs' => $this->_templates->formatAttributes($data, ['name'])
        ]);
    }

    public function secureFields(array $data)
    {
        return [$data['name']];
    }
}
```

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']
    ]
]);
```

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->widgetRegistry()->get('text'),
    $this->Form->widgetRegistry()->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 *`View\Helper\FormHelper::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)
<link
    href="http://example.com/favicon.ico"
    title="favicon.ico" type="image/x-icon"
    rel="alternate"
/>
<?= $this->Html->meta(
    'Comments',
    '/comments/index.rss',
    ['type' => 'rss']
);
?>
```

```
// 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'
]);
?>
// 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.

Creating DOCTYPE

Cake\View\Helper\HtmlHelper::**docType**(*string $type = 'html5'*)

Returns a (X)HTML DOCTYPE (document type declaration). Supply the document type according to the following table:

type	translated value
html4-strict	HTML 4.01 Strict
html4-trans	HTML 4.01 Transitional
html4-frame	HTML 4.01 Frameset
html5 (default)	HTML5
xhtml-strict	XHTML 1.0 Strict
xhtml-trans	XHTML 1.0 Transitional
xhtml-frame	XHTML 1.0 Frameset
xhtml11	XHTML 1.1

```
echo $this->Html->docType();
// Outputs: <!DOCTYPE html>

echo $this->Html->docType('html4-trans');
// Outputs:
// <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
//    "http://www.w3.org/TR/html4/loose.dtd">
```

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]);
```

Creating Links

`Cake\View\Helper\HtmlHelper::link` (*string $title*, *mixed $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',
    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.

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('http://code.jquery.com/jquery.min.js');
```

Will output:

```
<script src="http://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:

```
echo $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>
    <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']],
    ['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>
    <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**(*$templates*)

The `$templates` parameter can be either a string file path to the PHP file containing the tags you want to load, or an array of templates to add/replace:

```
// Load templates from config/my_html.php
$this->Html->setTemplates('my_html');

// Load specific templates.
```

```
$this->Html->setTemplates([
    'javascriptlink' => '<script src="{{url}}" type="text/javascript"{{attrs}}></
↪script>'
]);
```

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: ' {{content}} '

Creating Breadcrumb Trails with HtmlHelper

`Cake\View\Helper\HtmlHelper::addCrumb(string $name, string $link = null, mixed $options = null)`

`Cake\View\Helper\HtmlHelper::getCrumbs(string $separator = '»', string $startText = false)`

`Cake\View\Helper\HtmlHelper::getCrumbList(array $options = [], $startText = false)`

Many applications have breadcrumb trails to ease end user navigations. You can create a breadcrumb trail in your app with some help from HtmlHelper. To make bread crumbs, first the following in your layout template:

```
echo $this->Html->getCrumbs(' > ', 'Home');
```

The `$startText` option can also accept an array. This gives more control over the generated first link:

```
echo $this->Html->getCrumbs(' > ', [
    'text' => $this->Html->image('home.png'),
    'url' => ['controller' => 'Pages', 'action' => 'display', 'home'],
    'escape' => false
]);
```

Any keys that are not `text` or `url` will be passed to *`link()`* as the `$options` parameter.

Now, in your view you'll want to add the following to start the breadcrumb trails on each of the pages:

```
$this->Html->addCrumb('Users', '/users');
$this->Html->addCrumb('Add User', ['controller' => 'Users', 'action' => 'add']);
```

This will add the output of "**Home > Users > Add User**" in your layout where `getCrumbs` was added.

You can also fetch the crumbs formatted inside an HTML list:

```
echo $this->Html->getCrumbList();
```

As options you can use regular HTML parameter that fits in the `<ul>` (Unordered List) such as `class` and for the specific options, you have: `separator` (will be between the `<li>` elements), `firstClass` and `lastClass` like:

```
echo $this->Html->getCrumbList(
    [
        'firstClass' => false,
        'lastClass' => 'active',
        'class' => 'breadcrumb'
    ],
    'Home'
);
```

This method uses `Cake\View\Helper\HtmlHelper::tag()` to generate list and its elements. Works similar to *View\Helper\HtmlHelper::getCrumbs()*, so it uses options which every crumb was added with. You can use the `$startText` parameter to provide the first breadcrumb link/text. This is useful when you always want to include a root link. This option works the same as the `$startText` option for *View\Helper\HtmlHelper::getCrumbs()*.

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). 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()`*

Setting the Default Currency

`Cake\View\Helper\NumberHelper::`**`defaultCurrency`**(*$currency*)

Setter/getter 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 `false`, it will clear the currently stored value. By default, it will retrieve the `intl.default_locale` 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', [
    '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:

```
// 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. It works in tandem with `PaginatorComponent`.

See also *Pagination* for information on how to create paginated datasets and do paginated queries.

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()
{
    ...
    $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()
{
    ...
    $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');
// Prior to 3.4
$result = $this->Paginator->templates('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'.

Accepted keys for `$options`:

- `escape` Whether you want the contents HTML entity encoded, defaults to `true`.
- `model` The model to use, defaults to `PaginatorHelper::defaultModel()`.
- `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.

Supported options are:

- `before` Content to be inserted before the numbers.
- `after` Content to be inserted after the numbers.
- `model` Model to create numbers for, defaults to `PaginatorHelper::defaultModel()`.
- `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 params.

```
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.

`$options` supports the following keys:

- `escape` Whether you want the contents HTML entity encoded, defaults to `true`.
- `model` The model to use, defaults to `PaginatorHelper::defaultModel()`.
- `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"><span>&lt;&lt; previous</span></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`

`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.

The options parameter accepts the following:

- `model` The model to use defaults to `PaginatorHelper::defaultModel()`
- `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.

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]);
```

New in version 3.4.0: The `first` and `last` options were added in 3.4.0

Checking the Pagination State

`Cake\View\Helper\PaginatorHelper::current`(*string $model = null*)
Gets the current page of the recordset for the given model:

```
// Our URL is: http://example.com/comments/view/page:3
echo $this->Paginator->current('Comment');
// Output is 3
```

`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` (*string $model = null*)
Returns `true` if the given result set is not at the first page.

`Cake\View\Helper\PaginatorHelper::hasPage` (*string $model = null*, *integer $page = 1*)
Returns `true` if the given result set has the page number given by `$page`.

`Cake\View\Helper\PaginatorHelper::total` (*string $model = null*)
Returns the total number of pages for the provided model.

New in version 3.4.0.

Creating a Page Counter

`Cake\View\Helper\PaginatorHelper::counter` (*$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.

There are a number of options for `counter()`. The supported ones are:

- `format` Format of the counter. 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([
    'format' => 'range'
]);
```

- `model` The name of the model being paginated, defaults to `PaginatorHelper::defaultModel()`. This is used in conjunction with the custom string on 'format' option.

Generating Pagination URLs

`Cake\View\Helper\PaginatorHelper::generateUrl(array $options = [], $model = null, $full = false)`

By default returns a full pagination URL string for use in non-standard contexts (i.e. JavaScript).

```
echo $this->Paginator->generateUrl(['sort' => 'title']);
```

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.

New in version 3.5.0: The `limitControl()` method was added in 3.5.0

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. 'url' has a few sub options as well:
  - `sort` The key that the records are sorted by.
  - `direction` The direction of the sorting. Defaults to 'ASC'.
  - `page` The page number to display.

  The above mentioned options can be used to force particular pages/directions. You can also append additional URL content into all URLs generated in the helper:

  ```
  $this->Paginator->options([
      'url' => [
          'sort' => 'email',
          'direction' => 'desc',
          'page' => 6,
          'lang' => 'en'
      ]
  ]);
  ```

  The 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`.
- `model` The name of the model being paginated, defaults to `PaginatorHelper::defaultModel()`.

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[123] in the API. As mentioned, the PaginatorHelper also offers sorting features which can be integrated into your table column headers:

```
<!-- src/Template/Posts/index.ctp -->
<table>
    <tr>
        <th><?= $this->Paginator->sort('id', 'ID') ?></th>
        <th><?= $this->Paginator->sort('title', 'Title') ?></th>
    </tr>
       <?php foreach ($recipes as $recipe): ?>
    <tr>
        <td><?= $recipe->id ?> </td>
        <td><?= h($recipe->title) ?> </td>
    </tr>
    <?php endforeach; ?>
</table>
```

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>
```

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 »') ?>
```

[123] https://api.cakephp.org/3.x/class-Cake.View.Helper.PaginatorHelper.html

```
// 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 set the `model` option when generating pagination related elements. You can either use the `model` option on every method call you make to `PaginatorHelper`, or use `options()` to set the default model:

```
// Pass the model option
echo $this->Paginator->sort('title', ['model' => 'Articles']);

// Set the default model.
$this->Paginator->options(['defaultModel' => 'Articles']);
echo $this->Paginator->sort('title');
```

By using the `model` option, `PaginatorHelper` will automatically use the `scope` defined in when the query was paginated.

New in version 3.3.0: Multiple Pagination was added in 3.3.0

Rss

class `Cake\View\Helper\`**`RssHelper`**(*View $view*, *array $config = []*)

The RssHelper makes generating XML for RSS feeds[124] easy.

Deprecated since version 3.5.0: RssHelper is deprecated as of 3.5.0, and will be removed in 4.0.0

Creating an RSS Feed with the RssHelper

This example assumes you have a Articles Controller, Articles Table and an Article Entity already created and want to make an alternative view for RSS.

Creating an XML/RSS version of `articles/index` is a snap with CakePHP. After a few simple steps you can simply append the desired extension .rss to `articles/index` making your URL `articles/index.rss`. Before we jump too far ahead trying to get our webservice up and running we need to do a few things. First extensions parsing needs to be activated, this is done in **config/routes.php**:

```
Router::extensions('rss');
```

In the call above we've activated the .rss extension. When using *`Cake\Routing\Router::extensions()`* you can pass a string or an array of extensions as first argument. This will activate each extension/content-type for use in your application. Now when the address `articles/index.rss` is requested you will get an XML version of your `articles/index`. However, first we need to edit the controller to add in the rss-specific code.

[124] https://en.wikipedia.org/wiki/RSS

Controller Code

It is a good idea to add RequestHandler to your ArticlesController's `initialize()` method. This will allow a lot of automagic to occur:

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

Before we can make an RSS version of our `articles/index` we need to get a few things in order. It may be tempting to put the channel metadata in the controller action and pass it to your view using the *Cake\Controller\Controller::set()* method but this is inappropriate. That information can also go in the view. That will come later though, for now if you have a different set of logic for the data used to make the RSS feed and the data for the HTML view you can use the `Cake\Controller\Component\RequestHandler::isRss()` method, otherwise your controller can stay the same:

```
// Modify the Posts Controller action that corresponds to
// the action which deliver the rss feed, which is the
// Index action in our example.

public function index()
{
    if ($this->RequestHandler->isRss() ) {
        $articles = $this->Articles
            ->find()
            ->limit(20)
            ->order(['created' => 'desc']);
        $this->set(compact('articles'));
    } else {
        // this is not an Rss request, so deliver
        // data used by website's interface.
        $this->paginate = [
            'order' => ['created' => 'desc'],
            'limit' => 10
        ];
        $this->set('articles', $this->paginate($this->Articles));
        $this->set('_serialize', ['articles']);
    }
}
```

With all the View variables set we need to create an rss layout.

Layout

An Rss layout is very simple, put the following contents in **src/Template/Layout/rss/default.ctp**:

```
if (!isset($documentData)) {
    $documentData = [];
}
if (!isset($channelData)) {
    $channelData = [];
}
if (!isset($channelData['title'])) {
    $channelData['title'] = $this->fetch('title');
```

```
}
$channel = $this->Rss->channel([], $channelData, $this->fetch('content'));
echo $this->Rss->document($documentData, $channel);
```

It doesn't look like much but thanks to the power in the `RssHelper` it's doing a lot of lifting for us. We haven't set `$documentData` or `$channelData` in the controller, however in CakePHP your views can pass variables back to the layout. Which is where our `$channelData` array will come from setting all of the meta data for our feed.

Next up is view file for my articles/index. Much like the layout file we created, we need to create a **src/Template/Posts/rss/** directory and create a new **index.ctp** inside that folder. The contents of the file are below.

View

Our view, located at **src/Template/Posts/rss/index.ctp**, begins by setting the `$documentData` and `$channelData` variables for the layout, these contain all the metadata for our RSS feed. This is done by using the *`Cake\View\View::set()`* method which is analogous to the *`Cake\Controller\Controller::set()`* method. Here though we are passing the channel's metadata back to the layout:

```
$this->set('channelData', [
    'title' => __("Most Recent Posts"),
    'link' => $this->Url->build('/', true),
    'description' => __("Most recent posts."),
    'language' => 'en-us'
]);
```

The second part of the view generates the elements for the actual records of the feed. This is accomplished by looping through the data that has been passed to the view ($items) and using the `RssHelper::item()` method. The other method you can use, `RssHelper::items()` which takes a callback and an array of items for the feed. The callback method is usually called `transformRss()`. There is one downfall to this method, which is that you cannot use any of the other helper classes to prepare your data inside the callback method because the scope inside the method does not include anything that is not passed inside, thus not giving access to the TimeHelper or any other helper that you may need. The `RssHelper::item()` transforms the associative array into an element for each key value pair.

Note: You will need to modify the $link variable as appropriate to your application. You might also want to use a *virtual property* in your Entity.

```
foreach ($articles as $article) {
    $created = strtotime($article->created);

    $link = [
        'controller' => 'Articles',
        'action' => 'view',
        'year' => date('Y', $created),
        'month' => date('m', $created),
        'day' => date('d', $created),
        'slug' => $article->slug
    ];

    // Remove & escape any HTML to make sure the feed content will validate.
    $body = h(strip_tags($article->body));
    $body = $this->Text->truncate($body, 400, [
        'ending' => '...',
        'exact'  => true,
        'html'   => true,
```

```
    ]);

    echo  $this->Rss->item([], [
        'title' => $article->title,
        'link' => $link,
        'guid' => ['url' => $link, 'isPermaLink' => 'true'],
        'description' => $body,
        'pubDate' => $article->created
    ]);
}
```

You can see above that we can use the loop to prepare the data to be transformed into XML elements. It is important to filter out any non-plain text characters out of the description, especially if you are using a rich text editor for the body of your blog. In the code above we used `strip_tags()` and *`h()`* to remove/escape any XML special characters from the content, as they could cause validation errors. Once we have set up the data for the feed, we can then use the `RssHelper::item()` method to create the XML in RSS format. Once you have all this setup, you can test your RSS feed by going to your site `/posts/index.rss` and you will see your new feed. It is always important that you validate your RSS feed before making it live. This can be done by visiting sites that validate the XML such as Feed Validator or the w3c site at http://validator.w3.org/feed/.

Note: You may need to set the value of 'debug' in your core configuration to `false` to get a valid feed, because of the various debug information added automagically under higher debug settings that break XML syntax or feed validation rules.

Session

class `Cake\View\Helper\`**`SessionHelper`**(*View $view*, *array $config = []*)

Deprecated since version 3.0.0: The SessionHelper is deprecated in 3.x. Instead you should use either the *FlashHelper* or *access the session via the request*.

As a natural counterpart to the Session object, the Session Helper replicates most of the object's functionality and makes it available in your view.

The major difference between the SessionHelper and the Session object is that the helper does *not* have the ability to write to the session.

As with the session object, data is read by using *dot notation* array structures:

```
['User' => [
    'username' => 'super@example.com'
]];
```

Given the previous array structure, the node would be accessed by `User.username`, with the dot indicating the nested array. This notation is used for all SessionHelper methods wherever a `$key` is used.

`Cake\View\Helper\SessionHelper::`**`read`**(*string $key*)

Return type mixed

Read from the Session. Returns a string or array depending on the contents of the session.

`Cake\View\Helper\SessionHelper::`**`check`**(*string $key*)

Return type boolean

Check to see whether a key is in the Session. Returns a boolean representing the key's existence.

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 <br> 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:

```
Highlights $needle in $haystack <span class="highlight">using</span> the
$options['format'] string specified  or a default string.
```

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. An easy 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()[125] method.

[125] https://api.cakephp.org/3.x/class-Cake.I18n.Time.html#_i18nFormat

Url

class `Cake\View\Helper\`**`UrlHelper`**(*View $view*, *array $config = []*)

The UrlHelper makes it easy for you to generate URL's 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`**(*mixed $url = null*, *boolean|array $full = false*)

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 `full` 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
```

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 (starting with '/') with the full base URL prepended:

```
echo $this->Url->build('/posts', true);

// Output
http://somedomain.com/posts
```

URL with GET params 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 `?` key which is useful when you want to be explicit about the query string parameters you are using, or if you want a query string parameter that shares a name with one of your route placeholders.

URL for named route:

```
echo $this->Url->build(['_name' => 'product-page', 'slug' => 'i-m-slug']);

// Assuming route is setup like:
// $router->connect(
//     '/products/:slug',
//     [
//         'controller' => 'Products',
//         'action' => 'view'
//     ],
//     [
//         '_name' => 'product-page'
//     ]
// );
/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
]);
```

New in version 3.3.5: `build()` accepts an array as the 2nd argument as of 3.3.5

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');
```

New in version 3.2.4: The asset helper methods were added in 3.2.4.

For further information check Router::url[126] in the API.

Configuring Helpers

You load helpers in CakePHP by declaring them in a view class. An `AppView` class comes with every CakePHP application and is the ideal place to load helpers:

```
class AppView extends View
{
    public function initialize()
    {
        parent::initialize();
        $this->loadHelper('Html');
        $this->loadHelper('Form');
```

[126] https://api.cakephp.org/3.x/class-Cake.Routing.Router.html#_url

```
        $this->loadHelper('Flash');
    }
}
```

To load helpers from plugins use the *plugin syntax* used elsewhere in CakePHP:

```
$this->loadHelper('Blog.Comment');
```

You don't have to explicitly load 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 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 load helpers:

```
class AppView extends View
{
    public function initialize()
    {
        parent::initialize();
        if ($this->request->getParam('action') === 'index') {
            $this->loadHelper('ListPage');
        }
    }
}
```

You can also use your controller's `beforeRender` method to load helpers:

```
class ArticlesController extends AppController
{
    public function beforeRender(Event $event)
    {
        parent::beforeRender($event);
        $this->viewBuilder()->helpers(['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
```

```
{

    // initialize() hook is available since 3.2. For prior versions you can
    // override the constructor if required.
    public function initialize(array $config)
    {
        debug($config);
    }
}
```

Options can be specified when declaring helpers in controller as shown:

```
namespace App\Controller;

use App\Controller\AppController;

class AwesomeController extends AppController
{
    public $helpers = ['Awesome' => ['option1' => 'value1']];
}
```

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;

    protected $_defaultConfig = [
        'errorClass' => 'error',
        '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 `config()` method to read runtime configuration:

```
// Read the errorClass config option.
$class = $this->Awesome->config('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(Event $event)
    {
        parent::beforeRender($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()
    {
        $this->loadHelper('Html', [
            'className' => 'MyHtml'
        ]);
    }
}

// src/View/Helper/MyHtmlHelper.php
namespace App\View\Helper;

use Cake\View\Helper\HtmlHelper;

class MyHtmlHelper extends HtmlHelper
{
    // 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 class names you should omit the `Helper` suffix. For example: `$this->loadHelper('Link');`.

You'll also want to extend `Helper` to ensure things work correctly:

```
/* 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:

```
/* src/View/Helper/LinkHelper.php (using other helpers) */

namespace App\View\Helper;

use Cake\View\Helper;

class LinkHelper extends Helper
{
    public $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:

```
class AppView extends View
{
    public function initialize()
    {
        parent::initialize();
        $this->loadHelper('Link');
    }
}
```

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->_View->get()` like:

```
class AwesomeHelper extends Helper
{

    public $helpers = ['Html'];

    public someMethod()
    {
        // set meta description
        echo $this->Html->meta(
            'description', $this->_View->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->_View->element()` like:

```
class AwesomeHelper extends Helper
{
    public someFunction()
    {
        // output directly in your helper
        echo $this->_View->element(
            '/path/to/element',
            ['foo'=>'bar','bar'=>'foo']
        );

        // or return it to your view
        return $this->_View->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`(*Event $event*, *$viewFile*)
: Is called before each view file is rendered. This includes elements, views, parent views and layouts.

`Helper::afterRenderFile`(*Event $event*, *$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`(*Event $event*, *$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`(*Event $event*, *$viewFile*)
: Is called after the view has been rendered but before layout rendering has started.

`Helper::beforeLayout`(*Event $event*, *$layoutFile*)
: Is called before layout rendering starts. Receives the layout filename as an argument.

`Helper::afterLayout`(*Event $event*, *$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. The first are **repositories** or **table objects**. These objects provide access to collections of data. They allow you to save new records, modify/delete existing ones, define relations, and perform bulk operations. The second type of objects are **entities**. 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*.

Note: If you are familiar with previous versions of CakePHP, you should read the *New ORM Upgrade Guide* for important differences between CakePHP 3.0 and older versions of CakePHP.

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 could do:

```
use Cake\ORM\TableRegistry;

$articles = TableRegistry::get('Articles');

$query = $articles->find();

foreach ($query as $row) {
    echo $row->title;
}
```

Note that we didn't have to create any code or wire any configuration up. The conventions in CakePHP allow us to skip some boilerplate code and allow the framework to insert base classes when your application has not created a concrete

class. If we wanted to customize our ArticlesTable class adding some associations or defining some additional methods we would add the following to **src/Model/Table/ArticlesTable.php** after the `<?php` opening tag:

```
namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{

}
```

Table classes use the CamelCased version of the table name with the `Table` suffix as the class name. Once your class has been created you get a reference to it using the *`ORM\TableRegistry`* as before:

```
use Cake\ORM\TableRegistry;

// Now $articles is an instance of our ArticlesTable class.
$articles = TableRegistry::get('Articles');
```

Now that we have a concrete table class, we'll probably want to use a concrete entity class. 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\TableRegistry;

// Now an instance of ArticlesTable.
$articles = TableRegistry::get('Articles');
$query = $articles->find();

foreach ($query 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::config('default', ['url' => $dsn]);
```

Once created, you can access the connection object to start using it:

```
$connection = ConnectionManager::get('default');
```

Supported Databases

CakePHP supports the following relational database servers:

- MySQL 5.1+
- SQLite 3
- PostgreSQL 8+
- SQLServer 2008+
- Oracle (through a community plugin)

You will need the correct PDO extension installed for each of the above database drivers. Procedural APIs are not supported.

The Oracle database is supported through the Driver for Oracle Database[127] community plugin.

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:

[127] https://github.com/CakeDC/cakephp-oracle-driver

```
$results = $connection
    ->execute('SELECT * FROM articles WHERE id = :id', ['id' => 1])
    ->fetchAll('assoc');
```

It is also possible to use complex data types as arguments:

```
$results = $connection
    ->execute(
        'SELECT * FROM articles WHERE created >= :created',
        ['created' => new DateTime('1 day ago')],
        ['created' => 'datetime']
    )
    ->fetchAll('assoc');
```

Instead of writing the SQL manually, you can use the query builder:

```
$results = $connection
    ->newQuery()
    ->select('*')
    ->from('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;

$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' => 'sekret',
        'database' => 'my_app',
        'encoding' => 'utf8',
        '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::config()`. An example of that would be:

```
use Cake\Datasource\ConnectionManager;

ConnectionManager::config('default', [
    'className' => 'Cake\Database\Connection',
    'driver' => 'Cake\Database\Driver\Mysql',
    'persistent' => false,
    'host' => 'localhost',
    'username' => 'my_app',
    'password' => 'sekret',
    'database' => 'my_app',
    'encoding' => 'utf8',
    '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::config('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 implements 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.

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 (e.g. `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. If you wish to use UTF-8 encoding with MySQL connections you must use 'utf8' without the hyphen.

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 disable 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)

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.

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;

$conn = ConnectionManager::get('default');
```

Attempting to load connections that do not exist will throw an exception.

Creating Connections at Runtime

Using config() and get() you can create new connections that are not defined in your configuration files at runtime:

```
ConnectionManager::config('my_connection', $config);
$conn = ConnectionManager::get('my_connection');
```

See the *Configuration* for more information on the configuration data used when creating connections.

Data Types

class Cake\Database**Type**

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 Generally backed by CHAR or VARCHAR columns. Using the fixed option will force a CHAR column. In SQL Server, NCHAR and NVARCHAR types are 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.

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.

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 timezone naive DATE column type. The return value of this column type is `Cake\I18n\Date` which extends the native `DateTime` class.

datetime Maps to a timezone naive 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\Time`* which extends the built-in `DateTime` class and Chronos[128].

timestamp Maps to the TIMESTAMP 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. The 'json' type was added in 3.3.0

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.

Changed in version 3.3.0: The `json` type was added.

Changed in version 3.5.0: The `smallinteger` and `tinyinteger` types were added.

Adding Custom Types

static `Cake\Database\Type`**::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.

An easy way to fulfill the basic interface is to extend *`Cake\Database\Type`*. For example if we wanted to add a JSON type, we could make the following type class:

```
// in src/Database/Type/JsonType.php

namespace App\Database\Type;

use Cake\Database\Driver;
use Cake\Database\Type;
use PDO;

class JsonType extends Type
{

    public function toPHP($value, Driver $driver)
```

[128] https://github.com/cakephp/chronos

```
    {
        if ($value === null) {
            return null;
        }
        return json_decode($value, true);
    }

    public function marshal($value)
    {
        if (is_array($value) || $value === null) {
            return $value;
        }
        return json_decode($value, true);
    }

    public function toDatabase($value, Driver $driver)
    {
        return json_encode($value);
    }

    public function toStatement($value, Driver $driver)
    {
        if ($value === null) {
            return PDO::PARAM_NULL;
        }
        return PDO::PARAM_STR;
    }

}
```

By default the `toStatement()` method will treat values as strings which will work for our new type. 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:

```
use Cake\Database\Type;

Type::map('json', 'App\Database\Type\JsonType');
```

New in version 3.3.0: The JsonType described in this example was added to the core.

We can then overload the reflected schema data to use our new type, and CakePHP's database layer will automatically convert our JSON data when creating queries. You can use the custom types you've created by mapping the types in your Table's *_initializeSchema() method*:

```
use Cake\Database\Schema\TableSchema;

class WidgetsTable extends Table
{

    protected function _initializeSchema(TableSchema $schema)
    {
        $schema->columnType('widget_prefs', 'json');
        return $schema;
    }

}
```

Mapping Custom Datatypes to SQL Expressions

New in version 3.3.0: Support for mapping custom data types to SQL expressions was added in 3.3.0.

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($value)
    {
        // Parse the data from MySQL.
        return new static($value[0], $value[1]);
    }

    public function __construct($lat, $long)
    {
        $this->_lat = $lat;
        $this->_long = $long;
    }

    public function lat()
    {
        return $this->_lat;
    }

    public function long()
    {
        return $this->_long;
    }
}
```

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\Expression\FunctionExpression;
use Cake\Database\Type as BaseType;
use Cake\Database\Type\ExpressionTypeInterface;

class PointType extends BaseType implements ExpressionTypeInterface
{
    public function toPHP($value, Driver $d)
    {
        return Point::parse($value);
```

```
    }

    public function marshal($value)
    {
        if (is_string($value)) {
            $value = explode(',', $value);
        }
        if (is_array($value)) {
            return new Point($value[0], $value[1]);
        }
        return null;
    }

    public function toExpression($value)
    {
        if ($value instanceof Point) {
            return new FunctionExpression(
                'POINT',
                [
                    $value->lat(),
                    $value->long()
                ]
            );
        }
        if (is_array($value)) {
            return new FunctionExpression('POINT', [$value[0], $value[1]]);
        }
        // Handle other cases.
    }
}
```

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*.

Enabling Immutable DateTime Objects

New in version 3.2: Immutable date/time objects were added in 3.2.

Because Date/Time objects are easily mutated in place, CakePHP allows you to enable immutable value objects. This is best done in your application's **config/bootstrap.php** file:

```
Type::build('datetime')->useImmutable();
Type::build('date')->useImmutable();
Type::build('time')->useImmutable();
Type::build('timestamp')->useImmutable();
```

Note: New applications will have immutable objects enabled by default.

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::**query**(*$sql*)

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 `query()` which allows you to run already completed SQL queries:

```
$stmt = $conn->query('UPDATE articles SET published = 1 WHERE id = 2');
```

Cake\Database\Connection::**execute**(*$sql*, *$params*, *$types*)

The `query()` method does not allow for additional parameters. If you need additional parameters you should use the `execute()` method, which allows for placeholders to be used:

```
$stmt = $conn->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:

```
$stmt = $conn->execute(
    'UPDATE articles SET published_date = ? WHERE id = ?',
    [new DateTime('now'), 2],
    ['date', 'integer']
);
```

Cake\Database\Connection::**newQuery**()

This allows 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:

```
$query = $conn->newQuery();
$query->update('articles')
    ->set(['published' => true])
    ->where(['id' => 2]);
$stmt = $query->execute();
```

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 = $conn->newQuery();
$query->select('*')
    ->from('articles')
```

```
    ->where(['published' => true]);

foreach ($query as $row) {
    // Do something with the row.
}
```

Note: When you have an instance of *Cake\ORM\Query* you can use `all()` to get the result set for SELECT queries.

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:

```
$conn->begin();
$conn->execute('UPDATE articles SET published = ? WHERE id = ?', [true, 2]);
$conn->execute('UPDATE articles SET published = ? WHERE id = ?', [false, 4]);
$conn->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:

```
$conn->transactional(function ($conn) {
    $conn->execute('UPDATE articles SET published = ? WHERE id = ?', [true, 2]);
    $conn->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 `StatementDecorator` instance. It wraps the underlying basic statement object and provides a few additional features.

Preparing a Statement

You can create a statement object using `execute()`, or `prepare()`. The `execute()` method returns a statement with the provided values bound to it. While `prepare()` returns an incomplete statement:

```
// Statements from execute will have values bound to them already.
$stmt = $conn->execute(
    'SELECT * FROM articles WHERE published = ?',
    [true]
);

// Statements from prepare will be parameters for placeholders.
// You need to bind parameters before attempting to execute it.
$stmt = $conn->prepare('SELECT * FROM articles WHERE published = ?');
```

Once you've prepared a statement you can bind additional data and execute it.

Binding Values

Once you've created a prepared statement, you may need to bind additional data. You can bind multiple values at once using the `bind()` method, or bind individual elements using `bindValue`:

```
$stmt = $conn->prepare(
    'SELECT * FROM articles WHERE published = ? AND created > ?'
);

// Bind multiple values
$stmt->bind(
    [true, new DateTime('2013-01-01')],
    ['boolean', 'date']
);

// Bind a single value
$stmt->bindValue(1, true, 'boolean');
$stmt->bindValue(2, new DateTime('2013-01-01'), 'date');
```

When creating statements you can also use named array keys instead of positional ones:

```
$stmt = $conn->prepare(
    'SELECT * FROM articles WHERE published = :published AND created > :created'
);

// Bind multiple values
$stmt->bind(
    ['published' => true, 'created' => new DateTime('2013-01-01')],
    ['published' => 'boolean', 'created' => 'date']
);

// Bind a single value
$stmt->bindValue('published', true, 'boolean');
$stmt->bindValue('created', new DateTime('2013-01-01'), 'date');
```

Warning: You cannot mix positional and named array keys in the same statement.

Executing & Fetching Rows

After preparing a statement and binding data to it, you can execute it and fetch rows. Statements should be executed using the `execute()` method. Once executed, results can be fetched using `fetch()`, `fetchAll()` or iterating

the statement:

```
$stmt->execute();

// Read one row.
$row = $stmt->fetch('assoc');

// Read all rows.
$rows = $stmt->fetchAll('assoc');

// Read rows through iteration.
foreach ($rows as $row) {
    // Do work
}
```

Note: Reading rows through iteration will fetch rows in 'both' mode. This means you will get both the numerically indexed and associatively indexed results.

Getting Row Counts

After executing a statement, you can fetch the number of affected rows:

```
$rowCount = count($stmt);
$rowCount = $stmt->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 = $stmt->errorCode();
$info = $stmt->errorInfo();
```

Query Logging

Query logging can be enabled when configuring your connection by setting the `log` option to `true`. You can also toggle query logging at runtime, using `logQueries`:

```
// Turn query logging on.
$conn->logQueries(true);

// Turn query logging off
$conn->logQueries(false);
```

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::config('queries', [
    'className' => 'Console',
    'stream' => 'php://stderr',
    'scopes' => ['queriesLog']
]);

// File logging
Log::config('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:

```
$conn->driver()->autoQuoting(true);
```

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 *ORM Cache Shell* 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->query("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`**

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 Query Object

The easiest way to create a `Query` 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\TableRegistry;
$articles = TableRegistry::get('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\TableRegistry;

$query = TableRegistry::get('Articles')->find();

foreach ($query as $article) {
    debug($article->title);
}
```

For the remaining examples, assume that `$articles` is a *ORM\Table*. When inside controllers, you can use `$this->Articles` instead of `$articles`.

Almost every method in a `Query` object will return the same query, this means that `Query` 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 Query objects:

```
$query = $articles
    ->find()
    ->select(['id', 'name'])
    ->where(['id !=' => 1])
    ->order(['created' => 'DESC']);

foreach ($query as $article) {
    debug($article->created);
}
```

If you try to call `debug()` on a Query 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 `toArray()` methods:

```
$resultsIteratorObject = $articles
    ->find()
    ->where(['id >' => 1])
    ->all();

foreach ($resultsIteratorObject as $article) {
    debug($article->id);
}
```

```
$resultsArray = $articles
    ->find()
    ->where(['id >' => 1])
    ->toArray();

foreach ($resultsArray as $article) {
    debug($article->id);
}

debug($resultsArray[0]->title);
```

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 Query object to get its results. Query objects can also be used directly as the result object; trying to iterate the query, calling `toArray()` or some of the methods inherited from *Collection*, 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()->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');

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.

Queries 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 data. In short, it is important to remember that anything you can call on a Collection object, you can also do in a Query object:

```
// Use the combine() method from the collections library
// This is equivalent to find('list')
$keyValueList = $articles->find()->combine('id', 'title');

// An advanced example
$results = $articles->find()
    ->where(['id >' => 1])
    ->order(['title' => 'DESC'])
    ->map(function ($row) { // map() is a collection method, it executes the query
        $row->trimmedTitle = trim($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";
}
```

How Are Queries 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 `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.

If you want to take a look at what SQL CakePHP is generating, you can turn database *query logging* on.

The following sections will show you everything there is to know about using and combining the Query object methods to construct SQL statements and extract data.

Selecting Data

Most web applications make heavy use of `SELECT` queries. CakePHP makes building them a snap. To limit the fields fetched, you can use the `select()` method:

```
$query = $articles->find();
$query->select(['id', 'title', 'body']);
foreach ($query 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]);
```

See the *Advanced Conditions* section to find out how to construct more complex `WHERE` conditions. 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 `Query::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'], Query::OVERWRITE);
```

New in version 3.0.12: In addition to `order`, 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 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.

Selecting All Fields From a Table

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
```

New in version 3.1: Passing a table object to select() was added in 3.1.

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('*')]);
```

A number of commonly used functions can be created with the `func()` method:

- `sum()` Calculate a sum. The arguments will be treated as literal values.
- `avg()` Calculate an average. The arguments will be treated as literal values.
- `min()` Calculate the min of a column. The arguments will be treated as literal values.
- `max()` Calculate the max of a column. The arguments will be treated as literal values.
- `count()` Calculate the count. The arguments will be treated as literal values.
- `concat()` Concatenate two values together. The arguments are treated as bound parameters unless marked as literal.
- `coalesce()` Coalesce values. The arguments are treated as bound parameters unless marked as literal.

- `dateDiff()` Get the difference between two dates/times. The arguments are treated as bound parameters unless marked as literal.
- `now()` Take either 'time' or 'date' as an argument allowing you to get either the current time, or current date.
- `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.

New in version 3.1: `extract()`, `dateAdd()` and `dayOfWeek()` methods have been added.

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: ',
    'Categories.name' => 'identifier',
    ' - Age: ',
    '(DATEDIFF(NOW(), Articles.created))' => 'literal',
]);
$query->select(['link_title' => $concat]);
```

By making arguments with a value of `literal`, the ORM will know that the key should be treated as a literal SQL value. By making arguments with a value of `identifier`, the ORM will know that the key should be treated as a field identifier. The above would generate the following SQL on MySQL:

```
SELECT CONCAT(Articles.title, :c0, Categories.name, :c1, (DATEDIFF(NOW(), Articles.
↪created))) FROM articles;
```

The `:c0` value will have the `' -CAT:'` text bound when the query is executed.

In addition to the above functions, the `func()` method can be used to create any generic SQL function such as `year`, `date_format`, `convert`, etc. For example:

```
$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
]);
```

Would result in:

```
SELECT YEAR(created) as yearCreated, DATE_FORMAT(created, '%H:%i') as timeCreated␣
↪FROM articles;
```

You should remember to use the function builder whenever you need to put untrusted data into SQL functions or stored procedures:

```
// Use a stored procedure
$query = $articles->find();
$lev = $query->func()->levenshtein([$search, 'LOWER(title)' => 'literal']);
$query->where(function ($exp) use ($lev) {
    return $exp->between($lev, 0, $tolerance);
});

// Generated SQL would be
WHERE levenshtein(:c0, lower(street)) BETWEEN :c1 AND :c2
```

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)'
])
->group('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()
    ->addCase(
        $query->newExpr()->add(['published' => 'Y']),
        1,
        'integer'
    );
$unpublishedCase = $query->newExpr()
    ->addCase(
        $query->newExpr()->add(['published' => 'N']),
        1,
        'integer'
    );

$query->select([
    'number_published' => $query->func()->count($publishedCase),
    'number_unpublished' => $query->func()->count($unpublishedCase)
]);
```

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 = $cities->find()
    ->where(function ($exp, $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 = $cities->find()
    ->where(function ($exp, $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
```

Getting 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->hydrate(false); // Results as arrays intead 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' ...],
    ['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()`, `andWhere()` and `orWhere()`. The `where()` method works similar to the conditions arrays in previous versions of CakePHP:

```
$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 `orWhere()` and `andWhere()` methods to build your queries. Each method sets the combining operator used between the current and previous condition. For example:

```
$query = $articles->find()
    ->where(['author_id' => 2])
    ->orWhere(['author_id' => 3]);
```

The above will output SQL similar to:

```
SELECT * FROM articles WHERE (author_id = 2 OR author_id = 3)
```

By combining `orWhere()` and `andWhere()`, you can express complex conditions that use a mixture of operators:

```
$query = $articles->find()
    ->where(['author_id' => 2])
    ->orWhere(['author_id' => 3])
    ->andWhere([
        'published' => true,
        'view_count >' => 10
    ])
    ->orWhere(['promoted' => true]);
```

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
)
```

By using functions as the parameters to `orWhere()` and `andWhere()`, you can compose conditions together with the expression objects:

```
$query = $articles->find()
    ->where(['title LIKE' => '%First%'])
    ->andWhere(function ($exp) {
        return $exp->or_([
            'author_id' => 2,
            'is_highlighted' => true
        ]);
    });
```

The above would create SQL like:

```
SELECT *
FROM articles
WHERE (
    title LIKE '%First%'
    AND
    (author_id = 2 OR is_highlighted = 1)
)
```

The expression object that is passed into `where()` functions has two kinds of methods. The first type of methods are **combinators**. The `and_()` and `or_()` methods create new expression objects that change **how** conditions are combined. The second type of methods are **conditions**. Conditions are added into an expression where they are combined with the current combinator.

For example, calling `$exp->and_(...)` will create a new `Expression` object that combines all conditions it contains with `AND`. While `$exp->or_()` will create a new `Expression` object that combines all conditions added to it with `OR`. An example of adding conditions with an `Expression` object would be:

```
$query = $articles->find()
    ->where(function ($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. Much like how we would not need to call `or_()`, had we started our query with `orWhere()`. 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 = $articles->find()
    ->where(function ($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 `or_()` and `and_()` methods also allow you to use functions as their parameters. This is often easier to read than method chaining:

```
$query = $articles->find()
    ->where(function ($exp) {
        $orConditions = $exp->or_(function ($or) {
            return $or->eq('author_id', 2)
                ->eq('author_id', 5);
        });
        return $exp
            ->not($orConditions)
            ->lte('view_count', 10);
    });
```

You can negate sub-expressions using `not()`:

```
$query = $articles->find()
    ->where(function ($exp) {
        $orConditions = $exp->or_(['author_id' => 2])
            ->eq('author_id', 5);
        return $exp
            ->not($orConditions)
            ->lte('view_count', 10);
    });
```

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 = $articles->find()
    ->where(function ($exp, $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 = $cities->find()
    ->where(function ($exp, $q) {
        return $exp->eq('population', '10000');
    });
# WHERE population = 10000
```

- `notEq()` Creates an inequality condition:

```
$query = $cities->find()
    ->where(function ($exp, $q) {
        return $exp->notEq('population', '10000');
    });
# WHERE population != 10000
```

- `like()` Creates a condition using the `LIKE` operator:

```
$query = $cities->find()
    ->where(function ($exp, $q) {
        return $exp->like('name', '%A%');
    });
# WHERE name LIKE "%A%"
```

- `notLike()` Creates a negated `LIKE` condition:

```
$query = $cities->find()
    ->where(function ($exp, $q) {
        return $exp->notLike('name', '%A%');
    });
# WHERE name NOT LIKE "%A%"
```

- `in()` Create a condition using `IN`:

```
$query = $cities->find()
    ->where(function ($exp, $q) {
        return $exp->in('country_id', ['AFG', 'USA', 'EST']);
    });
# WHERE country_id IN ('AFG', 'USA', 'EST')
```

- `notIn()` Create a negated condition using `IN`:

```
$query = $cities->find()
    ->where(function ($exp, $q) {
        return $exp->notIn('country_id', ['AFG', 'USA', 'EST']);
    });
# WHERE country_id NOT IN ('AFG', 'USA', 'EST')
```

- `gt()` Create a > condition:

```
$query = $cities->find()
    ->where(function ($exp, $q) {
        return $exp->gt('population', '10000');
    });
# WHERE population > 10000
```

- `gte()` Create a >= condition:

```
$query = $cities->find()
    ->where(function ($exp, $q) {
```

```
        return $exp->gte('population', '10000');
    });
# WHERE population >= 10000
```

- `lt()` Create a < condition:

```
$query = $cities->find()
    ->where(function ($exp, $q) {
        return $exp->lt('population', '10000');
    });
# WHERE population < 10000
```

- `lte()` Create a <= condition:

```
$query = $cities->find()
    ->where(function ($exp, $q) {
        return $exp->lte('population', '10000');
    });
# WHERE population <= 10000
```

- `isNull()` Create an `IS NULL` condition:

```
$query = $cities->find()
    ->where(function ($exp, $q) {
        return $exp->isNull('population');
    });
# WHERE (population) IS NULL
```

- `isNotNull()` Create a negated `IS NULL` condition:

```
$query = $cities->find()
    ->where(function ($exp, $q) {
        return $exp->isNotNull('population');
    });
# WHERE (population) IS NOT NULL
```

- `between()` Create a `BETWEEN` condition:

```
$query = $cities->find()
    ->where(function ($exp, $q) {
        return $exp->between('population', 999, 5000000);
    });
# WHERE population BETWEEN 999 AND 5000000,
```

- `exists()` Create a condition using `EXISTS`:

```
$subquery = $cities->find()
    ->select(['id'])
    ->where(function ($exp, $q) {
        return $exp->equalFields('countries.id', 'cities.country_id');
    })
    ->andWhere(['population >', 5000000]);

$query = $countries->find()
    ->where(function ($exp, $q) use ($subquery) {
        return $exp->exists($subquery);
    });
# WHERE EXISTS (SELECT id FROM cities WHERE countries.id = cities.country_id AND␣
↪population > 5000000)
```

- `notExists()` Create a negated condition using `EXISTS`:

```
$subquery = $cities->find()
    ->select(['id'])
    ->where(function ($exp, $q) {
        return $exp->equalFields('countries.id', 'cities.country_id');
    })
    ->andWhere(['population >', 5000000]);

$query = $countries->find()
    ->where(function ($exp, $q) use ($subquery) {
        return $exp->notExists($subquery);
    });
# WHERE NOT EXISTS (SELECT id FROM cities WHERE countries.id = cities.country_id
↪AND population > 5000000)
```

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. See the *Using SQL Functions* section for how to safely include unsafe data into function calls.

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 create `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 create `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()`, `group()`, `select()` and many other methods.

Warning: Using expression objects leaves you vulnerable to SQL injection. You should avoid interpolating user data into expressions.

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 `Query` 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')
    ->group(['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 `Query` 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. The `Model.beforeFind` event is triggered.
2. If the query has results set, those will be returned.
3. The cache key will be resolved and cache data will be read. If the cache data is not empty, those results will be returned.
4. If the cache misses, the query will be executed 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 define eager loaded associations 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'
]);
```

You can select fields from all associations with multiple easy `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);
```

Passing Conditions to Contain

When using `contain()` you are able to restrict the data returned by the associations and filter them by conditions:

```
// In a controller or table method.

$query = $articles->find()->contain('Comments', function ($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 (\Cake\ORM\Query $query) {
        return $query->select(['body', 'author_id'])
        ->where(['Comments.approved' => true]);
    }
];
```

Note: When you limit the fields that are fetched from an association, you **must** ensure that the foreign key columns are selected. Failing to select foreign key fields will cause associated data to not be present in the final result.

It is also possible to restrict deeply-nested associations using the dot notation:

```
$query = $articles->find()->contain([
    'Comments',
    'Authors.Profiles' => function ($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 = $articles->find()->contain('Comments', function ($q) {
    return $q->find('approved')->find('popular');
});
```

Note: For `BelongsTo` and `HasOne` associations only the `where` and `select` clauses are used when loading the associated records. For the rest of the association types you can use every clause that the query object provides.

If you need full control over the query that is generated, you can tell `contain()` to not append the `foreignKey` constraints to the generated query. In that case you should use an array passing `foreignKey` and `queryBuilder`:

```
$query = $articles->find()->contain([
    'Authors' => [
        'foreignKey' => false,
        'queryBuilder' => function ($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, if you have multiple associations, you can use `enableAutoFields()`:

```
// Select id & title from articles, but all fields off of Users, Comments
// and Tags.
$query->select(['id', 'title'])
    ->contain(['Comments', 'Tags'])
    ->enableAutoFields(true) // Prior to 3.4.0 use autoFields(true)
    ->contain(['Users' => function($q) {
        return $q->autoFields(true);
    }]);
```

New in version 3.1: Selecting columns via an association object was added in 3.1

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 is surprisingly easy, and the syntax should be already familiar to you:

```
// 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 you both match and contain the same association, you can expect to get both the `_matchingData` and standard association properties in your results.

Using innerJoinWith

Using the `matching()` function, as we saw already, will create an `INNER JOIN` with the specified association and will also load the fields into the result set.

There may be cases where you want to use `matching()` but are not interested in loading the fields into the result set. For this purpose, you can use `innerJoinWith()`:

```
$query = $articles->find();
$query->innerJoinWith('Tags', function ($q) {
    return $q->where(['Tags.name' => 'CakePHP']);
});
```

The `innerJoinWith()` method works the same as `matching()`, that means that you can use dot notation to join deeply nested associations:

```
$query = $products->find()->innerJoinWith(
    'Shops.Cities.Countries', function ($q) {
        return $q->where(['Countries.name' => 'Japan']);
    }
);
```

Again, the only difference is that no additional columns will be added to the result set, and no `_matchingData` property will be set.

New in version 3.1: Query::innerJoinWith() was added in 3.1

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.

New in version 3.1: Query::notMatching() was added in 3.1

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')
    ->group(['Articles.id'])
    ->enableAutoFields(true); // Prior to 3.4.0 use autoFields(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']);
    })
    ->group(['Authors.id'])
    ->enableAutoFields(true); // Prior to 3.4.0 use autoFields(true);
```

This function will not load any columns from the specified associations into the result set.

New in version 3.1: Query::leftJoinWith() was added in 3.1

Adding Joins

In addition to loading related data with `contain()`, you can also add additional joins with the query builder:

```
$query = $articles->find()
    ->hydrate(false)
    ->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()
    ->hydrate(false)
    ->join([
        'c' => [
            'table' => 'comments',
            'type' => 'LEFT',
            'conditions' => 'c.article_id = articles.id',
        ],
        'u' => [
            'table' => 'users',
            'type' => 'INNER',
            'conditions' => 'u.id = articles.user_id',
        ]
    ]);
```

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()
    ->hydrate(false)
```

```
    ->join([
        'c' => [
            'table' => 'comments',
            'type' => 'LEFT',
            'conditions' => [
                'c.created >' => new DateTime('-5 days'),
                'c.moderated' => true,
                'c.article_id = articles.id'
            ]
        ],
    ], ['c.created' => 'datetime', 'c.moderated' => 'boolean']);
```

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' => [
                '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 not use `find()` to create insert queries. Instead, create a new `Query` object using `query()`:

```
$query = $articles->query();
$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->query();
$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 *ORM\Table::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 = $articles->query()
    ->insert(['title', 'body', 'published'])
    ->values($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 `query()`:

```
$query = $articles->query();
$query->update()
    ->set(['published' => true])
    ->where(['id' => $id])
    ->execute();
```

Generally, it is easier to update data using entities and `ORM\Table::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 should not use `find()` to create delete queries. Instead, create new a query object using `query()`:

```
$query = $articles->query();
$query->delete()
    ->where(['id' => $id])
    ->execute();
```

Generally, it is easier to delete data using entities and *ORM\Table::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:

```
$query->where(function ($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. Similar to *binding values to prepared statements*, 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

The query builder is capable of building complex queries like `UNION` queries and sub-queries.

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 are a powerful feature in relational databases and building them in CakePHP is fairly intuitive. By composing queries together, you can make subqueries:

```
$matchingComment = $articles->association('Comments')->find()
    ->select(['article_id'])
    ->distinct()
    ->where(['comment LIKE' => '%CakePHP%']);

$query = $articles->find()
    ->where(['id' => $matchingComment]);
```

Subqueries are accepted anywhere a query expression can be used. For example, in the `select()` and `join()` methods.

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()`.

Executing Complex Queries

While the query builder makes it easy to build most queries, 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 using the `setTable()` method:

```
namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{

    public function initialize(array $config)
    {
        $this->setTable('my_table');

        // Prior to 3.4.0
        $this->table('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)
    {
        $this->setPrimaryKey('my_id');

        // Prior to 3.4.0
        $this->primaryKey('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)
    {
        $this->setEntityClass('App\Model\Entity\PO');

        // Prior to 3.4.0
        $this->entityClass('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 `TableRegistry` class:

```
// In a controller or table method.
use Cake\ORM\TableRegistry;

$articles = TableRegistry::get('Articles');
```

The TableRegistry class 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 TableRegistry* 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 = TableRegistry::get('PluginName.Articles');

// Vendor prefixed plugin table
$articlesTable = TableRegistry::get('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.

Event List

- `Model.initialize`
- `Model.beforeMarshal`
- `Model.beforeFind`
- `Model.buildValidator`
- `Model.buildRules`
- `Model.beforeRules`
- `Model.afterRules`
- `Model.beforeSave`
- `Model.afterSave`
- `Model.afterSaveCommit`
- `Model.beforeDelete`

- Model.afterDelete
- Model.afterDeleteCommit

initialize

Cake\ORM\Table::**initialize**(*Event $event*, *ArrayObject $data*, *ArrayObject $options*)

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 array(
            'Model.initialize' => 'initializeEvent',
        );
    }
    public function initializeEvent($event)
    {
        $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**(*Event $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.

beforeFind

Cake\ORM\Table::**beforeFind**(*Event $event*, *Query $query*, *ArrayObject $options*, *$primary*)

The Model.beforeFind event is fired before each find operation. By stopping the event and supplying a return value you can bypass the find operation entirely. 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.

You might use this callback to restrict find operations based on a user's role, or make caching decisions based on the current load.

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`(*Event $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`(*Event $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`(*Event $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`(*Event $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`(*Event $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. How to stop an event is documented *here*.

afterSave

`Cake\ORM\Table::afterSave`(*Event $event*, *EntityInterface $entity*, *ArrayObject $options*)

The `Model.afterSave` event is fired after an entity is saved.

afterSaveCommit

`Cake\ORM\Table::**afterSaveCommit**` (*Event $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**` (*Event $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. How to stop an event is documented *here*.

afterDelete

`Cake\ORM\Table::**afterDelete**` (*Event $event*, *EntityInterface $entity*, *ArrayObject $options*)

The `Model.afterDelete` event is fired after an entity has been deleted.

afterDeleteCommit

`Cake\ORM\Table::**afterDeleteCommit**` (*Event $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.

Behaviors

`Cake\ORM\Table::**addBehavior**` (*$name*, *array $options = []*)

Behaviors provide an easy 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)
    {
        $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)
    {
        $this->addBehavior('Timestamp', [
            'events' => [
                'Model.beforeSave' => [
                    'created_at' => 'new',
                    'modified_at' => 'always'
                ]
            ]
        ]);
    }
}
```

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() {
        return 'replica_db';
    }
}
```

Note: The `defaultConnectionName()` method **must** be static.

Using the TableRegistry

class `Cake\ORM\`**TableRegistry**

As we've seen earlier, the TableRegistry class provides an easy to use factory/registry for accessing your applications table instances. It provides a few other useful features as well.

Configuring Table Objects

static Cake\ORM\TableRegistry::**get**(*$alias*, *$config*)

When loading tables from the registry you can customize their dependencies, or use mock objects by providing an `$options` array:

```
$articles = TableRegistry::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 `config()` method. Configuration data is stored *per alias*, and can be overridden by an object's `initialize()` method:

```
TableRegistry::config('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

static Cake\ORM\TableRegistry::**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:

```
TableRegistry::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 persistent properties and methods to manipulate and access the data they contain.

Entities are created for you by CakePHP each time you use `find()` on a table object.

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 properties 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')
]);
```

Another way of getting new entities is using the `newEntity()` method from the `Table` objects:

```
use Cake\ORM\TableRegistry;

$article = TableRegistry::get('Articles')->newEntity();
$article = TableRegistry::get('Articles')->newEntity([
    'id' => 1,
    'title' => 'New Article',
    'created' => new DateTime('now')
]);
```

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:

```
$article->set('title', 'This is my first post');
echo $article->get('title');
```

When using `set()` you can update multiple properties 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 whitelist which fields can be set with mass assignment.

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 properties are read or set.

Accessors use the convention of `_get` followed by the CamelCased version of the field name.

`Cake\ORM\Entity::get`(*$field*)

They receive the basic value stored in the `_properties` array as their only argument. Accessors will be used when saving entities, so be careful when defining methods that format data, as the formatted data will be persisted. For example:

```
namespace App\Model\Entity;

use Cake\ORM\Entity;

class Article extends Entity
{
    protected function _getTitle($title)
    {
        return ucwords($title);
    }
}
```

The accessor would be run when getting the property through any of these two ways:

```
echo $user->title;
echo $user->get('title');
```

You can customize how properties get set by defining a mutator:

`Cake\ORM\Entity::set` (*$field = null*, *$value = null*)

Mutator methods should always return the value that should be stored in the property. As you can see above, you can also use mutators to set other calculated properties. When doing this, be careful to not introduce any loops, as CakePHP will not prevent infinitely looping mutator methods.

Mutators allow you to convert properties as they are set, or create calculated data. Mutators and accessors are applied when properties are read using object notation, or using `get()` and `set()`. For example:

```
namespace App\Model\Entity;

use Cake\ORM\Entity;
use Cake\Utility\Text;

class Article extends Entity
{

    protected function _setTitle($title)
    {
        return Text::slug($title);
    }

}
```

The mutator would be run when setting the property through any of these two ways:

```
$user->title = 'foo'; // slug is set as well
$user->set('title', 'foo'); // slug is set as well
```

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 properties as those are not persisted.

Creating Virtual Properties

By defining accessors you can provide access to fields/properties 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->_properties['first_name'] . '  ' .
            $this->_properties['last_name'];
    }

}
```

You can access virtual properties as if they existed on the entity. The property name will be the lower case and underscored version of the method:

```
echo $user->full_name;
```

Do bear in mind that virtual properties cannot be used in finds. If you want virtual properties to be part of JSON or array representations of your entities, see *Exposing Virtual Properties*.

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 properties 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->dirty('title');
```

You can also flag fields as being modified. This is handy when appending into array properties:

```
// Add a comment and mark the field as changed.
$article->comments[] = $newComment;
$article->dirty('comments', true);
```

In addition you can also base your conditional code on the original property values by using the `getOriginal()` method. This method will either return the original value of the property if it has been modified or its actual value.

You can also check for changes to any property in the entity:

```
// See if the entity has changed
$article->dirty();
```

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 properties of an `Entity` you may call:

```
$dirtyFields = $entity->getDirty();
```

New in version 3.4.3: `getDirty()` has been added.

Validation Errors

`Cake\ORM\Entity::errors`(*$field = null*, *$errors = null*)

After you *save an entity* any validation errors will be stored on the entity itself. You can access any validation errors using the `getErrors()` or `getError()` method:

```
// Get all the errors
$errors = $user->getErrors();
// Prior to 3.4.0
$errors = $user->errors();

// Get the errors for a single field.
$errors = $user->getError('password');
// Prior to 3.4.0
$errors = $user->errors('password');
```

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']]);
// Prior to 3.4.0
$user->errors('password', ['Password is required.']);
```

Mass Assignment

While setting properties 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 properties 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 $_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 $_accessible = [
        'title' => true,
        'body' => true,
        '*' => false,
    ];
}
```

Note: If the `*` property 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 `accessible` method:

```
// Make user_id accessible.
$article->accessible('user_id', true);

// Make title guarded.
$article->accessible('title', false);
```

Note: Modifying accessible fields effects 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:

```
$article->set($properties, ['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 `isNew()` as a setter:

```
$article->isNew(false);

$article->isNew(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[129]

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:

[129] https://github.com/jeremyharris/cakephp-lazyload

```
// 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:

```
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 Properties

By default virtual fields are not exported when converting entities to arrays or JSON. In order to expose virtual properties you need to make them visible. When defining your entity class you can provide a list of virtual properties 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 `virtualProperties`:

```
$user->virtualProperties(['full_name', 'is_admin']);
```

Hiding Properties

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 properties 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 `hiddenProperties`:

```
$user->hiddenProperties(['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.

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($query)` Shows the SQL and bound params, does not show results.
- `debug($query->all())` Shows the ResultSet properties (not the results).
- `debug($query->toArray())` An easy way to show each of the results.

- `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 view 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, [
    'finder' => 'translations',
]);
```

Using Finders to Load Data

`Cake\ORM\Table::find`(*$type*, *$options = []*)

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 an easy 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* object. The Query class allows you to further refine a query after creating it. Query 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');

// Iteration will execute the query.
foreach ($query as $row) {
}

// 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->toArray();

// Converting the query to an array will 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()`. This can help with testing as there are fewer methods to mock:

```
// 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 options supported by find() 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->enableHydration(false);
// Prior to 3.4.0
$query->hydrate(false);

// $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. 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)
    {
        $this->setDisplayField('title');
        // Prior to 3.4.0
        $this->displayField('title');
    }
}
```

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' => 'title'
]);
$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' => 'title',
    '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']);
```

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->_properties['first_name'] . ' ' . $this->_properties['last_name']
      . ' / ' . __('User ID %s', $this->_properties['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');
    }
]);
```

You can also fetch the label in the list directly using.

```
// In AuthorsTable::initialize():
$this->setDisplayField('label'); // Will utilize Author::_getLabel()
// 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 published articles we would do the following:

```
use Cake\ORM\Query;
use Cake\ORM\Table;

class ArticlesTable extends Table
{

    public function findOwnedBy(Query $query, array $options)
    {
        $user = $options['user'];
        return $query->where(['author_id' => $user->id]);
    }

}

// In a controller or table method.
$articles = TableRegistry::get('Articles');
$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:

```
// In a controller or table method.
$articles = TableRegistry::get('Articles');
$query = $articles->find('published')->find('recent');
```

While all the examples so far have show 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');

// In a table method
$users = TableRegistry::get('Users');
// The following two calls are equal.
$query = $users->findByUsername('joebob');
$query = $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 come with some additional performance overhead.

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 define eager loaded associations 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'
]);
```

You can select fields from all associations with multiple easy `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);
```

Passing Conditions to Contain

When using `contain()` you are able to restrict the data returned by the associations and filter them by conditions:

```
// In a controller or table method.

$query = $articles->find()->contain('Comments', function ($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 (\Cake\ORM\Query $query) {
        return $query->select(['body', 'author_id'])
        ->where(['Comments.approved' => true]);
    }
];
```

Note: When you limit the fields that are fetched from an association, you **must** ensure that the foreign key columns are selected. Failing to select foreign key fields will cause associated data to not be present in the final result.

It is also possible to restrict deeply-nested associations using the dot notation:

```
$query = $articles->find()->contain([
    'Comments',
    'Authors.Profiles' => function ($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 = $articles->find()->contain('Comments', function ($q) {
    return $q->find('approved')->find('popular');
});
```

Note: For `BelongsTo` and `HasOne` associations only the `where` and `select` clauses are used when loading the associated records. For the rest of the association types you can use every clause that the query object provides.

If you need full control over the query that is generated, you can tell `contain()` to not append the `foreignKey` constraints to the generated query. In that case you should use an array passing `foreignKey` and `queryBuilder`:

```
$query = $articles->find()->contain([
    'Authors' => [
        'foreignKey' => false,
        'queryBuilder' => function ($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, if you have multiple associations, you can use `enableAutoFields()`:

```
// Select id & title from articles, but all fields off of Users, Comments
// and Tags.
$query->select(['id', 'title'])
    ->contain(['Comments', 'Tags'])
    ->enableAutoFields(true) // Prior to 3.4.0 use autoFields(true)
    ->contain(['Users' => function($q) {
        return $q->autoFields(true);
    }]);
```

New in version 3.1: Selecting columns via an association object was added in 3.1

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) {
    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 is surprisingly easy, and the syntax should be already familiar to you:

```
// 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 you both match and contain the same association, you can expect to get both the `_matchingData` and standard association properties in your results.

Using innerJoinWith

Using the `matching()` function, as we saw already, will create an `INNER JOIN` with the specified association and will also load the fields into the result set.

There may be cases where you want to use `matching()` but are not interested in loading the fields into the result set. For this purpose, you can use `innerJoinWith()`:

```
$query = $articles->find();
$query->innerJoinWith('Tags', function ($q) {
```

```
    return $q->where(['Tags.name' => 'CakePHP']);
});
```

The `innerJoinWith()` method works the same as `matching()`, that means that you can use dot notation to join deeply nested associations:

```
$query = $products->find()->innerJoinWith(
    'Shops.Cities.Countries', function ($q) {
        return $q->where(['Countries.name' => 'Japan']);
    }
);
```

Again, the only difference is that no additional columns will be added to the result set, and no `_matchingData` property will be set.

New in version 3.1: Query::innerJoinWith() was added in 3.1

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.

New in version 3.1: Query::notMatching() was added in 3.1

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')
    ->group(['Articles.id'])
    ->enableAutoFields(true); // Prior to 3.4.0 use autoFields(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']);
    })
    ->group(['Authors.id'])
    ->enableAutoFields(true); // Prior to 3.4.0 use autoFields(true);
```

This function will not load any columns from the specified associations into the result set.

New in version 3.1: Query::leftJoinWith() was added in 3.1

Changing Fetching Strategies

As you may know already, `belongsTo` and `hasOne` associations are loaded using a `JOIN` in the main finder query. While this improves query and fetching speed and allows for creating more expressive conditions when retrieving data, this may be a problem when you want to apply certain clauses to the finder query for the association, such as `order()` or `limit()`.

For example, if you wanted to get the first comment of an article as an association:

```
$articles->hasOne('FirstComment', [
    'className' => 'Comments',
```

```
    'foreignKey' => 'article_id'
]);
```

In order to correctly fetch the data from this association, we will need to tell the query to use the `select` strategy, since we want order by a particular column:

```
$query = $articles->find()->contain([
    'FirstComment' => [
            'strategy' => 'select',
            'queryBuilder' => function ($q) {
                return $q->order(['FirstComment.created' =>'ASC'])->limit(1);
            }
    ]
]);
```

Dynamically changing the strategy in this way will only apply to a specific query. If you want to make the strategy change permanent you can do:

```
$articles->FirstComment->setStrategy('select');
// Prior to 3.4.0
$articles->FirstComment->strategy('select');
```

Using the `select` strategy is also a great way of making associations with tables in another database, since it would not be possible to fetch records using `joins`.

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**.

You can also make the strategy permanent for the association by doing:

```
$articles->Comments->setStrategy('subquery');
// Prior to 3.4.0
$articles->Comments->strategy('subquery');
```

Lazy Loading Associations

While CakePHP makes it easy to eager load 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. Like Query objects, 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. If you need work with a data set that does not fit into memory you can disable buffering on the query to stream results:

```
$query->enableBufferResults(false);
// Prior to 3.4.0
$query->bufferResults(false);
```

Turning buffering off has a few caveats:

1. You will not be able to iterate a result set more than once.
2. You will also not be able to iterate & cache the results.
3. Buffering cannot be disabled for queries that eager load hasMany or belongsToMany associations, as these association types require eagerly loading all results so that dependent queries can be generated.

Warning: Streaming results will still allocate memory for the entire results when using PostgreSQL and SQL Server. This is due to limitations in PDO.

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.

In addition to making serialization easy, 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.
$articles = TableRegistry::get('Articles');
$query = $articles->find()->contain(['Tags']);

$reducer = function ($output, $value) {
    if (!in_array($value, $output)) {
        $output[] = $value;
    }
    return $output;
};

$uniqueTags = $query->all()
    ->extract('tags.name')
    ->reduce($reducer, []);
```

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
$articles = TableRegistry::get('Articles');
$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 Query or ResultSet is Empty

You can use the `isEmpty()` method on a Query or ResultSet object to see if it has any rows in it. Calling `isEmpty()` on a Query object will evaluate the query:

```
// Check a query.
$query->isEmpty();

// 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']);
```

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 property generation or special data formatting, sometimes you need to change the data structure in a more fundamental way.

For those cases, the `Query` 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);

foreach ($articlesByStatus as $status => $articles) {
    echo sprintf("There are %d %s articles", count($articles), $status);
}
```

The above will ouput 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 = $articles->find()
    ->where(['published' => true])
    ->andWhere(['published_date >=' => new DateTime('2014-01-01')])
    ->enableHydrate(false) // Prior to 3.4.0 use hydrate(false)
    ->mapReduce($mapper, $reducer)
    ->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']);
    $mr->emitIntermediate(-$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 = $friends->find()
    ->enableHydrate(false) // Prior to 3.4.0 use hydrate(false)
    ->mapReduce($mapper, $reducer)
    ->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(Query $query, array $options)
{
    return $query->where(['published' => true]);
}

public function findRecent(Query $query, array $options)
{
    return $query->where(['created >=' => new DateTime('1 day ago')]);
}

public function findCommonWords(Query $query, array $options)
{
    // Same as in the common words example in the previous section
    $mapper = ...;
    $reducer = ...;
    return $query->mapReduce($mapper, $reducer);
}

$commonWords = $articles
    ->find('commonWords')
    ->find('published')
    ->find('recent');
```

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);
```

Removing All Stacked Map-reduce Operations

Under some circumstances you may want to modify a `Query` 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->errors()) {
    // Entity failed validation.
}
```

New in version 3.4.0: The `getErrors()` function was added.

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
            ->requirePresence('title', 'create')
            ->notEmpty('title');

        $validator
            ->allowEmpty('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
            ->add('title', 'notEmpty', [
                'rule' => 'notEmpty',
                'message' => __('You need to provide a title'),
            ])
            ->add('body', 'notEmpty', [
                'rule' => 'notEmpty',
                'message' => __('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, it is easy to break their construction process into multiple reusable steps:

```
// UsersTable.php

public function validationDefault(Validator $validator)
{
    $validator->notEmpty('username');
    $validator->notEmpty('password');
    $validator->add('email', 'valid-email', ['rule' => 'email']);
    ...

    return $validator;
}

public function validationHardened(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\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
            ->add('role', 'validRole', [
                'rule' => 'isValidRole',
                'message' => __('You need to provide a valid role'),
                'provider' => 'table',
            ]);
        return $validator;
    }

    public function isValidRole($value, array $context)
    {
        return in_array($value, ['admin', 'editor', 'author'], true);
    }

}
```

You can also use closures for validation rules:

```
$validator->add('name', 'myRule', [
    'rule' => function ($data, $provider) {
        if ($data > 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->validator('default');

$hardenedValidator = $usersTable->validator('hardened');
```

Deprecated since version 3.5.0: `validator()` is deprecated. Use `getValidator()` instead.

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)
{
    $this->_validatorClass = '\FullyNamespaced\Custom\Validator';
}
```

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 domain rules are:

- Ensuring email uniqueness
- State transitions or workflow steps (e.g., updating an invoice's status).
- Preventing the modification of soft deleted items.
- Enforcing usage/rate limit caps.

Domain 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)
{
    // 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 `errors()` method on the entity:

```
$entity->errors(); // 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. This means that setting `$user->account->id` will not trigger the above rule.

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 (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]
));

// 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]
));
```

New in version 3.3.0: The `allowNullableNulls` and `allowMultipleNulls` options were added.

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'));
```

New in version 3.3.0: The `validCount()` method was added in 3.3.0.

Using Entity Methods as Rules

You may want to use entity methods as domain rules:

```
$rules->add(function ($entity, $options) {
    return $entity->isOkLooking();
}, 'ruleName');
```

Creating Custom re-usable Rules

You may want to re-use custom domain rules. You can do so by creating your own invokable rule:

```
use App\ORM\Rule\IsUniqueWithNulls;
// ...
public function buildRules(RulesChecker $rules)
{
    $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
{
    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 make your domain rules easy to test.

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(...);
    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)
{
    $rules->add($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)
{
    $check = function($order) {
        return $order->price < 100 && $order->shipping_mode === 'free';
    };
    $rules->add($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->add('email', 'valid', [
        'rule' => 'email',
        'message' => 'Invalid email'
```

```
    ]);
    ...
    return $validator;
}

public function buildRules(RulesChecker $rules)
{
    // Add validation rules
    $rules->add(function($entity) {
        $data = $entity->extract($this->schema()->columns(), true);
        $validator = $this->validator('default');
        $errors = $validator->errors($data, $entity->isNew());
        $entity->errors($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->errors('email'); // Invalid email
```

The same result can be expected when using `newEntity()` or `patchEntity()`:

```
$userEntity = $usersTable->newEntity(['email' => 'not an email!!']);
$userEntity->errors('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\TableRegistry;

$articlesTable = TableRegistry::get('Articles');
$article = $articlesTable->newEntity();
```

```
$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 equally easy, and the `save()` method is also used for that purpose:

```
use Cake\ORM\TableRegistry;

$articlesTable = TableRegistry::get('Articles');
$article = $articlesTable->get(12); // Return article with id 12

$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 = TableRegistry::get('Articles');
$author = $articlesTable->Authors->findByUserName('mark')->first();

$article = $articlesTable->newEntity();
$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->newEntity();
$firstComment->body = 'The CakePHP features are outstanding';

$secondComment = $articlesTable->Comments->newEntity();
$secondComment->body = 'CakePHP performance is terrific!';

$tag1 = $articlesTable->Tags->findByName('cakephp')->first();
$tag2 = $articlesTable->Tags->newEntity();
$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->newEntity();
$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']])
    ->toArray();

$articlesTable->Tags->unlink($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 easy and efficient way to convert one or many entities from request data. You can convert a single entity using:

```
// In a controller
$articles = TableRegistry::get('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 = TableRegistry::get('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 = TableRegistry::get('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:

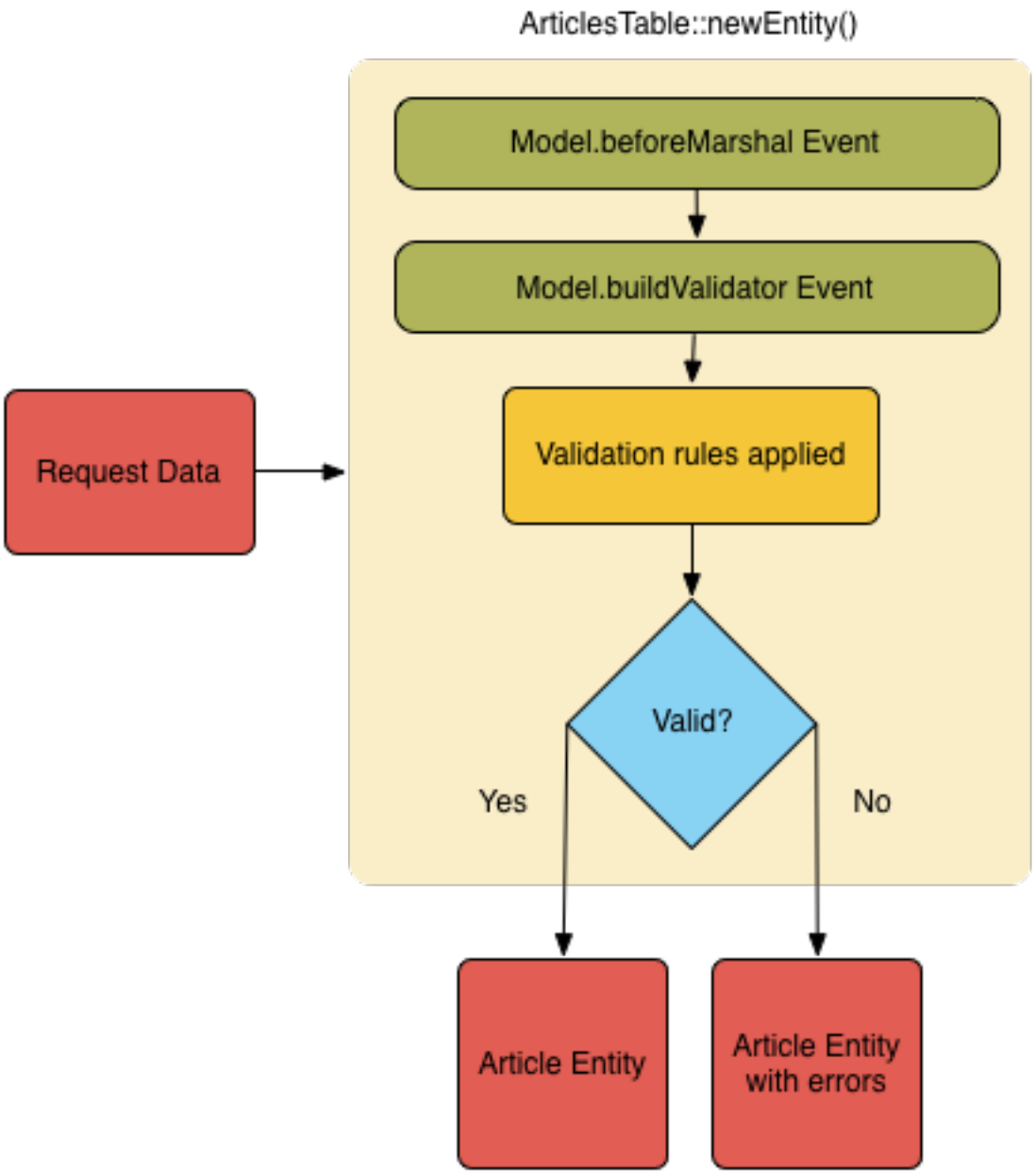
```
// In a controller
$articles = TableRegistry::get('Articles');

// 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' => [
        ['tag' => 'CakePHP'],
        ['tag' => '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 the new entity creation, by using the `onlyIds` option. When enabled, this option restricts belongsToMany marshalling to only use the `_ids` key and ignore all other data.

New in version 3.1.0: The `onlyIds` option was added in 3.1.0

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'],
        ['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.

New in version 3.1.0: The `onlyIds` option was added in 3.1.0

Converting Multiple Records

When creating forms that create/update multiple records at once you can use `newEntities()`:

```
// In a controller.
$articles = TableRegistry::get('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()` or `delete()` them:

```
// In a controller.
foreach ($entities as $entity) {
    // Save entity
    $articles->save($entity);

    // Delete entity
    $articles->delete($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 `transactional()`:

```
// In a controller.
$articles->getConnection()->transactional(function () use ($articles, $entities) {
    foreach ($entities as $entity) {
        $articles->save($entity, ['atomic' => false]);
    }
});
```

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 = TableRegistry::get('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 = TableRegistry::get('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 = TableRegistry::get('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'];
$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 $_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.
$comments = TableRegistry::get('Comments');
$present = (new Collection($entity->comments))->extract('id')->filter()->toArray();
$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 = TableRegistry::get('Articles');
$list = $articles->find('popular')->toArray();
$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\Event;
use ArrayObject;

// In a table or behavior class
public function beforeMarshal(Event $event, ArrayObject $data, ArrayObject $options)
{
    if (isset($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 whitelist. 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\Event;
use ArrayObject;

// In a table or behavior class
public function beforeMarshal(Event $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.

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 `fieldList` 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, [
    'fieldList' => ['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, [
    'fieldList' => ['title', 'tags'],
    'associated' => ['Tags' => ['fieldList' => ['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.

The `fieldList` options is also accepted by the `newEntity()`, `newEntities()` and `patchEntities()` methods.

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 = TableRegistry::get('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 = TableRegistry::get('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:

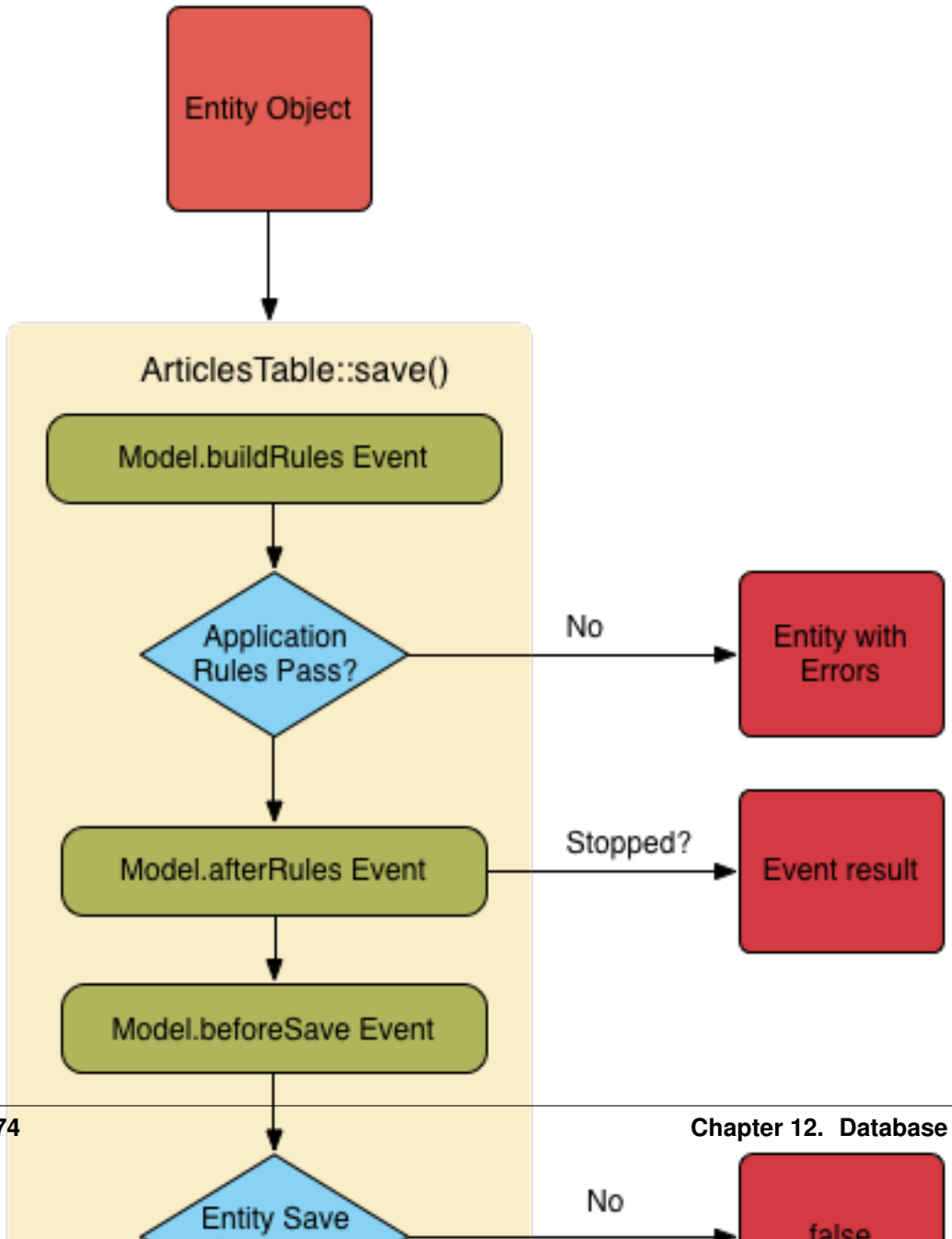
```
INSERT INTO articles (title) VALUES ('My new title');
```

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 are made to the entity when it is saved, the callbacks will not fire because no save is performed.

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:

Saving Associations

When you are saving an entity, you can also elect 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.
$companies->save($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 `dirty()`:

```
$company->author->name = 'Master Chef';
$company->dirty('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 = TableRegistry::get('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 = TableRegistry::get('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 = TableRegistry::get('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 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->comments[] = $comment;
$article->dirty('comments', true);
```

Without the call to `dirty()` the updated comments will not be saved.

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 = TableRegistry::get('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 easy to build 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->dirty('tags', true);
```

Without the call to `dirty()` 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 `_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:

```
// In config/bootstrap.php
use Cake\Database\Type;
Type::map('json', 'Cake\Database\Type\JsonType');

// In src/Model/Table/UsersTable.php
use Cake\Database\Schema\TableSchema;

class UsersTable extends Table
{

    protected function _initializeSchema(TableSchema $schema)
    {
        $schema->columnType('preferences', 'json');
        return $schema;
    }

}
```

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' => [
        '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 perfoms a *`Cake\ORM\Table::save()`* call, all corresponding save events will be triggered.

New in version 3.4.1.

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 = TableRegistry::get('Articles');
$entities = $articles->newEntities($data);
$result = $articles->saveMany($entities);
```

The result will be updated entities on success or `false` on failure.

New in version 3.2.8.

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:

```
// Publish all the unpublished articles.
function publishAllUnpublished()
{
    $this->updateAll(
        ['published' => true], // fields
        ['published' => false]); // conditions
}
```

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->query()
        ->update()
        ->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]);
```

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::`**`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.

Warning: deleteAll will *not* trigger beforeDelete/afterDelete events. If you need those first load a collection of records and delete them.

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 save, 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.

New in version 3.4.1.

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. CakePHP makes managing these associations easy. 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)
    {
        $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)
    {
        $this->belongsTo('Authors', [
                'className' => 'Publishing.Authors'
            ])
            ->setForeignKey('authorid')
            ->setProperty('person');
    }
}
```

You can also use arrays to customize your associations:

```
$this->belongsTo('Authors', [
    'className' => 'Publishing.Authors',
    'foreignKey' => 'authorid',
    'propertyName' => 'person'
]);
```

Arrays however do not offer the typehinting and autocomplete benefit 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)
    {
        $this->hasMany('Comments')
            ->setConditions(['approved' => true]);

        $this->hasMany('UnapprovedComments', [
                'className' => 'Comments'
            ])
            ->setConditions(['approved' => false])
            ->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)
    {
        $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)
    {
        $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 an 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. 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 use of any foreignKey in your associations definitions. Nevertheless sticking to conventions will make your code less repetitive, easier to read and to maintain.

If we had the `UsersTable` and `AddressesTable` classes made we could make the association with the following code:

```
class UsersTable extends Table
{
    public function initialize(array $config)
    {
        $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)
    {
        $this->hasOne('Addresses')
            ->setName('Addresses')
            ->setConditions(['Addresses.primary' => '1'])
            ->setDependent(true);
    }
}
```

Possible keys for hasOne association arrays include:

- **className**: the class name of the table being associated to the current model. If you're defining a 'User hasOne Address' relationship, the className key should equal 'Addresses'.
- **foreignKey**: the name of the foreign key found in the other table. This is especially handy if you need to define multiple hasOne relationships. The default value for this key is the underscored, singular name of the current model, suffixed with '_id'. In the example above it would default to 'user_id'.
- **bindingKey**: The name of the column in the current table, that will be used for matching the `foreignKey`. If not specified, the primary key (for example the id column of the `Users` table) will be used.
- **conditions**: an array of find() compatible conditions such as `['Addresses.primary' => true]`
- **joinType**: the type of the join to use in the SQL query, default is LEFT. You can use INNER if your hasOne association is always present.
- **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**: Defines the query strategy to use. Defaults to 'join'. The other valid value is 'select', which utilizes a separate query instead.
- **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 = $users->find('all')->contain(['Addresses']);
foreach ($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)
    {
        $this->belongsTo('Users');
    }
}
```

We can also define a more specific relationship using the setters:

```
class AddressesTable extends Table
{

    public function initialize(array $config)
    {
        $this->belongsTo('Users')
            ->setForeignKey('user_id')
            ->setJoinType('INNER');
    }
}
```

Possible keys for belongsTo association arrays include:

- **className**: the class name of the model being associated to the current model. If you're defining a 'Profile belongsTo User' relationship, the className key should equal 'Users'.
- **foreignKey**: the name of the foreign key found in the current table. This is especially handy if you need to define multiple belongsTo relationships to the same model. The default value for this key is the underscored, singular name of the other model, suffixed with `_id`.
- **bindingKey**: The name of the column in the other table, that will be used for matching the `foreignKey`. If not specified, the primary key (for example the id column of the `Users` table) will be used.
- **conditions**: an array of find() compatible conditions or SQL strings such as `['Users.active' => true]`
- **joinType**: the type of the join to use in the SQL query, default is LEFT which may not fit your needs in all situations, INNER may be helpful when you want everything from your main and associated models or nothing at all.
- **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**: Defines the query strategy to use. Defaults to 'join'. The other valid value is 'select', which utilizes a separate query instead.
- **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 = $addresses->find('all')->contain(['Users']);
foreach ($query as $address) {
    echo $address->user->username;
}
```

The above would emit SQL that is similar to:

```
SELECT * FROM addresses LEFT JOIN users ON addresses.user_id = users.id;
```

HasMany Associations

An example of a hasMany association is "Article 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
Article hasMany Comment	Comment.article_id
Product hasMany Option	Option.product_id
Doctor hasMany Patient	Patient.doctor_id

We can define the hasMany association in our Articles model as follows:

```
class ArticlesTable extends Table
{

    public function initialize(array $config)
    {
        $this->hasMany('Comments');
    }
}
```

We can also define a more specific relationship using the setters:

```
class ArticlesTable extends Table
{

    public function initialize(array $config)
    {
        $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('Reviews')
    ->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('Reviews')
    ->setForeignKey([
        'article_id',
        'article_hash'
    ])
    ->setBindingKey([
        'whatever_id',
        'whatever_hash'
    ]);
```

It is important to note that `foreignKey` values refers to the **reviews** table and `bindingKey` values refers to the **articles** table.

Possible keys for hasMany association arrays include:

- **className**: the class name of the model being associated to the current model. If you're defining a 'User hasMany Comment' relationship, the className key should equal 'Comments'.
- **foreignKey**: the name of the foreign key found in the other table. This is especially handy if you need to define multiple hasMany relationships. The default value for this key is the underscored, singular name of the actual model, suffixed with '_id'.
- **bindingKey**: The name of the column in the current table, that will be used for matching the `foreignKey`. If not specified, the primary key (for example the id column of the `Articles` table) will be used.
- **conditions**: an array of find() compatible conditions or SQL strings such as `['Comments.visible' => true]`
- **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'. 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 null able column or if `dependent` is true records will be orphaned.
- **finder**: The finder method to use when loading associated records.

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 = $articles->find('all')->contain(['Comments']);
foreach ($query as $article) {
```

```
    echo $article->comments[0]->text;
}
```

The above would emit SQL that is 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

Note: In 3.0 and onward `hasAndBelongsToMany` / `HABTM` has been renamed to `belongsToMany` / `BTM`.

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 are 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
Article belongsToMany Tag	articles_tags.id, articles_tags.tag_id, articles_tags.article_id
Patient belongsToMany Doctor	doctors_patients.id, doctors_patients.doctor_id, doctors_patients.patient_id.

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)
    {
        $this->belongsToMany('Tags');
    }
}
```

```
// In src/Model/Table/TagsTable.php
class TagsTable extends Table
{

    public function initialize(array $config)
    {
        $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)
    {
        $this->belongsToMany('Articles', [
            'joinTable' => 'articles_tags',
        ]);
    }
}
```

Possible keys for belongsToMany association arrays include:

- **className**: the class name of the model being associated to the current model. If you're defining a 'Article belongsToMany Tag' relationship, the className key should equal 'Tags'.
- **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.
- **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 name 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.

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 = $articles->find('all')->contain(['Tags']);
foreach ($query as $article) {
    echo $article->tags[0]->text;
}
```

The above would emit SQL that is 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)
    {
        $this->belongsToMany('Courses', [
            'through' => 'CoursesMemberships',
        ]);
    }
}

class CoursesTable extends Table
{
    public function initialize(array $config)
    {
        $this->belongsToMany('Students', [
            'through' => 'CoursesMemberships',
        ]);
    }
}

class CoursesMembershipsTable extends Table
{
    public function initialize(array $config)
    {
        $this->belongsTo('Students');
        $this->belongsTo('Courses');
    }
}
```

The CoursesMemberships join table uniquely identifies a given Student's participation on a Course in addition to extra meta-information.

Default Association Conditions

The `finder` option allows you to use a *custom finder* to load associated record data. This lets you encapsulate your queries better and keep your code DRY'er. 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.

Loading Associations

Once you've defined your associations you can *eager load associations* when fetching results.

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 an easy 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)
    {
        $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)
    {
        $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)
    {
        $this->addBehavior('CounterCache', [
            'Articles' => ['comment_count']
        ]);
    }
}
```

The CounterCache configuration should be a map of relation names and the specific configuration for that relation.

The counter's value will be updated each time an entity is saved or deleted. The counter **will not** be updated when you use `updateAll()` or `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 method. This callable must return the count value to be stored:

```
$this->addBehavior('CounterCache', [
    'Articles' => [
        'rating_avg' => function ($event, $entity, $table) {
            return 4.5;
        }
    ]
]);
```

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.

It is possible though to make this work for `belongsToMany` associations. You need to enable the CounterCache behavior in a custom `through` table configured in association options. See how to configure a custom join table *Using the 'through' Option*.

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)
    {
        $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)
    {
        $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->dirty('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. It does so by using a separate `i18n` table where it stores the translation for each of the fields of any given Table object that it's bound to.

> **Warning:** The TranslateBehavior does not support composite primary keys at this point in time.

A Quick Tour

After creating the `i18n` table in your database attach the behavior to any Table object you want to make translatable:

```
class ArticlesTable extends Table
{
    public function initialize(array $config)
    {
        $this->addBehavior('Translate', ['fields' => ['title']]);
    }
}
```

Now, select a language to be used for retrieving entities by changing the application language, which will affect all translations:

```
// In a controller. Change the locale, e.g. to Spanish
I18n::setLocale('es');
$this->loadModel('Articles');
```

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';
```

It is equally easy to 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.

Initializing the i18n Database Table

In order to use the behavior, 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**.

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[130]).

[130] https://en.wikipedia.org/wiki/UN_M.49

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 Behaviour` 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.

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)
    {
        $this->addBehavior('Translate', ['fields' => ['title', 'body']]);
    }
}
```

The first thing to note is that 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.

Using a Separate Translations Table

If you wish to use a table other than `i18n` for translating a particular repository, you can specify it 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)
    {
        $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 `locale()` method to choose the active translation for entities that are loaded:

```
// Load I18n core functions at the beginning of your Controller:
use Cake\I18n\I18n;

// Then you can change the language in your action:
I18n::setLocale('es');
```

```
$this->loadModel('Articles');

// 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' ...]
```

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)
    {
        $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::locale

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 `setLocale()` method:

```
I18n::setLocale('en'); // reset for illustration

$this->loadModel('Articles');
$this->Articles->locale('es'); // specific locale

$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 langauge of associated data. To affect associated data it's necessary to call locale on each table for example:

```
I18n::setLocale('en'); // reset for illustration

$this->loadModel('Articles');
$this->Articles->locale('es');
$this->Articles->Categories->locale('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->locale('es');
$data = $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)
    {
        $this->addBehavior('Translate', ['fields' => ['title', 'body']]);
    }
}

// in src/Model/Entity/Article.php
class Article extends Entity
{
    use TranslateTrait;
}

// In a Controller
$articles = $this->loadModel('Articles');
```

```
$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:

```
I18n::setLocale('es');
$article->title = 'Mi Primer Artículo';
$this->Articles->save($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]);
}

$this->Articles->save($article);
```

As of 3.3.0, working with multiple translations has been streamlined. You can 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, but with the `translations` option enabled:

```
$article = $this->Articles->newEntity($this->request->getData(), [
    'translations' => true
]);
$this->Articles->save($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)
    {
        $this->addBehavior('Translate', [
            'fields' => ['title'],
            'validator' => 'translated'
        ]);
    }
}
```

The above will use the validator created by `validationTranslated` to validated translated entities.

New in version 3.3.0: Validating translated entities, and streamlined translation saving was added in 3.3.0

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
- `lft` (integer, signed) Used to maintain the tree structure
- `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[131]

> **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)
    {
        $this->addBehavior('Tree');
    }
}
```

Once added, you can let CakePHP build the internal structure if the table is already holding some rows:

```
$categories = TableRegistry::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);
```

Getting a flat list of the descendants for a node is equally easy:

[131] http://www.sitepoint.com/hierarchical-data-database-2/

```
$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%']);

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";
}
```

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');

// 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' => ' '
]);
```

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]);

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);

// 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)
{
    $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 upto a certain level for e.g. 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)
    {
        $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->config('scope', ['country_name' => 'France']);
```

Optionally, you can have a finer grain control of the scope by passing a closure as the scope:

```
$this->behaviors()->Tree->config('scope', function ($query) {
    $country = $this->getConfigureContry(); // A made-up function
    return $query->where(['country_name' => $country]);
});
```

Recovering with custom sort field

New in version 3.0.14.

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 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]);

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
    }
}
```

The TreeBehavior reorders the lft and rght values of records in the table when a node is deleted. As such, the lft and rght values of the entities inside `$descendants` (saved before the delete operation) will be inaccurate. Entities will have to be loaded and modified on the fly to prevent inconsistencies in the table.

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;

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)
{
    // 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)
    {
        $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\Event;
use Cake\ORM\Behavior;
use Cake\ORM\Entity;
use Cake\ORM\Query;
use Cake\Utility\Text;
```

```
class SluggableBehavior extends Behavior
{
    protected $_defaultConfig = [
        'field' => 'title',
        'slug' => 'slug',
        'replacement' => '-',
    ];

    public function slug(Entity $entity)
    {
        $config = $this->config();
        $value = $entity->get($config['field']);
        $entity->set($config['slug'], Text::slug($value, $config['replacement']));
    }

    public function beforeSave(Event $event, EntityInterface $entity, ArrayObject
↪$options)
    {
        $this->slug($entity);
    }

}
```

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 saving from continuing simply stop event propagation in your callback:

```
public function beforeSave(Event $event, EntityInterface $entity, ArrayObject
↪$options)
{
    if (...) {
        $event->stopPropagation();
        return;
    }
    $this->slug($entity);
}
```

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(Query $query, array $options)
{
    return $query->where(['slug' => $options['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 $_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.

New in version 3.3.0: The ability for behaviors to participate in marshalling was added in 3.3.0

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 (e.g. an `AppTable`) class 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 $options)
    {
        parent::initialize($options);

        // e.g. if our parent calls $this->addBehavior('Timestamp');
        // and we want to add an additional event
        if ($this->behaviors()->has('Timestamp')) {
            $this->behaviors()->get('Timestamp')->config([
                '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')
->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 contraints. 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->schemaCollection();

// Get the table names
$tables = $collection->listTables();

// Get a single table (instance of Schema\TableSchema)
$tableSchema = $collection->describe('posts');
```

ORM Cache Shell

The OrmCacheShell 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 orm_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 orm_cache build --connection default articles
```

In addition to building cached data, you can use the OrmCacheShell to remove cached metadata as well:

```
# Clear all metadata
bin/cake orm_cache clear
```

```
# Clear a single table
bin/cake orm_cache clear articles
```

CHAPTER 13

Bake Console

CakePHP's bake console is another effort to get you up and running in CakePHP – fast. The bake console can create any of CakePHP's basic ingredients: models, behaviors, views, helpers, controllers, components, test cases, fixtures and plugins. And we aren't just talking skeleton classes: Bake can create a fully functional application in just a few minutes. In fact, Bake is a natural step to take once an application has been scaffolded.

Installation

Before trying to use or extend bake, make sure it is installed in your application. Bake is provided as a plugin that you can install with Composer:

```
composer require --dev cakephp/bake:~1.0
```

The above will install bake as a development dependency. This means that it will not be installed when you do production deployments. The following sections cover bake in more detail:

Code Generation with Bake

The cake console is run using the PHP CLI (command line interface). If you have problems running the script, ensure that:

1. You have the PHP CLI installed and that it has the proper modules enabled (eg: MySQL, intl).
2. Users also might have issues if the database host is 'localhost' and should try '127.0.0.1' instead, as localhost can cause issues with PHP CLI.
3. Depending on how your computer is configured, you may have to set execute rights on the cake bash script to call it using `bin/cake bake`.

Before running bake you should make sure you have at least one database connection configured. See the section on *database configuration* for more information.

When run with no arguments `bin/cake bake` will output a list of available tasks. You should see something like:

```
$ bin/cake bake

Welcome to CakePHP v3.4.6 Console
---------------------------------------------------------------
```

```
App : src
Path: /var/www/cakephp.dev/src/
PHP : 5.6.20
---------------------------------------------------------------
The following commands can be used to generate skeleton code for your application.

Available bake commands:

- all
- behavior
- cell
- component
- controller
- fixture
- form
- helper
- mailer
- migration
- migration_diff
- migration_snapshot
- model
- plugin
- seed
- shell
- shell_helper
- task
- template
- test

By using `cake bake [name]` you can invoke a specific bake task.
```

You can get more information on what each task does, and what options are available using the `--help` option:

```
$ bin/cake bake --help

Welcome to CakePHP v3.4.6 Console
---------------------------------------------------------------
App : src
Path: /var/www/cakephp.dev/src/
PHP : 5.6.20
---------------------------------------------------------------
The Bake script generates controllers, models and template files for
your application. If run with no command line arguments, Bake guides the
user through the class creation process. You can customize the
generation process by telling Bake where different parts of your
application are using command line arguments.

Usage:
cake bake.bake [subcommand] [options]

Subcommands:

all                 Bake a complete MVC skeleton.
behavior            Bake a behavior class file.
cell                Bake a cell class file.
component           Bake a component class file.
controller          Bake a controller skeleton.
fixture             Generate fixtures for use with the test suite. You
```

```
                    can use `bake fixture all` to bake all fixtures.
form                Bake a form class file.
helper              Bake a helper class file.
mailer              Bake a mailer class file.
migration           Bake migration class.
migration_diff      Bake migration class.
migration_snapshot  Bake migration snapshot class.
model               Bake table and entity classes.
plugin              Create the directory structure, AppController class
                    and testing setup for a new plugin. Can create
                    plugins in any of your bootstrapped plugin paths.
seed                Bake seed class.
shell               Bake a shell class file.
shell_helper        Bake a shell_helper class file.
task                Bake a task class file.
template            Bake views for a controller, using built-in or
                    custom templates.
test                Bake test case skeletons for classes.

To see help on a subcommand use `cake bake.bake [subcommand] --help`

Options:

--connection, -c   Database connection to use in conjunction with `bake
                   all`. (default: default)
--everything       Bake a complete MVC skeleton, using all the available
                   tables. Usage: "bake all --everything"
--force, -f        Force overwriting existing files without prompting.
--help, -h         Display this help.
--plugin, -p       Plugin to bake into.
--prefix           Prefix to bake controllers and templates into.
--quiet, -q        Enable quiet output.
--tablePrefix      Table prefix to be used in models.
--theme, -t        The theme to use when baking code. (choices:
                   Bake|Migrations)
--verbose, -v      Enable verbose output.
```

Bake Themes

The theme option is common to all bake commands, and allows changing the bake template files used when baking. To create your own templates, see the *bake theme creation documentation*.

Extending Bake

Bake features an extensible architecture that allows your application or plugins to modify or add-to the base functionality. Bake makes use of a dedicated view class which does not use standard PHP syntax.

Bake Events

As a view class, `BakeView` emits the same events as any other view class, plus one extra initialize event. However, whereas standard view classes use the event prefix "View.", `BakeView` uses the event prefix "Bake.".

The initialize event can be used to make changes which apply to all baked output, for example to add another helper to the bake view class this event can be used:

```php
<?php
// config/bootstrap_cli.php

use Cake\Event\Event;
use Cake\Event\EventManager;

EventManager::instance()->on('Bake.initialize', function (Event $event) {
    $view = $event->getSubject();

    // In my bake templates, allow the use of the MySpecial helper
    $view->loadHelper('MySpecial', ['some' => 'config']);

    // And add an $author variable so it's always available
    $view->set('author', 'Andy');

});
```

If you want to modify bake from within another plugin, putting your plugin's bake events in the plugin `config/bootstrap.php` file is a good idea.

Bake events can be handy for making small changes to existing templates. For example, to change the variable names used when baking controller/template files one can use a function listening for `Bake.beforeRender` to modify the variables used in the bake templates:

```php
<?php
// config/bootstrap_cli.php

use Cake\Event\Event;
use Cake\Event\EventManager;

EventManager::instance()->on('Bake.beforeRender', function (Event $event) {
    $view = $event->getSubject();

    // Use $rows for the main data variable in indexes
    if ($view->get('pluralName')) {
        $view->set('pluralName', 'rows');
    }
    if ($view->get('pluralVar')) {
        $view->set('pluralVar', 'rows');
    }

    // Use $theOne for the main data variable in view/edit
    if ($view->get('singularName')) {
        $view->set('singularName', 'theOne');
    }
    if ($view->get('singularVar')) {
        $view->set('singularVar', 'theOne');
    }

});
```

You may also scope the `Bake.beforeRender` and `Bake.afterRender` events to a specific generated file. For instance, if you want to add specific actions to your UsersController when generating from a **Controller/controller.ctp** file, you can use the following event:

```php
<?php
// config/bootstrap_cli.php
```

```
use Cake\Event\Event;
use Cake\Event\EventManager;
use Cake\Utility\Hash;

EventManager::instance()->on(
    'Bake.beforeRender.Controller.controller',
    function (Event $event) {
        $view = $event->getSubject();
        if ($view->viewVars['name'] == 'Users') {
            // add the login and logout actions to the Users controller
            $view->viewVars['actions'] = [
                'login',
                'logout',
                'index',
                'view',
                'add',
                'edit',
                'delete'
            ];
        }
    }
);
```

By scoping event listeners to specific bake templates, you can simplify your bake related event logic and provide callbacks that are easier to test.

Bake Template Syntax

Bake template files use erb-style (`<% %>`) tags to denote template logic, and treat everything else including php tags as plain text.

Note: Bake template files do not use, and are insensitive to, `asp_tags` php ini setting.

`BakeView` implements the following tags:

- `<%` A Bake template php open tag
- `%>` A Bake template php close tag
- `<%=` A Bake template php short-echo tag
- `<%-` A Bake template php open tag, stripping any leading whitespace before the tag
- `-%>` A Bake template php close tag, stripping trailing whitespace after the tag

One way to see/understand how bake templates works, especially when attempting to modify bake template files, is to bake a class and compare the template used with the pre-processed template file which is left in the application's **tmp/bake** folder.

So, for example, when baking a shell like so:

```
bin/cake bake shell Foo
```

The template used (**vendor/cakephp/bake/src/Template/Bake/Shell/shell.ctp**) looks like this:

```
<?php
namespace <%= $namespace %>\Shell;
```

```
use Cake\Console\Shell;

/**
 * <%= $name %> shell command.
 */
class <%= $name %>Shell extends Shell
{

    /**
     * main() method.
     *
     * @return bool|int Success or error code.
     */
    public function main()
    {
    }

}
```

The pre-processed template file (**tmp/bake/Bake-Shell-shell-ctp.php**), which is the file actually rendered, looks like this:

```
<CakePHPBakeOpenTagphp
namespace <?= $namespace ?>\Shell;

use Cake\Console\Shell;

/**
 * <?= $name ?> shell command.
 */
class <?= $name ?>Shell extends Shell
{

    /**
     * main() method.
     *
     * @return bool|int Success or error code.
     */
    public function main()
    {
    }

}
```

And the resultant baked class (**src/Shell/FooShell.php**) looks like this:

```
<?php
namespace App\Shell;

use Cake\Console\Shell;

/**
 * Foo shell command.
 */
class FooShell extends Shell
{
```

```
    /**
     * main() method.
     *
     * @return bool|int Success or error code.
     */
    public function main()
    {
    }

}
```

Creating a Bake Theme

If you wish to modify the output produced by the "bake" command, you can create your own bake 'theme' which allows you to replace some or all of the templates that bake uses. The best way to do this is:

1. Bake a new plugin. The name of the plugin is the bake 'theme' name
2. Create a new directory **plugins/[name]/src/Template/Bake/Template/**.
3. Copy any templates you want to override from **vendor/cakephp/bake/src/Template/Bake/Template** to matching files in your plugin.
4. When running bake use the `--theme` option to specify the bake-theme you want to use.

Customizing the Bake Templates

If you wish to modify the default output produced by the "bake" command, you can create your own bake templates in your application. This way does not use the `--theme` option in the command line when baking. The best way to do this is:

1. Create a new directory **/src/Template/Bake/**.
2. Copy any templates you want to override from **vendor/cakephp/bake/src/Template/Bake/** to matching files in your application.

Creating New Bake Command Options

It's possible to add new bake command options, or override the ones provided by CakePHP by creating tasks in your application or plugins. By extending `Bake\Shell\Task\BakeTask`, bake will find your new task and include it as part of bake.

As an example, we'll make a task that creates an arbitrary foo class. First, create the task file **src/Shell/Task/FooTask.php**. We'll extend the `SimpleBakeTask` for now as our shell task will be simple. `SimpleBakeTask` is abstract and requires us to define 3 methods that tell bake what the task is called, where the files it generates should go, and what template to use. Our FooTask.php file should look like:

```
<?php
namespace App\Shell\Task;

use Bake\Shell\Task\SimpleBakeTask;

class FooTask extends SimpleBakeTask
{
    public $pathFragment = 'Foo/';
```

```
    public function name()
    {
        return 'foo';
    }

    public function fileName($name)
    {
        return $name . 'Foo.php';
    }

    public function template()
    {
        return 'foo';
    }

}
```

Once this file has been created, we need to create a template that bake can use when generating code. Create **src/Template/Bake/foo.ctp**. In this file we'll add the following content:

```
<?php
namespace <%= $namespace %>\Foo;

/**
 * <%= $name %> foo
 */
class <%= $name %>Foo
{
    // Add code.
}
```

You should now see your new task in the output of `bin/cake bake`. You can run your new task by running `bin/cake bake foo Example`. This will generate a new `ExampleFoo` class in **src/Foo/ExampleFoo.php** for your application to use.

If you want the `bake` call to also create a test file for your `ExampleFoo` class, you need to overwrite the `bakeTest()` method in the `FooTask` class to register the class suffix and namespace for your custom command name:

```
public function bakeTest($className)
{
    if (!isset($this->Test->classSuffixes[$this->name()])) {
      $this->Test->classSuffixes[$this->name()] = 'Foo';
    }

    $name = ucfirst($this->name());
    if (!isset($this->Test->classTypes[$name])) {
      $this->Test->classTypes[$name] = 'Foo';
    }

    return parent::bakeTest($className);
}
```

- The **class suffix** will be appened to the name provided in your `bake` call. In the previous example, it would create a `ExampleFooTest.php` file.
- The **class type** will be the sub-namespace used that will lead to your file (relative to the app or the plugin you are baking into). In the previous example, it would create your test with the namespace

```
App\Test\TestCase\Foo.
```

CHAPTER 14

Caching

class `Cake\Cache\`**`Cache`**

Caching is frequently used to reduce the time it takes to create or read from other resources. Caching is often used to make reading from expensive resources less expensive. You can store the results of expensive queries, or remote webservice access that doesn't frequently change in a cache. Once in the cache, re-reading the stored resource from the cache is much cheaper than accessing the remote resource.

Caching in CakePHP is primarily facilitated by the `Cache` class. This class provides a set of static methods that provide a uniform API to dealing with all different types of Caching implementations. CakePHP comes with several cache engines built-in, and provides an easy system to implement your own caching systems. The built-in caching engines are:

- `FileCache` 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.
- `ApcCache` APC 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.
- `Wincache` Wincache uses the Wincache[133] extension. Wincache is similar to APC in features and performance, but optimized for Windows and IIS.
- `XcacheEngine` Xcache[134] is a PHP extension that provides similar features to APC.
- `MemcachedEngine` Uses the Memcached[135] extension.
- `RedisEngine` Uses the phpredis[136] extension. Redis provides a fast and persistent cache system similar to Memcached, also provides atomic operations.

Regardless of the CacheEngine you choose to use, your application interacts with *`Cake\Cache\Cache`* in a consistent manner. You can swap cache engines as your application grows.

[132] http://php.net/apcu
[133] http://php.net/wincache
[134] http://xcache.lighttpd.net/
[135] http://php.net/memcached
[136] https://github.com/nicolasff/phpredis

Configuring Cache Class

static `Cake\Cache\Cache::config`(*$key*, *$config* = *null*)

Configuring the Cache class can be done anywhere, but generally you will want to configure Cache during bootstrapping. The **config/app.php** file is the conventional location to do this. You can configure as many cache configurations as you need, and use any mixture of cache engines. CakePHP uses two cache configurations internally. `_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. If you are using APC or Memcached you should make sure to set unique keys for the core caches. This will prevent multiple applications from overwriting each other's cached data.

Using multiple 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::config('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::config('short', [
    'className' => 'File',
    'duration' => '+1 hours',
    'path' => CACHE,
    'prefix' => 'cake_short_'
]);

// Using a fully namespaced name.
Cache::config('long', [
    'className' => 'Cake\Cache\Engine\FileEngine',
    'duration' => '+1 week',
    'probability' => 100,
    'path' => CACHE . 'long' . DS,
```

```
]);

// Using a constructed object.
$object = new FileEngine($config);
Cache::config('other', $object);
```

The name of these configurations 'short' or 'long' is 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 classname without 'Engine' or a namespace. This will infer that you want to use a Cache engine in `Cake\Cache\Engine` or `App\Cache\Engine`.
- Using *plugin syntax* which allows you to load engines from a specific plugin.
- Using a fully qualified namespaced classname. This allows you to use classes located outside of the conventional locations.
- Using an object that extends the `CacheEngine` class.

Note: When using the FileEngine you might need to use the `mask` option to ensure cache files are made with the correct permissions.

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::config('redis', [
    'className' => 'Redis',
    'duration' => '+1 hours',
    'prefix' => 'cake_redis_',
    'host' => '127.0.0.1',
    'port' => 6379,
    'fallback' => 'default',
]);
```

If the Redis server unexpectedly failed, writing to the `redis` cache configuration would fall back to writing to the `default` cache configuration. If writing to the `default` cache configuration *also* failed in this scenario, the engine would fall back once again to the `NullEngine` and prevent the application from throwing an uncaught exception.

New in version 3.5.0: Cache engine fallbacks were added.

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::config()*. 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:

```
if (($posts = Cache::read('posts')) === false) {
    $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 API's 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]
```

Read Through Caching

static Cake\Cache\Cache::**remember**(*$key*, *$callable*, *$config = 'default'*)

Cache makes it easy to do 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 false if the cache has expired or doesn't exist. The contents of the cache might evaluate false, so make sure you use the strict comparison operators: === or !==.

For example:

```
$cloud = Cache::read('cloud');

if ($cloud !== false) {
    return $cloud;
}

// Generate cloud data
// ...

// Store data in cache
Cache::write('cloud', $cloud);
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 API's 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');
```

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 API's 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`(*$check*, *$config = 'default'*)

Destroy all cached values for a cache configuration. In engines like: Apc, Memcached, and Wincache, the cache configuration's prefix is used to remove cache entries. Make sure that different cache configurations have different prefixes:

```
// Will only clear expired keys.
Cache::clear(true);

// Will clear all keys.
Cache::clear(false);
```

static `Cake\Cache\Cache::gc`(*$config*)

Garbage collects entries in the cache configuration. This is primarily used by FileEngine. It should be implemented by any Cache engine that requires manual eviction of cached data.

Note: Because APC and Wincache use 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 in an easy way. 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 APC, Wincache, 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::config('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);
        }
    }
}
```

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 Storage Engine for Cache

You can provide custom `Cache` adapters in `App\Cache\Engine` as well as in plugins using `$plugin\Cache\Engine`. src/plugin cache engines can also override the core engines. Cache adapters 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** as an app/libs. Or in **plugin/Cache/Engine/MyCustomCacheEngine.php** as part of a plugin. Cache configs from plugins need to use the plugin dot syntax.

```
Cache::config('custom', [
    'className' => 'CachePack.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*, *$config = 'default'*)

Returns boolean for success.

Write value for a key into cache, optional string $config specifies configuration name to write to.

Cake\Cache\CacheEngine::**read**(*$key*)

Returns The cached value or `false` for failure.

Read a key from the cache. Return `false` 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

Cake\Cache\CacheEngine::**gc**()

Not required, but used to do clean up when resources expire. FileEngine uses this to delete files containing expired content.

CHAPTER 15

Console Tools, Shells & Tasks

CakePHP features not only a web framework but also a console framework for creating console applications. Console applications are ideal for handling a variety of background tasks such as maintenance, and completing work outside of the request-response cycle. CakePHP console applications allow you to reuse your application classes from the command line.

CakePHP comes with a number of console applications out of the box. Some of these applications are used in concert with other CakePHP features (like i18n), and others are for general use to get you working faster.

The CakePHP Console

This section provides an introduction into CakePHP at the command-line. Console tools are ideal for use in cron jobs, or command line based utilities that don't need to be accessible from a web browser.

PHP provides a CLI client that makes interfacing with your file system and applications much smoother. The CakePHP console provides a framework for creating shell scripts. The Console uses a dispatcher-type setup to load a shell or task, and provide its parameters.

Note: A command-line (CLI) build of PHP must be available on the system if you plan to use the Console.

Before we get into specifics, let's make sure you can run the CakePHP console. First, you'll need to bring up a system shell. The examples shown in this section will be in bash, but the CakePHP Console is Windows-compatible as well. This example assumes that the user is currently logged into a bash prompt and is currently at the root of a CakePHP application.

A CakePHP application contains **src/Shell** and **src/Shell/Task** directories that contain all of its shells and tasks. 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 produces this help message:

```
Welcome to CakePHP v3.5.0 Console
---------------------------------------------------------------
App : App
Path: /Users/markstory/Sites/cakephp-app/src/
---------------------------------------------------------------
Current Paths:

 -app: src
 -root: /Users/markstory/Sites/cakephp-app
 -core: /Users/markstory/Sites/cakephp-app/vendor/cakephp/cakephp

Changing Paths:

Your working path should be the same as your application path. To change your path␣
 ↪use the '-app' param.
Example: -app relative/path/to/myapp or -app /absolute/path/to/myapp

Available Shells:

- version
- help
- cache
- completion
- i18n
- orm_cache
- plugin
- routes
- server
- bug
- console
- event
- orm
- bake
- bake.bake
- migrations
- migrations.migrations

To run a command, type `cake shell_name [args|options]`
To get help on a specific command, type `cake shell_name --help`
```

The first information printed relates to paths. This is helpful if you're running the console from different parts of the filesystem.

You could then run the any of the listed shells by using its name:

```
# run server shell
bin/cake server

# run migrations shell
bin/cake migrations -h

# run bake (with plugin prefix)
bin/cake bake.bake -h
```

Plugin shells can be invoked without a plugin prefix if the shell's name does not overlap with an application or framework shell. In the case that two plugins provide a shell with the same name, the first loaded plugin will get the short alias. You can always use the `plugin.shell` format to unambiguously reference a shell.

class Cake\Console**Shell**

Creating a Shell

Let's create a shell for use in the Console. For this example, we'll create a simple Hello world shell. In your application's **src/Shell** directory create **HelloShell.php**. Put the following code inside it:

```
namespace App\Shell;

use Cake\Console\Shell;

class HelloShell extends Shell
{
    public function main()
    {
        $this->out('Hello world.');
    }
}
```

The conventions for shell classes are that the class name should match the file name, with the suffix of Shell. In our shell we created a `main()` method. This method is called when a shell is called with no additional commands. We'll add some more commands in a bit, but for now let's just run our shell. From your application directory, run:

```
bin/cake hello
```

You should see the following output:

```
Hello world.
```

As mentioned before, the `main()` method in shells is a special method called whenever there are no other commands or arguments given to a shell. Since our main method wasn't very interesting let's add another command that does something:

```
namespace App\Shell;

use Cake\Console\Shell;

class HelloShell extends Shell
{
    public function main()
    {
        $this->out('Hello world.');
    }

    public function heyThere($name = 'Anonymous')
    {
        $this->out('Hey there ' . $name);
    }
}
```

After saving this file, you should be able to run the following command and see your name printed out:

```
bin/cake hello hey_there your-name
```

Any public method not prefixed by an _ is allowed to be called from the command line. As you can see, methods invoked from the command line are transformed from the underscored shell argument to the correct camel-cased

method name in the class.

In our `heyThere()` method we can see that positional arguments are provided to our `heyThere()` function. Positional arguments are also available in the `args` property. You can access switches or options on shell applications, which are available at `$this->params`, but we'll cover that in a bit.

When using a `main()` method you won't be able to use the positional arguments. This is because the first positional argument or option is interpreted as the command name. If you want to use arguments, you should use method names other than `main`.

Shell Tasks

There will be times when building more advanced console applications, you'll want to compose functionality into re-usable classes that can be shared across many shells. Tasks allow you to extract commands into classes. For example the `bake` command is made almost entirely of tasks. You define a tasks for a shell using the `$tasks` property:

```
class UserShell extends Shell
{
    public $tasks = ['Template'];
}
```

You can use tasks from plugins using the standard *plugin syntax*. Tasks are stored in `Shell/Task/` in files named after their classes. So if we were to create a new 'FileGenerator' task, you would create **src/Shell/Task/FileGeneratorTask.php**.

Each task must at least implement a `main()` method. The ShellDispatcher, will call this method when the task is invoked. A task class looks like:

```
namespace App\Shell\Task;

use Cake\Console\Shell;

class FileGeneratorTask extends Shell
{
    public function main()
    {

    }
}
```

A shell can also access its tasks as properties, which makes tasks great for making re-usable chunks of functionality similar to *Components*:

```
// Found in src/Shell/SeaShell.php
class SeaShell extends Shell
{
    // Found in src/Shell/Task/SoundTask.php
    public $tasks = ['Sound'];

    public function main()
    {
        $this->Sound->main();
    }
}
```

You can also access tasks directly from the command line:

```
$ cake sea sound
```

Note: In order to access tasks directly from the command line, the task **must** be included in the shell class' $tasks property.

Also, the task name must be added as a sub-command to the Shell's OptionParser:

```
public function getOptionParser()
{
    $parser = parent::getOptionParser();
    $parser->addSubcommand('sound', [
        // Provide help text for the command list
        'help' => 'Execute The Sound Task.',
        // Link the option parsers together.
        'parser' => $this->Sound->getOptionParser(),
    ]);
    return $parser;
}
```

Loading Tasks On The Fly with TaskRegistry

You can load tasks on the fly using the Task registry object. You can load tasks that were not declared in $tasks this way:

```
$project = $this->Tasks->load('Project');
```

Would load and return a ProjectTask instance. You can load tasks from plugins using:

```
$progressBar = $this->Tasks->load('ProgressBar.ProgressBar');
```

Using Models in Your Shells

You'll often need access to your application's business logic in shell utilities; CakePHP makes that super easy. You can load models in shells, just as you would in a controller using `loadModel()`. The loaded models are set as properties attached to your shell:

```
namespace App\Shell;

use Cake\Console\Shell;

class UserShell extends Shell
{

    public function initialize()
    {
        parent::initialize();
        $this->loadModel('Users');
    }

    public function show()
    {
```

```
        if (empty($this->args[0])) {
            // Use error() before CakePHP 3.2
            return $this->abort('Please enter a username.');
        }
        $user = $this->Users->findByUsername($this->args[0])->first();
        $this->out(print_r($user, true));
    }
}
```

The above shell, will fetch a user by username and display the information stored in the database.

Shell Helpers

If you have complex output generation logic, you can use *Shell Helpers* to encapsulate this logic in a re-usable way.

Invoking Other Shells from Your Shell

`Cake\Console\Shell::dispatchShell`(*$args*)

There are still many cases where you will want to invoke one shell from another though. `Shell::dispatchShell()` gives you the ability to call other shells by providing the `argv` for the sub shell. You can provide arguments and options either as var args or as a string:

```
// As a string
$this->dispatchShell('schema create Blog --plugin Blog');

// As an array
$this->dispatchShell('schema', 'create', 'Blog', '--plugin', 'Blog');
```

The above shows how you can call the schema shell to create the schema for a plugin from inside your plugin's shell.

Passing extra parameters to the dispatched Shell

New in version 3.1.

It can sometimes be useful to pass on extra parameters (that are not shell arguments) to the dispatched Shell. In order to do this, you can now pass an array to `dispatchShell()`. The array is expected to have a `command` key as well as an `extra` key:

```
// Using a command string
$this->dispatchShell([
   'command' => 'schema create Blog --plugin Blog',
   'extra' => [
        'foo' => 'bar'
    ]
]);

// Using a command array
$this->dispatchShell([
   'command' => ['schema', 'create', 'Blog', '--plugin', 'Blog'],
   'extra' => [
        'foo' => 'bar'
```

```
    ]
]);
```

Parameters passed through `extra` will be merged in the `Shell::$params` property and are accessible with the `Shell::param()` method. By default, a `requested` extra param is automatically added when a Shell is dispatched using `dispatchShell()`. This `requested` parameter prevents the CakePHP console welcome message from being displayed on dispatched shells.

Getting User Input

`Cake\Console\Shell::in`(*$question*, *$choices* = *null*, *$default* = *null*)

When building interactive console applications you'll need to get user input. CakePHP provides an easy way to do this:

```
// Get arbitrary text from the user.
$color = $this->in('What color do you like?');

// Get a choice from the user.
$selection = $this->in('Red or Green?', ['R', 'G'], 'R');
```

Selection validation is case-insensitive.

Creating Files

`Cake\Console\Shell::createFile`(*$path*, *$contents*)

Many Shell applications help automate development or deployment tasks. Creating files is often important in these use cases. CakePHP provides an easy way to create a file at a given path:

```
$this->createFile('bower.json', $stuff);
```

If the Shell is interactive, a warning will be generated, and the user asked if they want to overwrite the file if it already exists. If the shell's interactive property is `false`, no question will be asked and the file will simply be overwritten.

Console Output

The `Shell` class provides a few methods for outputting content:

```
// Write to stdout
$this->out('Normal message');

// Write to stderr
$this->err('Error message');

// Write to stderr and raise a stop exception
$this->abort('Fatal error');

// Before CakePHP 3.2. Write to stderr and exit()
$this->error('Fatal error');
```

It also provides two convenience methods regarding the output level:

```
// Would only appear when verbose output is enabled (-v)
$this->verbose('Verbose message');

// Would appear at all levels.
$this->quiet('Quiet message');
```

Shell also includes methods for clearing output, creating blank lines, or drawing a line of dashes:

```
// Output 2 newlines
$this->out($this->nl(2));

// Clear the user's screen
$this->clear();

// Draw a horizontal line
$this->hr();
```

Lastly, you can update the current line of text on the screen using `_io->overwrite()`:

```
$this->out('Counting down');
$this->out('10', 0);
for ($i = 9; $i > 0; $i--) {
    sleep(1);
    $this->_io->overwrite($i, 0, 2);
}
```

It is important to remember, that you cannot overwrite text once a new line has been output.

Console Output Levels

Shells often need different levels of verbosity. When running as cron jobs, most output is un-necessary. And there are times when you are not interested in everything that a shell has to say. 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 shell. `Cake\Console\Shell::out()` supports 3 types of output by default.

- `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.
$this->out('Quiet message', 1, Shell::QUIET);
$this->quiet('Quiet message');

// Would not appear when quiet output is toggled.
$this->out('normal message', 1, Shell::NORMAL);
$this->out('loud message', 1, Shell::VERBOSE);
$this->verbose('Verbose output');

// Would only appear when verbose output is enabled.
$this->out('extra message', 1, Shell::VERBOSE);
$this->verbose('Verbose output');
```

You can control the output level of shells, by using the `--quiet` and `--verbose` options. These options are added by default, and allow you to consistently control output levels inside your CakePHP shells.

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. ConsoleOutput will replace these tags with the correct ansi code sequence, or remove the tags 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

- `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 `$this->stdout->styles()`. To declare a new output style you could do:

```
$this->_io->styles('flashy', ['text' => 'magenta', 'blink' => true]);
```

This would then allow you to use a `<flashy>` tag in your shell output, and if ansi colours 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 colours 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 Colouring

Although colouring is pretty awesome, there may be times when you want to turn it off, or force it on:

```
$this->_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 colour output. On Windows systems, plain output is the default unless the `ANSICON` environment variable is present.

Stopping Shell Execution

When your shell commands have reached a condition where you want execution to stop, you can use `abort()` to raise a `StopException` that will halt the process:

```
$user = $this->Users->get($this->args[0]);
if (!$user) {
    // Halt with an error message and error code.
    $this->abort('User cannot be found', 128);
}
```

New in version 3.2: The abort() method was added in 3.2. In prior versions you can use `error()` to output a message and stop execution.

Status and Error Codes

Command-line tools should return 0 to indicate success, or a non-zero value to indicate an error condition. Since PHP methods usually return `true` or `false`, the Cake Shell `dispatch` function helps to bridge these semantics by converting your `null` and `true` return values to 0, and all other values to 1.

The Cake Shell `dispatch` function also catches the `StopException` and uses its exception code value as the shell's exit code. As described above, you can use the `abort()` method to print a message and exit with a specific code, or raise the `StopException` directly as shown in the example:

```
namespace App\Shell\Task;

use Cake\Console\Shell;

class ErroneousShell extends Shell
{
    public function main()
    {
        return true;
    }

    public function itFails()
    {
```

```
        return false;
    }

    public function itFailsSpecifically()
    {
        throw new StopException("", 2);
    }
}
```

The example above will return the following exit codes when executed on a command-line:

```
$ bin/cake erroneousshell ; echo $?
0
$ bin/cake erroneousshell itFails ; echo $?
1
$ bin/cake erroneousshell itFailsSpecifically ; echo $?
2
```

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.

Note: You can read more about conventional exit codes in the sysexit manual page on most Unix systems (`man sysexits`), or the `System Error Codes` help page in Windows.

Hook Methods

`Cake\Console\Shell::initialize()`
: Initializes the Shell, acts as constructor for subclasses and allows configuration of tasks prior to shell execution.

`Cake\Console\Shell::startup()`
: Starts up the Shell and displays the welcome message. Allows for checking and configuring prior to command or main execution.

Tip: Override the `startup()` method if you want to remove the welcome information, or otherwise modify the pre-command flow.

Configuring Options and Generating Help

class `Cake\Console\ConsoleOptionParser`

`ConsoleOptionParser` provides a command line option and argument parser.

OptionParsers allow you to accomplish two goals at the same time. First, they allow you to define the options and arguments for your commands. This allows you to separate basic input validation and your console commands. Secondly, it allows you to provide documentation, that is used to generate a well formatted help file.

The console framework in CakePHP gets your shell's option parser by calling `$this->getOptionParser()`. Overriding this method allows you to configure the OptionParser to define the expected inputs of your shell. You

can also configure subcommand option parsers, which allow you to have different option parsers for subcommands and tasks. The ConsoleOptionParser implements a fluent interface and includes methods for setting multiple options/arguments at once:

```
public function getOptionParser()
{
    $parser = parent::getOptionParser();
    // Configure parser
    return $parser;
}
```

Configuring an Option Parser with the Fluent Interface

All of the methods that configure an option parser can be chained, allowing you to define an entire option parser in one series of method calls:

```
public function getOptionParser()
{
    $parser = parent::getOptionParser();
    $parser->addArgument('type', [
        'help' => 'Either a full path or type of class.'
    ])->addArgument('className', [
        'help' => 'A CakePHP core class name (e.g: Component, HtmlHelper).'
    ])->addOption('method', [
        'short' => 'm',
        'help' => __('The specific method you want help on.')
    ])->setDescription(__('Lookup doc block comments for classes in CakePHP.'));
    return $parser;
}
```

The methods that allow chaining are:

- addArgument()
- addArguments()
- addOption()
- addOptions()
- addSubcommand()
- addSubcommands()
- setCommand()
- setDescription()
- setEpilog()

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']);
// Prior to 3.4
$parser->description(['line one', 'line two']);

// Read the current value
$parser->getDescription();
```

The **src/Shell/ConsoleShell.php** is a good example of the `description()` method in action:

```
/**
 * Display help for this console.
 *
 * @return ConsoleOptionParser
 */
public function getOptionParser()
{
    $parser = new ConsoleOptionParser('console');
    $parser->setDescription(
        'This shell provides a REPL that you can use to interact ' .
        'with your application in an interactive fashion. You can use ' .
        'it to run adhoc queries with your models, or experiment ' .
        'and explore the features of CakePHP and your application.' .
        "\n\n" .
        'You will need to have psysh installed for this Shell to work.'
    );
    return $parser;
}
```

The console's `description` output can be seen by executing the following command:

```
$ bin/cake console --help

Welcome to CakePHP v3.0.13 Console
---------------------------------------------------------------
App : src
Path: /home/user/cakeblog/src/
---------------------------------------------------------------
This shell provides a REPL that you can use to interact with your
application in an interactive fashion. You can use it to run adhoc
queries with your models, or experiment and explore the features of
CakePHP and your application.

You will need to have psysh installed for this Shell to work.

Usage:
cake console [-h] [-v] [-q]

Options:

--help, -h     Display this help.
--verbose, -v  Enable verbose output.
--quiet, -q    Enable quiet output.
```

Set a help alias

`Cake\Console\ConsoleOptionParser::setHelpAlias(`*$alias*`)`

If you want to change the command name, you can use the `setHelpAlias()` method:

```
$parser->setHelpAlias('my-shell');
```

This will change the usage output to `my-shell` instead of the default `cake` value:

```
Usage:
my-shell console [-h] [-v] [-q]
```

New in version 3.5.0: The `setHelpAlias` method was added in 3.5.0

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. Calling with no arguments will return the current value:

```
// Set multiple lines at once
$parser->setEpilog(['line one', 'line two']);
// Prior to 3.4
$parser->epilog(['line one', 'line two']);

// Read the current value
$parser->getEpilog();
```

To illustrate the `epilog()` method in action lets add a call to the `getOptionParser()` method used above in the **src/Shell/ConsoleShell.php**:

```
/**
 * Display help for this console.
 *
 * @return ConsoleOptionParser
 */
public function getOptionParser()
{
    $parser = new ConsoleOptionParser('console');
    $parser->setDescription(
        'This shell provides a REPL that you can use to interact ' .
        'with your application in an interactive fashion. You can use ' .
        'it to run adhoc queries with your models, or experiment ' .
        'and explore the features of CakePHP and your application.' .
        "\n\n" .
        'You will need to have psysh installed for this Shell to work.'
    );
    $parser->setEpilog('Thank you for baking with CakePHP!');
    return $parser;
}
```

The text added with the `setEpilog()` method can be seen in the output from the following console command:

```
$ bin/cake console --help

Welcome to CakePHP v3.0.13 Console
---------------------------------------------------------------
App : src
```

```
Path: /home/user/cakeblog/src/
---------------------------------------------------------------
This shell provides a REPL that you can use to interact with your
application in an interactive fashion. You can use it to run adhoc
queries with your models, or experiment and explore the features of
CakePHP and your application.

You will need to have psysh installed for this Shell to work.

Usage:
cake console [-h] [-v] [-q]

Options:

--help, -h     Display this help.
--verbose, -v  Enable verbose output.
--quiet, -q    Enable quiet output.

Thank you for baking with CakePHP!
```

Adding 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.

`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.

Adding Options

`Cake\Console\ConsoleOptionParser::addOption`(*$name*, *$options = []*)

Options or flags are also frequently used in command line tools. `ConsoleOptionParser` supports creating options with both verbose and short aliases, supplying defaults and creating boolean switches. Options are created with either `$parser->addOption()` or `$parser->addOptions()`.

```
$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. You can also create boolean switches, these switches do not consume values, and their presence just enables them in the parsed parameters.

```
$parser->addOption('no-commit', ['boolean' => true]);
```

With this option, when calling a shell like `cake myshell --no-commit something` the no-commit param would have a value of `true`, and 'something' would be a treated as a positional argument. The built-in `--help`, `--verbose`, and `--quiet` options use this feature.

When creating options you can use the following options to define the behavior of the option:

- **`short` - The single letter variant for this option, leave undefined for** none.
- **`help` - Help text for this option. Used when generating help for the** option.
- **`default` - The default value for this option. If not defined the default** will be `true`.
- `boolean` - The option uses no value, it's just a boolean switch. Defaults to `false`.
- `choices` - An array of valid choices for this option. If left empty all values are valid. An exception will be raised when parse() encounters an invalid value.

`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.

Option values are stored in the `$this->params` array. You can also use the convenience method `$this->param()` to avoid errors when trying to access non-present options.

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 result in `$this->params['verbose']` always being available. This lets you omit `empty()` or `isset()` checks for boolean flags:

```
if ($this->params['verbose']) {
    // Do something.
}
```

Since the boolean options are always defined as `true` or `false` you can omit additional check methods when using the array access. The `$this->param()` method makes these checks unnecessary for all cases.

Adding Subcommands

`Cake\Console\ConsoleOptionParser::addSubcommand`(*$name*, *$options = []*)

Console applications are often made of subcommands, and these subcommands may require special option parsing and have their own help. A perfect example of this is `bake`. Bake is made of many separate tasks that all have their own help and options. `ConsoleOptionParser` allows you to define subcommands and provide command specific option parsers so the shell knows how to parse commands for its tasks:

```
$parser->addSubcommand('model', [
    'help' => 'Bake a model',
    'parser' => $this->Model->getOptionParser()
]);
```

The above is an example of how you could provide help and a specialized option parser for a shell's task. By calling the Task's `getOptionParser()` we don't have to duplicate the option parser generation, or mix concerns in our shell. Adding subcommands in this way has two advantages. First, it lets your shell document its subcommands in

the generated help. It also gives easy access to the subcommand help. With the above subcommand created you could call `cake myshell --help` and see the list of subcommands, and also run `cake myshell model --help` to view the help for just the model task.

Note: Once your Shell defines subcommands, all subcommands must be explicitly defined.

When defining a subcommand you can use the following options:

- `help` - Help text for the subcommand.
- `parser` - A ConsoleOptionParser for the subcommand. This allows you to create method specific option parsers. When help is generated for a subcommand, if a parser is present it will be used. You can also supply the parser as an array that is compatible with *Cake\Console\ConsoleOptionParser::buildFromArray()*

Adding subcommands can be done as part of a fluent method chain.

Changed in version 3.5.0: When adding multi-word subcommands you can now invoke those commands using `snake_case` in addition to the camelBacked form.

Building a ConsoleOptionParser from an Array

`Cake\Console\ConsoleOptionParser::buildFromArray`(*$spec*)

As previously mentioned, when creating subcommand option parsers, you can define the parser spec as an array for that method. This can help make building subcommand parsers easier, as everything is an array:

```
$parser->addSubcommand('check', [
    'help' => __('Check the permissions between an ACO and ARO.'),
    'parser' => [
        'description' => [
            __("Use this command to grant ACL permissions. Once executed, the "),
            __("ARO specified (and its children, if any) will have ALLOW access "),
            __("to the specified ACO action (and the ACO's children, if any).")
        ],
        'arguments' => [
            'aro' => ['help' => __('ARO to check.'), 'required' => true],
            'aco' => ['help' => __('ACO to check.'), 'required' => true],
            'action' => ['help' => __('Action to check')]
        ]
    ]
]);
```

Inside the parser spec, you can define keys for `arguments`, `options`, `description` and `epilog`. You cannot define `subcommands` inside an array style builder. The values for arguments, and options, should follow the format that *Cake\Console\ConsoleOptionParser::addArguments()* and *Cake\Console\ConsoleOptionParser::addOptions()* use. You can also use buildFromArray on its own, to build an option parser:

```
public function getOptionParser()
{
    return ConsoleOptionParser::buildFromArray([
        'description' => [
            __("Use this command to grant ACL permissions. Once executed, the "),
            __("ARO specified (and its children, if any) will have ALLOW access "),
            __("to the specified ACO action (and the ACO's children, if any).")
        ],
```

```
        'arguments' => [
            'aro' => ['help' => __('ARO to check.'), 'required' => true],
            'aco' => ['help' => __('ACO to check.'), 'required' => true],
            'action' => ['help' => __('Action to check')]
        ]
    ]);
}
```

Merging ConsoleOptionParsers

`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

With the addition of ConsoleOptionParser getting help from shells is done in a consistent and uniform way. By using the `--help` or `-h` option you can view the help for any core shell, and any shell that implements a ConsoleOptionParser:

```
cake bake --help
cake bake -h
```

Would both generate the help for bake. If the shell supports subcommands you can get help for those in a similar fashion:

```
cake bake model --help
cake bake model -h
```

This would get you the help specific to bake's model task.

Getting Help as XML

When building automated tools or development tools that need to interact with CakePHP shells, it's nice to have help available in a machine parse-able format. The ConsoleOptionParser can provide help in xml by setting an additional argument:

```
cake bake --help xml
cake bake -h xml
```

The above would return an XML document with the generated help, options, arguments and subcommands for the selected shell. A sample XML document would look like:

```
<?xml version="1.0"?>
<shell>
    <command>bake fixture</command>
    <description>Generate fixtures for use with the test suite. You can use
        `bake fixture all` to bake all fixtures.</description>
```

```
    <epilog>
        Omitting all arguments and options will enter into an interactive
        mode.
    </epilog>
    <subcommands/>
    <options>
        <option name="--help" short="-h" boolean="1">
            <default/>
            <choices/>
        </option>
        <option name="--verbose" short="-v" boolean="1">
            <default/>
            <choices/>
        </option>
        <option name="--quiet" short="-q" boolean="1">
            <default/>
            <choices/>
        </option>
        <option name="--count" short="-n" boolean="">
            <default>10</default>
            <choices/>
        </option>
        <option name="--connection" short="-c" boolean="">
            <default>default</default>
            <choices/>
        </option>
        <option name="--plugin" short="-p" boolean="">
            <default/>
            <choices/>
        </option>
        <option name="--records" short="-r" boolean="1">
            <default/>
            <choices/>
        </option>
    </options>
    <arguments>
        <argument name="name" help="Name of the fixture to bake.
            Can use Plugin.name to bake plugin fixtures." required="">
            <choices/>
        </argument>
    </arguments>
</shell>
```

Renaming Commands

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 the commands that are exposed:

```
namespace App;

use App\Shell\UserShell;
use App\Shell\VersionShell;
use Cake\Http\BaseApplication;
```

```
class Application extends BaseApplication
{
    public function console($commands)
    {
        // Add by classname
        $commands->add('user', UserShell::class);

        // Add instance
        $commands->add('version', new VersionShell());

        return $commands;
    }
}
```

In the above example, the only commands available would be `help`, `version` and `user`.

New in version 3.5.0: The `console` hook was added.

Routing in Shells / CLI

In command-line interface (CLI), specifically your shells and tasks, `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();
// Prior to 3.4 use domain()
$email->setDomain('www.example.org');
```

This asserts that the generated message IDs are valid and fit to the domain the emails are sent from.

More Topics

Shell Helpers

New in version 3.1: Shell Helpers were added in 3.1.0

Shell Helpers let you package up complex output generation code. Shell Helpers can be accessed and used from any shell or task:

```
// Output some data as a table.
$this->helper('Table')->output($data);

// Get a helper from a plugin.
$this->helper('Plugin.HelperName')->output($data);
```

You can also get instances of helpers and call any public methods on them:

```
// Get and use the Progress Helper.
$progress = $this->helper('Progress');
$progress->increment(10);
$progress->draw();
```

Creating Helpers

While CakePHP comes with a few shell 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/Shell/Helper/HeadingHelper.php** and put the following in it:

```
<?php
namespace App\Shell\Helper;

use Cake\Console\Helper;

class HeadingHelper extends Helper
{
    public function output($args)
    {
        $args += ['', '#', 3];
        $marker = str_repeat($args[1], $args[2]);
        $this->_io->out($marker . ' ' . $args[0] . ' ' . $marker);
    }
}
```

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.

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'],
];
$this->helper('Table')->output($data);

// Outputs
+--------------+---------------+---------------+
| Header 1     | Header        | Long Header   |
+--------------+---------------+---------------+
```

```
| short        | Longish thing | short          |
| Longer thing | short         | Longest Value |
+--------------+---------------+----------------+
```

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:

```
$this->helper('Progress')->output(['callback' => function ($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:

```
$this->helper('Progress')->output([
    'total' => 10,
    'width' => 20,
    'callback' => function ($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 = $this->helper('Progress');
$progress->init([
    'total' => 10,
    'width' => 20,
]);

$progress->increment(4);
$progress->draw();
```

Interactive Console (REPL)

The CakePHP app skeleton comes with a built in REPL(Read Eval Print Loop) that makes it easy to explore some CakePHP and your application in an interactive console. 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

Welcome to CakePHP v3.0.0 Console
---------------------------------------------------------------
App : App
Path: /Users/mark/projects/cakephp-app/src/
---------------------------------------------------------------
>>> $articles = Cake\ORM\TableRegistry::get('Articles');
// object(Cake\ORM\Table)(
//
// )
>>> $articles->find()->all();
```

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 URL's:

```
>>> 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`.

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)
# |    |    \--------------- day of month (1 - 31)
# |    \-------------------- hour (0 - 23)
# \------------------------- min (0 - 59)
```

You can see more info here: http://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 myshell myparam` might not work. Instead you can use `php /full/path/to/root/bin/cake.php myshell 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`

I18N Shell

The i18n features of CakePHP use po files[137] as their translation source. PO files integrate with commonly used translation tools like Poedit[138].

The i18n shell provides a quick and easy 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 shell. The result of this command will be the file **src/Locale/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
```

[137] http://en.wikipedia.org/wiki/GNU_gettext
[138] http://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 Test,Vendor
```

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
```

Completion Shell

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}) )
        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         orm_cache   routes
console      migrations   plugin      server
```

Subcommands

Sample output for subcommands autocompletion:

```
$ bin/cake bake <tab>
behavior            helper              shell
cell                mailer              shell_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
```

Plugin Shell

The plugin shell 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 **config/bootstrap.php**:

```
Plugin::load('MyPlugin');
```

Adding the `-b` or `-r` switch to the load task will enable loading of the plugin's `bootstrap` and `routes` values:

```
bin/cake plugin load -b MyPlugin

// Load the bootstrap.php from the plugin
Plugin::load('MyPlugin', ['bootstrap' => true]);

bin/cake plugin load -r MyPlugin
```

```
// Load the routes.php from the plugin
Plugin::load('MyPlugin', ['routes' => true]);
```

If you are loading a plugin that only provides CLI tools - like bake - you can update your `bootstrap_cli.php` with:

```
bin/cake plugin load --cli MyPlugin
bin/cake plugin unload --cli MyPlugin
```

New in version 3.4.0: As of 3.4.0 the `--cli` option is supported

Unloading Plugins

You can unload a plugin by specifying its name:

```
bin/cake plugin unload MyPlugin
```

This will remove the line `Plugin::load('MyPlugin',...)` from your **config/bootstrap.php**.

Plugin Assets

CakePHP by default serves plugins assets using the `AssetFilter` dispatcher filter. 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 Shell

New in version 3.1: The RoutesShell was added in 3.1

The RoutesShell 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 /bookmarks/edit/1
```

If your route contains any query string parameters remember to surround the URL in quotes:

```
bin/cake routes check "/bookmarks/?page=1&sort=title&direction=desc"
```

Testing URL Generation

You can see how which URL a *routing array* will generate using the `generate` method:

```
bin/cake routes generate controller:Bookmarks action:edit 1
```

Upgrade Shell

The upgrade shell will do most of the work to upgrade your CakePHP application from 2.x to 3.x.

It is provided by a standalone Upgrade plugin[139]. Please read the README file to get all information on how to upgrade your application.

Server Shell

The `ServerShell` 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 shell 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.

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
```

Cache Shell

To help you better manage cached data from a CLI environment, a shell 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
```

New in version 3.3.0: The cache shell was added in 3.3.0

[139] https://github.com/cakephp/upgrade

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`.

New in version 3.3.0: Calling this method will return passed `$var`, so that you can, for instance, place it in return statements, for example:

```
return debug($data); // will return $data in any case.
```

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()

New in version 3.1.

If you have Psysh[140] installed you can use this function in CLI enviroments to open an interactive console with the current local scope:

```
// Some code
eval(breakpoint());
```

[140] http://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`.

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',
]);
```

New in version 3.4.0: Output masking was added in 3.4.0

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.
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

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:

```
// 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 *Debug Kit* chapter for how to install and use DebugKit.

CHAPTER 17

Deployment

Once your application is complete, or even before that you'll want to deploy it. There are a few things you should do when deploying a CakePHP application.

Moving files

You are encouraged to create a git commit and pull or clone that commit or repository on your server and 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 at least 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 dev box.

Adjust config/app.php

Adjusting app.php, specifically 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 types of things:

- Debug messages, created with *`pr()`*, *`debug()`* and *`dd()`* are disabled.
- Core CakePHP caches are by default flushed every year (about 365 days), instead of every 10 seconds as in development.
- Error views are less informative, and give generic error messages instead.
- PHP Errors are not displayed.
- Exception stack traces are disabled.

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.php**:

```
$debug = (bool)getenv('CAKEPHP_DEBUG');

return [
    'debug' => $debug,
    .....
];
```

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* component.
- You may want to enable the *Security* 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` shell:

```
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
```

Deploying an update

After deployment of an update you might also want to run `bin/cake orm_cache clear`, part of the *ORM Cache Shell* shell.

CHAPTER 18

Email

Warning: Before version 3.1, the `Email` and `Transport` classes were under the `Cake\Network\Email` namespace instead of the `Cake\Mailer` namespace.

class `Cake\Mailer\`**`Email`** (*mixed $profile = null*)

`Email` is a new class to send 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\Email;
```

After you've loaded `Email`, you can send an email with the following:

```
$email = new Email('default');
$email->from(['me@example.com' => 'My Site'])
    ->to('you@example.com')
    ->subject('About')
    ->send('My message');
```

Since `Email`'s setter methods return the instance of the class, you are able to set its properties with method chaining.

`Email` has several methods for defining recipients - `to()`, `cc()`, `bcc()`, `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:

```
$email = new Email();
$email->to('to@example.com', 'To Example');
$email->addTo('to2@example.com', 'To2 Example');
// The email's To recipients are: to@example.com and to2@example.com
$email->to('test@example.com', 'ToTest Example');
// The email's To recipient is: test@example.com
```

Deprecated since version 3.4.0: Use `setFrom()`, `setTo()`, `setCc()` , `setBcc()` and `setSubject()` instead.

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 `sender()`:

```
$email = new Email();
$email->sender('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.

Deprecated since version 3.4.0: Use `setSender()` instead.

Configuration

Configuration for `Email` defaults is created using `config()` and `configTransport()`. You should put your email presets in the **config/app.php** file. The **config/app.default.php** file is an example of this file. It is not required to define email configuration in **config/app.php**. `Email` can be used without it and use the respective methods to set all configurations separately or load an array of configs.

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 `profile()` method or pass it to the constructor of `Email`:

```
$email = new Email();
$email->profile('default');

// Or in constructor
$email = new Email('default');
```

Instead of passing a string which matches a preset configuration name, you can also just load an array of options:

```
$email = new Email();
$email->profile(['from' => 'me@example.org', 'transport' => 'my_custom']);

// Or in constructor
$email = new Email(['from' => 'me@example.org', 'transport' => 'my_custom']);
```

Changed in version 3.1: The `default` email profile is automatically set when an `Email` instance is created.

Deprecated since version 3.4.0: Use `setProfile()` instead of `profile()`.

Configuring Transports

static `Cake\Mailer\Email::configTransport`(*$key*, *$config*)

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:

```
use Cake\Mailer\Email;

// Sample Mail configuration
Email::configTransport('default', [
    'className' => 'Mail'
]);

// Sample SMTP configuration.
Email::configTransport('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\Email;

Email::configTransport('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.

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[141].

[141] https://support.google.com/accounts/answer/6010255

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:

```
Email::configTransport('default', [
    'url' => 'smtp://my@gmail.com:secret@smtp.gmail.com:465?tls=true',
]);
```

When using a DSN string you can define any additional parameters/options as query string arguments.

Deprecated since version 3.4.0: Use `setConfigTransport()` instead of `configTransport()`.

static `Cake\Mailer\Email::dropTransport`(*$key*)

Once configured, transports cannot be modified. In order to modify a transport you must first drop it and then reconfigure it.

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'`: Email or array of sender. See `Email::from()`.
- `'sender'`: Email or array of real sender. See `Email::sender()`.
- `'to'`: Email or array of destination. See `Email::to()`.
- `'cc'`: Email or array of carbon copy. See `Email::cc()`.
- `'bcc'`: Email or array of blind carbon copy. See `Email::bcc()`.
- `'replyTo'`: Email or array to reply the e-mail. See `Email::replyTo()`.
- `'readReceipt'`: Email address or an array of addresses to receive the receipt of read. See `Email::readReceipt()`.
- `'returnPath'`: Email address or an array of addresses to return if have some error. See `Email::returnPath()`.
- `'messageId'`: Message ID of e-mail. See `Email::messageId()`.
- `'subject'`: Subject of the message. See `Email::subject()`.
- `'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 `Email::headers()`.
- `'viewRender'`: If you are using rendered content, set the view classname. See `Email::viewRender()`.
- `'template'`: If you are using rendered content, set the template name. See `Email::template()`.
- `'theme'`: Theme used when rendering template. See `Email::theme()`.
- `'layout'`: If you are using rendered content, set the layout to render. If you want to render a template without layout, set this field to null. See `Email::template()`.
- `'viewVars'`: If you are using rendered content, set the array with variables to be used in the view. See `Email::viewVars()`.
- `'attachments'`: List of files to attach. See `Email::attachments()`.
- `'emailFormat'`: Format of email (html, text or both). See `Email::emailFormat()`.
- `'transport'`: Transport configuration name. See *`Mailer\Email::configTransport()`*.
- `'log'`: Log level to log the email headers and message. `true` will use LOG_DEBUG. See also *Using Levels*.
- `'helpers'`: Array of helpers used in the email template. `Email::helpers()`.

All these configurations are optional, except `'from'`.

Note: The values of above keys using Email or array, like from, to, cc, etc will be passed as first parameter of corresponding methods. The equivalent for: `Email::from('my@example.com','My Site')` would be defined as `'from' => ['my@example.com' => 'My Site']` in your config

---

Setting Headers

In `Email` you are free to set whatever headers you want. When migrating to use Email, do not forget to put the `X-` prefix in your headers.

See `Email::headers()` and `Email::addHeaders()`

Deprecated since version 3.4.0: Use `setHeaders()` instead of `headers()`.

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 in your application's `Template` directory called `Email`. Email views can also use layouts and elements just like normal views:

```
$email = new Email();
$email
    ->template('welcome', 'fancy')
    ->emailFormat('html')
    ->to('bob@example.com')
    ->from('app@domain.com')
    ->send();
```

The above would use **src/Template/Email/html/welcome.ctp** for the view and **src/Template/Layout/Email/html/fancy.ctp** for the layout. You can send multipart templated email messages as well:

```
$email = new Email();
$email
    ->template('welcome', 'fancy')
    ->emailFormat('both')
    ->to('bob@example.com')
    ->from('app@domain.com')
    ->send();
```

This would use the following template files:

- **src/Template/Email/text/welcome.ctp**
- **src/Template/Layout/Email/text/fancy.ctp**
- **src/Template/Email/html/welcome.ctp**
- **src/Template/Layout/Email/html/fancy.ctp**

When sending templated emails you have the option of sending either `text`, `html` or `both`.

You can set view variables with `Email::viewVars()`:

```
$email = new Email('templated');
$email->viewVars(['value' => 12345]);
```

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 `helpers()` method:

```
$email->helpers(['Html', 'Custom', 'Text']);
```

When setting helpers be sure to include 'Html' or it will be removed from the helpers loaded in your email template.

If you want to send email using templates in a plugin you can use the familiar *plugin syntax* to do so:

```
$email = new Email();
$email->template('Blog.new_comment', 'Blog.auto_message');
```

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 by telling Email to use appropriate theme using `Email::theme()` method:

```
$email = new Email();
$email->template('Blog.new_comment', 'Blog.auto_message');
$email->theme('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: **src/Template/Plugin/TestTheme/Plugin/Blog/Email/text/new_comment.ctp**.

Deprecated since version 3.4.0: Use `setTemplate()` instead of `template()`. Use `setLayout()` instead of the layout argument of `template()`. Use `setTheme()` instead of `theme()`.

Sending Attachments

`Cake\Mailer\Email::attachments`(*$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. String: `$email->attachments('/full/file/path/file.png')` will attach this file with the name file.png.
2. Array: `$email->attachments(['/full/file/path/file.png'])` will have the same behavior as using a string.
3. Array with key: `$email->attachments(['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.
4. Nested arrays:

   ```
   $email->attachments([
       'photo.png' => [
           'file' => '/full/some_hash.png',
           '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.

   4.1. When you are using the `contentId`, you can use the file in the HTML body like `<img src="cid:my-content-id">`.

4.2. 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.

4.3 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.

Deprecated since version 3.4.0: Use `setAttachments()` instead of `attachments()`.

Using Transports

Transports are classes designed to send the e-mail over some protocol or method. CakePHP supports the Mail (default), Debug and SMTP transports.

To configure your method, you must use the `Cake\Mailer\Email::transport()` method or have the transport in your configuration:

```
$email = new Email();

// Use a named transport already configured using Email::configTransport()
$email->transport('gmail');

// Use a constructed object.
$transport = new DebugTransport();
$email->transport($transport);
```

Deprecated since version 3.4.0: Use `setTransport()` instead of `transport()`.

Creating Custom Transports

You are able to create your custom transports to integrate with others email systems (like SwiftMailer). To create your transport, first create the file **src/Mailer/Transport/ExampleTransport.php** (where Example is the name of your transport). To start off your file should look like:

```
namespace App\Mailer\Transport;

use Cake\Mailer\AbstractTransport;
use Cake\Mailer\Email;

class ExampleTransport extends AbstractTransport
{
    public function send(Email $email)
    {
        // Do something.
    }
}
```

You must implement the method `send(Email $email)` with your custom logic. Optionally, you can implement the `config($config)` method. `config()` is called before send() and allows you to accept user configurations. By default, this method puts the configuration in protected attribute `$_config`.

If you need to call additional methods on the transport before send, you can use `Cake\Mailer\Email::getTransport()` to get an instance of the transport object. Example:

```
$yourInstance = $email->getTransport()->transportClass();
$yourInstance->myCustomMethod();
$email->send();
```

Relaxing Address Validation Rules

`Cake\Mailer\Email::emailPattern(*$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 Japanese ISP's:

```
$email = new Email('default');

// Relax the email pattern, so you can send
// to non-conformant addresses.
$email->emailPattern($newPattern);
```

Deprecated since version 3.4.0: Use `setEmailPattern()` instead of `emailPattern()`.

Sending Messages Quickly

Sometimes you need a quick way to fire off an email, and you don't necessarily want do setup a bunch of configuration ahead of time. `Cake\Mailer\Email::deliver()` is intended for that purpose.

You can create your configuration using `Cake\Mailer\Email::config()`, or use an array with all options that you need and use the static method `Email::deliver()`. Example:

```
Email::deliver('you@example.com', 'Subject', 'Message', ['from' => 'me@example.com']);
```

This method will send an email to "you@example.com", from "me@example.com" with subject "Subject" and content "Message".

The return of `deliver()` is a *`Cake\Mailer\Email`* instance with all configurations set. If you do not want to send the email right away, and wish to configure a few things before sending, you can pass the 5th parameter as `false`.

The 3rd parameter is the content of message or an array with variables (when using rendered content).

The 4th parameter can be an array with the configurations or a string with the name of configuration in `Configure`.

If you want, you can pass the to, subject and message as null and do all configurations in the 4th parameter (as array or using `Configure`). Check the list of *configurations* to see all accepted configs.

Sending Emails from CLI

When sending emails within a CLI script (Shells, Tasks, ...) you should manually set the domain name for Email to use. It will serve as the host name for the message id (since there is no host name in a CLI environment):

```
$email->domain('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.

Deprecated since version 3.4.0: Use `setDomain()` instead of `domain()`.

Creating Reusable Emails

New in version 3.1.0.

Mailers allow you to create 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:

```
namespace App\Mailer;

use Cake\Mailer\Mailer;

class UserMailer extends Mailer
{
    public function welcome($user)
    {
        $this
            ->to($user->email)
            ->subject(sprintf('Welcome %s', $user->name))
            ->template('welcome_mail', 'custom'); // By default template with same␣
↪name as method name is used.
    }

    public function resetPassword($user)
    {
        $this
            ->to($user->email)
            ->subject('Reset password')
            ->set(['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:

```
namespace App\Controller;

use Cake\Mailer\MailerAwareTrait;

class UsersController extends AppController
{
    use MailerAwareTrait;

    public function register()
    {
        $user = $this->Users->newEntity();
        if ($this->request->is('post')) {
            $user = $this->Users->patchEntity($user, $this->request->getData())
            if ($this->Users->save($user)) {
                $this->getMailer('User')->send('welcome', [$user]);
            }
        }
```

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

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(Event $event, EntityInterface $entity, ArrayObject
↪$options)
{
    if ($entity->isNew()) {
        $this->send('welcome', [$entity]);
    }
}
```

The mailer object could now be registered as an event listener, and the `onRegistration()` method would be invoked every time the `Model.afterSave` event would be fired. For information on how to register event listener objects, please refer to the *Registering Listeners* documentation.

CHAPTER 19

Error & Exception Handling

Many of PHP's internal methods use errors to communicate failures. These errors will need to be trapped and dealt with. CakePHP comes with default error trapping that prints and or logs errors as they occur. This same error handler is used to catch uncaught exceptions from controllers and other parts of your application.

Error & Exception Configuration

Error configuration is done inside your application's **config/app.php** file. By default CakePHP uses the `ErrorHandler` or `ConsoleErrorHandler` class to trap errors and print/log the errors. You can replace this behavior by changing out the default error handler. The default error handler also handles uncaught exceptions.

Error handling accepts a few options that allow you to tailor error handling for your application:

- `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.
- `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.

ErrorHandler by default, displays errors when `debug` is `true`, and logs errors when debug is `false`. The type of errors captured in both cases is controlled by `errorLevel`. 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.

Creating your Own Error Handler

You can create an error handler out of any callback type. For example you could use a class called `AppError` to handle your errors. By extending the `BaseErrorHandler` you can supply custom logic for handling errors. An example would be:

```
// In config/bootstrap.php
use App\Error\AppError;

$errorHandler = new AppError();
$errorHandler->register();

// In src/Error/AppError.php
namespace App\Error;

use Cake\Error\BaseErrorHandler;

class AppError extends BaseErrorHandler
{
    public function _displayError($error, $debug)
    {
        echo 'There has been an error!';
    }
    public function _displayException($exception)
    {
        echo 'There has been an exception!';
    }
}
```

The `BaseErrorHandler` defines two abstract methods. `_displayError()` is used when errors are triggered. The `_displayException()` method is called when there is an uncaught exception.

Changing Fatal Error Behavior

The default error handlers convert fatal errors into exceptions and re-use the exception handling logic to render an error page. If you do not want to show the standard error page, you can override it like:

```
// In config/bootstrap.php
use App\Error\AppError;

$errorHandler = new AppError();
$errorHandler->register();

// In src/Error/AppError.php
namespace App\Error;

use Cake\Error\BaseErrorHandler;

class AppError extends BaseErrorHandler
{
    // Other methods.

    public function handleFatalError($code, $description, $file, $line)
    {
        return 'A fatal error has happened';
```

```
    }
}
```

Exception Classes

There are a number of exception classes in CakePHP. The built in exception handling will capture any uncaught exceptions and render a useful page. Exceptions that do not specifically use a 400 range code, will be treated as an Internal Server Error.

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\Network\Exception\`**`BadRequestException`**
Used for doing 400 Bad Request error.

exception `Cake\Network\Exception\`**`UnauthorizedException`**
Used for doing a 401 Unauthorized error.

exception `Cake\Network\Exception\`**`ForbiddenException`**
Used for doing a 403 Forbidden error.

New in version 3.1: InvalidCsrfTokenException has been added.

exception `Cake\Network\Exception\`**`InvalidCsrfTokenException`**
Used for doing a 403 error caused by an invalid CSRF token.

exception `Cake\Network\Exception\`**`NotFoundException`**
Used for doing a 404 Not found error.

exception `Cake\Network\Exception\`**`MethodNotAllowedException`**
Used for doing a 405 Method Not Allowed error.

exception `Cake\Network\Exception\`**`NotAcceptableException`**
Used for doing a 406 Not Acceptable error.

New in version 3.1.7: NotAcceptableException has been added.

exception `Cake\Network\Exception\`**`ConflictException`**
Used for doing a 409 Conflict error.

New in version 3.1.7: ConflictException has been added.

exception `Cake\Network\Exception\`**`GoneException`**
Used for doing a 410 Gone error.

New in version 3.1.7: GoneException has been added.

For more details on HTTP 4xx error status codes see **RFC 2616#section-10.4**[142].

exception `Cake\Network\Exception\`**`InternalErrorException`**
Used for doing a 500 Internal Server Error.

[142] https://tools.ietf.org/html/rfc2616.html#section-10.4

exception Cake\Network\Exception**NotImplementedException**
Used for doing a 501 Not Implemented Errors.

exception Cake\Network\Exception**ServiceUnavailableException**
Used for doing a 503 Service Unavailable error.

New in version 3.1.7: Service Unavailable has been added.

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\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->set('_serialize', ['article']);
}
```

By using exceptions for HTTP errors, you can keep your code both clean, and give RESTful responses to client applications and users.

Other Built In Exceptions

In addition, the following framework layer exceptions are available, and will be thrown from a number of CakePHP core components:

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.

[143] https://tools.ietf.org/html/rfc2616.html#section-10.5

exception Cake\Controller\Exception**PrivateActionException**
Accessing private/protected/_ prefixed actions.

exception Cake\Console\Exception**ConsoleException**
A console library class encounter an error.

exception Cake\Console\Exception**MissingTaskException**
A configured task could not found.

exception Cake\Console\Exception**MissingShellException**
The shell class could not be found.

exception Cake\Console\Exception**MissingShellMethodException**
The chosen shell class has no method of that name.

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()*.

New in version 3.4.1: PersistenceFailedException has been added.

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\Routing\Exception**MissingDispatcherFilterException**
The dispatcher filter could not be found.

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. All of the standard Exceptions that CakePHP will throw also extend Exception.

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."
```

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->set('_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.

Exception Renderer

class `Cake\Core\Exception\`**ExceptionRenderer**(*Exception $exception*)

The ExceptionRenderer class with the help of `ErrorController` takes care of rendering the error pages for all the exceptions thrown by your application.

The error page views are located at **src/Template/Error/**. For all 4xx and 5xx errors the template files **error400.ctp** and **error500.ctp** are used respectively. You can customize them as per your needs. By default your **src/Template/Layout/error.ctp** is used for error pages too. If for example, you want to use another layout **src/Template/Layout/my_error.ctp** for your error pages, simply edit the error views and add the statement `$this->layout = 'my_error;` to the **error400.ctp** and **error500.ctp**.

Each framework layer exception has its own view file located in the core templates but you really don't need to bother customizing them as they are used only during development. With debug turned off all framework layer exceptions are converted to `InternalErrorException`.

Creating your own Application Exceptions

You can create your own application exceptions using any of the built in SPL exceptions[144], `Exception` itself, or *`Cake\Core\Exception\Exception`*. If your application contained the following exception:

```
use Cake\Core\Exception\Exception;

class MissingWidgetException extends Exception
{};
```

[144] http://php.net/manual/en/spl.exceptions.php

You could provide nice development errors, by creating **src/Template/Error/missing_widget.ctp**. When in production mode, the above error would be treated as a 500 error. The constructor for *Cake\Core\Exception\Exception* has been extended, allowing you to pass in hashes of data. These hashes are interpolated into the the messageTemplate, as well as into the view that is used to represent the error in development mode. This allows you to create data rich exceptions, by providing more context for your errors. You can also provide a message template which allows the native `__toString()` methods to work as normal:

```
use Cake\Core\Exception\Exception;

class MissingWidgetException extends Exception
{
    protected $_messageTemplate = 'Seems that %s is missing.';
}

throw new MissingWidgetException(['widget' => 'Pointy']);
```

When caught by the built in exception handler, you would get a `$widget` variable in your error view template. In addition if you cast the exception as a string or use its `getMessage()` method you will get `Seems that Pointy is missing.`. This allows you to quickly create your own rich development errors, just like CakePHP uses internally.

Creating Custom Status Codes

You can create custom HTTP status codes by changing the code used when creating an exception:

```
throw new MissingWidgetHelperException('Its not here', 501);
```

Will create a 501 response code, you can use any HTTP status code you want. In development, if your exception doesn't have a specific template, and you use a code equal to or greater than 500 you will see the **error500.ctp** template. For any other error code you'll get the **error400.ctp** template. If you have defined an error template for your custom exception, that template will be used in development mode. If you'd like your own exception handling logic even in production, see the next section.

Extending and Implementing your own Exception Handlers

You can implement application specific exception handling in one of a few ways. Each approach gives you different amounts of control over the exception handling process.

- Create and register your own custom error handlers.
- Extend the `BaseErrorHandler` provided by CakePHP.
- Set the `exceptionRenderer` option on the default error handler.

In the next few sections, we will detail the various approaches and the benefits each has.

Create and Register your own Exception Handler

Creating your own exception handler gives you full control over the exception handling process. You will have to call `set_exception_handler` yourself in this situation.

Extend the BaseErrorHandler

The *Error & Exception Configuration* section has an example of this.

Using the exceptionRenderer Option of the Default Handler

If you don't want to take control of the exception handling, but want to change how exceptions are rendered you can use the `exceptionRenderer` option in **config/app.php** to choose a class that will render exception pages. By default *`Cake\Core\Exception\ExceptionRenderer`* is used. Your custom exception renderer class should be placed in **src/Error**. In a custom exception rendering class you can provide specialized handling for application specific errors:

```
// In src/Error/AppExceptionRenderer.php
namespace App\Error;

use Cake\Error\ExceptionRenderer;

class AppExceptionRenderer extends ExceptionRenderer
{
    public function missingWidget($error)
    {
        return 'Oops that widget is missing!';
    }
}


// In config/app.php
'Error' => [
    'exceptionRenderer' => 'App\Error\AppExceptionRenderer',
    // ...
],
// ...
```

The above would handle any exceptions of the type `MissingWidgetException`, and allow you to provide custom display/handling logic for those application exceptions. Exception handling methods get the exception being handled as their argument. Your custom exception rendering can return either a string or a `Response` object. Returning a `Response` will give you full control over the response.

Note: Your custom renderer should expect an exception in its constructor, and implement a render method. Failing to do so will cause additional errors.

If you are using a custom exception handling, configuring the renderer will have no effect. Unless you reference it inside your implementation.

Creating a Custom Controller to Handle Exceptions

By convention CakePHP will use `App\Controller\ErrorController` if it exists. Implementing this class can give you a configuration free way of customizing error page output.

If you are using custom exception renderer, you can use the `_getController()` method to return a customized controller. By implementing `_getController()` in your exception renderer you can use any controller you want:

```
// in src/Error/AppExceptionRenderer
namespace App\Error;

use App\Controller\SuperCustomErrorController;
use Cake\Error\ExceptionRenderer;

class AppExceptionRenderer extends ExceptionRenderer
{
    protected function _getController($exception)
    {
        return new SuperCustomErrorController();
    }
}

// in config/app.php
'Error' => [
    'exceptionRenderer' => 'App\Error\AppExceptionRenderer',
    // ...
],
// ...
```

The error controller, whether custom or conventional, is used to render the error page view and receives all the standard request life-cycle events.

Logging Exceptions

Using the built-in exception handling, you can log all the exceptions that are dealt with by ErrorHandler 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.

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('Model.Order.afterPlace', $this, [
                'order' => $order
            ]);
            $this->eventManager()->dispatch($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 `eventManager()`:

```
$events = $this->eventManager();
```

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(
    'Model.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)`.

New in version 3.2.11: Event tracking and `Cake\Event\EventList` were added.

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:

```
use Cake\Event\EventListenerInterface;

class UserStatistic implements EventListenerInterface
{

    public function implementedEvents()
    {
```

```
        return [
            'Model.Order.afterPlace' => 'updateBuyStatistic',
        ];
    }

    public function updateBuyStatistic($event, $order)
    {
        // Code to update statistics
    }
}

// Attach the UserStatistic object to the Order's event manager
$statistics = new UserStatistic();
$this->Orders->eventManager()->on($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;

$this->Orders->eventManager()->on('Model.Order.afterPlace', function ($event) {
    Log::write(
        'info',
        'A new order was placed with id: ' . $event->getSubject()->id
    );
});
```

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('Model.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 could 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\ORM\TableRegistry;
// If sending emails
use Cake\Mailer\Email;

TableRegistry::get('ThirdPartyPlugin.Feedbacks')
```

```
    ->eventManager()
    ->on('Model.afterSave', function($event, $entity)
    {
        // For example we can send an email to the admin
        $email = new Email('default');
        $email->from('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.
$this->eventManager()->on('User.Registration', [$this, 'userRegistration']);
$this->eventManager()->on('User.Verification', [$this, 'userVerification']);
$this->eventManager()->on('User.Authorization', [$this, 'userAuthorization']);

// Somewhere else in your application.
$events = $this->eventManager()->matchingListeners('Verification');
if (!empty($events)) {
    // Perform logic related to presence of 'Verification' event listener.
    // For example removing the listener if present.
    $this->eventManager()->off('User.Verification');
} else {
    // Perform logic related to absence of 'Verification' event listener
}
```

Note: The pattern passed to the `matchingListeners` method is case sensitive.

New in version 3.2.3: The `matchingListeners` method returns an array of events matching a search pattern.

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->eventManager()->on(
    'Model.Order.afterPlace',
    ['priority' => 2],
    $callback
);

// Setting priority for a listener
class UserStatistic implements EventListenerInterface
{
    public function implementedEvents()
    {
        return [
            'Model.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->eventManager()
    ->dispatch(new Event('View.afterRender', $this, ['view' => $viewFileName]));
```

The listeners of the `View.afterRender` callback should have the following signature:

```
function (Event $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 `Event\EventManager::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('Model.Order.afterPlace', $this, [
```

```
    'order' => $order
]);
$this->eventManager()->dispatch($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 (e.g. `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 `Event\EventManager::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 (e.g. 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('Model.Order.beforePlace', $this, ['order' => $order]);
    $this->eventManager()->dispatch($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)
{
    // ...
    $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('Model.Order.beforePlace', $this, ['order' => $order]);
    $this->eventManager()->dispatch($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 params you used for attaching it:

```
// Attaching a function
$this->eventManager()->on('My.event', [$this, 'doSomething']);

// Detaching the function
$this->eventManager()->off('My.event', [$this, 'doSomething']);

// Attaching an anonymous function.
$myFunction = function ($event) { ... };
$this->eventManager()->on('My.event', $myFunction);

// Detaching the anonymous function
$this->eventManager()->off('My.event', $myFunction);

// Adding a EventListener
$listener = new MyEventLister();
$this->eventManager()->on($listener);

// Detaching a single event key from a listener
$this->eventManager()->off('My.event', $listener);

// Detaching all callbacks implemented by a listener
$this->eventManager()->off($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*
- *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[145] format. Files need to be placed under **src/Locale/** and within this directory, there should be a subfolder for each language the application needs to support:

```
/src
    /Locale
        /en_US
            default.po
        /en_GB
            default.po
```

[145] http://en.wikipedia.org/wiki/Gettext

```
            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 **src/Locale/**. The CakePHP localized library[146] 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: **src/Locale/<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
    /src
        /Locale
            /fr
                my_plugin.po
            /de
                my_plugin.po
```

Translation folders can either be the two letter ISO code of the language 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.

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"
```

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 shell. 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.

[146] https://github.com/cakephp/localized

Changing the Locale at Runtime

To change the language for translated strings you can call this method:

```
use Cake\I18n\I18n;

// Prior to 3.5 use I18n::locale()
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 your plugins and they're namespaced, you must name your domain string `Namespace/PluginName`. But the related language file will become `plugins/Namespace/PluginName/src/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[147] 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}',
    ['Charles', new FrozenTime('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`

[147] http://php.net/manual/en/messageformatter.format.php

- `full`

The word `time` after the placeholder number is also accepted and it understands the same options as `date`.

Note: Named placeholders are supported in PHP 5.5+ and are formatted as `{name}`. When using named placeholders pass the variables in an array using key/value pairs, for example `['name' => 'Sara','age' => 12]`.

It is recommended to use PHP 5.5 or higher when making use of internationalization features in CakePHP. The `php5-intl` extension must be installed and the ICU version should be above 48.x.y (to check the ICU version `Intl::getIcuVersion()`).

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[148] 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"
```

[148] http://www.unicode.org/cldr/charts/latest/supplemental/language_plural_rules.html

```
msgstr[1] "{0} datoteke su uklonjene"
msgstr[2] "{0} datoteka je uklonjeno"
```

Please visit the Launchpad languages page[149] for a detailed explanation of the plural form numbers for each language.

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 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 `Aura\Intl\Package` object. Once the code is in place you can use the translation functions as usual:

```
// Prior to 3.5 use I18n::locale()
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 src/Locale/folder/sub_folder/filename.po
// Prior to 3.5 use translator()
I18n::setTranslator(
    'animals',
    new Loader('filename', 'folder/sub_folder', 'po'),
    'fr_FR'
);
```

[149] https://translations.launchpad.net/+languages

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 **src/Locale/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;

// Prior to 3.5 use translator()
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 Aura\Intl\Package;

I18n::config('default', function ($domain, $locale) {
    $locale = Locale::parseLocale($locale);
    $language = $locale['language'];
    $messages = file_get_contents("http://example.com/translations/$lang.json");

    return new Package(
        'default', // Formatter
        null, // Fallback (none for default domain)
        json_decode($messages, true)
    )
});
```

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.
});
```

New in version 3.4.0: Replacing the `_fallback` loader was added in 3.4.0

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\Time;
use Cake\I18n\Number;

// Prior to 3.5 use I18n::locale()
I18n::setLocale('fr-FR');

$date = new Time('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\Time` 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 *Dispatcher Filters* you can configure the Date, Time, and DateTime types to parse localized formats:

```
use Cake\Database\Type;

// Enable default locale format parsing.
Type::build('datetime')->useLocaleParser();

// Configure a custom datetime format parser format.
Type::build('datetime')->useLocaleParser()->setLocaleFormat('dd-M-y');

// You can also use IntlDateFormatter constants.
Type::build('datetime')->useLocaleParser()
    ->setLocaleFormat([IntlDateFormatter::SHORT, -1]);
```

The default parsing format is the same as the default string format.

Automatically Choosing the Locale Based on Request Data

By using the `LocaleSelectorFilter` 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
```

```
public function middleware($middleware)
{
    // Add middleware and set the valid locales
    $middleware->add(new LocaleSelectorMiddleware(['en_US', 'fr_FR']));
}


// Prior to 3.3.0, use the DispatchFilter
// in config/bootstrap.php
DispatcherFactory::add('LocaleSelector');

// Restrict the locales to only en_US, fr_FR
DispatcherFactory::add('LocaleSelector', ['locales' => ['en_US', 'fr_FR']]);
```

The `LocaleSelectorFilter` 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.

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 easy - the log() function 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\Core\Log`. An example would be:

```
use Cake\Log\Log;

// Short classname
Log::config('debug', [
    'className' => 'File',
    'path' => LOGS,
    'levels' => ['notice', 'info', 'debug'],
    'file' => 'debug',
]);

// Fully namespaced name.
Log::config('error', [
    'className' => 'Cake\Log\Engine\FileLog',
    'path' => LOGS,
    'levels' => ['warning', 'error', 'critical', 'alert', 'emergency'],
    'file' => 'error',
]);
```

The above creates two loggers. One named `debug` the other named `error`. Each is configured to handle different levels of messages. They also store their log messages in separate files, so it's easy to 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::config()*.

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::config('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::config('error', [
    'url' => 'file:///?levels[]=warning&levels[]=error&file=error',
]);
```

Note: Loggers are required to implement the `Psr\Log\LoggerInterface` interface.

Creating Log Adapters

Log adapters 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 adapters you should use *Cake\Log\Log::config()*. For example configuring our DatabaseLog would look like:

```
// For src/Log
Log::config('otherFile', [
    'className' => 'Database',
    'model' => 'LogEntry',
    // ...
]);

// For plugin called LoggingPack
Log::config('otherFile', [
    'className' => 'LoggingPack.Database',
    'model' => 'LogEntry',
    // ...
]);
```

When configuring a log adapter the `className` parameter is used to locate and load the log handler. All of the other configuration properties are passed to the log adapter's constructor as an array.

```
namespace App\Log\Engine;
use Cake\Log\Engine\BaseLog;

class DatabaseLog extends BaseLog
{
    public function __construct($options = [])
    {
        parent::__construct($options);
        // ...
```

```
    }

    public function log($level, $message, array $context = [])
    {
        // Write to the database.
    }
}
```

CakePHP requires that all logging adapters implement `Psr\Log\LoggerInterface`. The class `CakeLogEngineBaseLog` is an easy way to satisfy the interface as it only requires you to implement the `log()` method. `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.

Warning: Engines have the suffix `Log`. You should avoid class names like `SomeLogLog` which include the suffix twice at the end.

Note: You should configure loggers during bootstrapping. **config/app.php** is the conventional place to configure log adapters.

In debug mode missing directories will be automatically created to avoid unnecessary errors thrown when using the FileEngine.

Error and Exception Logging

Errors and Exceptions can also be logged. By configuring the co-responding values in your app.php file. Errors will be displayed when debug > 0 and logged when debug is `false`. To log uncaught exceptions, set the `log` option to `true`. See *Configuration* for more information.

Interacting with Log Streams

You can introspect the configured streams with *`Cake\Log\Log::configured()`*. The return of `configured()` is an array of all the currently configured streams. You can remove streams using *`Cake\Log\Log::drop()`*. Once a log stream has been dropped it will no longer receive messages.

Using the FileLog Adapter

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::config('custom_path', [
    'className' => 'File',
    'path' => '/path/to/custom/place/'
]);
```

Warning: If you do not configure a logging adapter, log messages will not be stored.

Logging to Syslog

In production environments it is highly recommended that you setup your system to use syslog instead of the files 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 **bootstrap.php** file:

```
Log::config('default', [
    'engine' => 'Syslog'
]);
```

The configuration array accepted for the Syslog logging engine understands the following keys:

- `format`: An sprintf template strings 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.
- `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

Writing to Logs

Writing to the log files can be done in 2 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 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 adapters `log()` will return `false` and no log messages will be written.

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:

```
// Configure logs/shops.log to receive all levels, but only
// those with `orders` and `payments` scope.
Log::config('shops', [
    'className' => 'File',
    'path' => LOGS,
    'levels' => [],
    'scopes' => ['orders', 'payments'],
    'file' => 'shops.log',
]);
```

```
// Configure logs/payments.log to receive all levels, but only
// those with `payments` scope.
Log::config('payments', [
    'className' => 'File',
    'path' => LOGS,
    'levels' => [],
    'scopes' => ['payments'],
    'file' => 'payments.log',
]);

Log::warning('this gets written only to shops.log', ['scope' => ['orders']]);
Log::warning('this gets written to both shops.log and payments.log', ['scope' => [
↪'payments']]);
```

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.

Log API

class `Cake\Log\`**Log**

A simple class for writing to logs.

static `Cake\Log\Log::`**config**(*$key*, *$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::**debug**(*$message*, *$scope = []*)

static Cake\Log\Log::**info**(*$message*, *$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. If `$msg` isn't a string it will be converted with `print_r` before being logged.

Using Monolog

Monolog is a popular logger for PHP. Since it implements the same interfaces as the CakePHP loggers, it is easy to use in your application as the default logger.

After installing Monolog using composer, configure the logger using the `Log::config()` method:

```
// config/bootstrap.php

use Monolog\Logger;
use Monolog\Handler\StreamHandler;

Log::config('default', function () {
    $log = new Logger('app');
    $log->pushHandler(new StreamHandler('path/to/your/combined.log'));
    return $log;
});

// 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::config('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 peristent 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)
    {
        return $schema->addField('name', 'string')
            ->addField('email', ['type' => 'string'])
            ->addField('body', ['type' => 'text']);
    }

    protected function _buildValidator(Validator $validator)
    {
        return $validator->add('name', 'length', [
                'rule' => ['minLength', 10],
                'message' => 'A name is required'
            ])->add('email', 'format', [
                'rule' => 'email',
                'message' => 'A valid email address is required',
```

```
            ]);
    }

    protected function _execute(array $data)
    {
        // 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.
- `_buildValidator` 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. We could have also used the `validate()` method to only validate the request data:

```
$isValid = $form->validate($this->request->getData());
```

Setting Form Values

In order to set the values for the fields of a modelless form, one can define the values using `$this->request->data()`, like in all other forms created by the FormHelper:

```
// 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')) {
            // Values from the User Model e.g.
            $this->request->data('name', 'John Doe');
            $this->request->data('email','john.doe@example.com');
        }

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

Values should only be defined if the request method is GET, otherwise you will overwrite your previous POST Data which might have been incorrect and not been saved.

Getting Form Errors

Once a form has been validated you can retrieve the errors from it:

```
$errors = $form->errors();
/* $errors contains
[
    'email' => ['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

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 a great user management, simple blog, or web services module 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 ultimately 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 application-specific properties (e.g. database connectivity parameters) which are defined and shared through the application's configuration.

In CakePHP 3.0 each plugin defines its 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[150] 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.

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.

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

[150] http://packagist.org

this file to locate plugins when they are loaded with `load()` or `loadAll()`. You generally won't need to edit this file by hand, as Composer and the `plugin-installer` package will manage it for you.

Loading a Plugin

After installing a plugin and setting up the autoloader, you should load the plugin. You can load plugins one by one, or all of them with a single method:

```
// In config/bootstrap.php
// Or in Application::bootstrap()

// Loads a single plugin
Plugin::load('ContactManager');

// Loads a plugin with a vendor namespace at top level.
Plugin::load('AcmeCorp/ContactManager');

// Loads all plugins at once
Plugin::loadAll();
```

`loadAll()` loads all plugins available, while allowing you to set certain settings for specific plugins. `load()` works similarly, but only loads the plugins you explicitly specify.

Note: `Plugin::loadAll()` won't load vendor namespaced plugins that are not defined in **vendor/cakephp-plugins.php**.

There is also a handy shell command to enable the plugin. Execute the following line:

```
bin/cake plugin load ContactManager
```

This will put the `Plugin::load('ContactManager');` snippet in the bootstrap for you.

Autoloading Plugin Classes

When using `bake` for creating a plugin or when installing a plugin using Composer, you don't typically need to make any changes to your application in order to make CakePHP recognize the classes that live inside it.

In any other cases you may need to modify your application's composer.json file to contain the following information:

```
"psr-4": {
    (...)
    "MyPlugin\\": "./plugins/MyPlugin/src",
    "MyPlugin\\Test\\": "./plugins/MyPlugin/tests"
}
```

If you are using vendor namespaces for your plugins, the namespace to path mapping should resemble the following:

```
"psr-4": {
    (...)
    "AcmeCorp\\Users\\": "./plugins/AcmeCorp/Users/src",
    "AcmeCorp\\Users\\Test\\": "./plugins/AcmeCorp/Users/tests"
}
```

Additionally, you will need to tell Composer to refresh its autoloading cache:

```
$ php composer.phar dumpautoload
```

If you are unable to use Composer for any reason, you can also use a fallback autoloading for your plugin:

```
Plugin::load('ContactManager', ['autoload' => true]);
```

Plugin Configuration

The `load()` and `loadAll()` methods can assist with plugin configuration and routing. Perhaps you want to load all plugins automatically while specifying custom routes and bootstrap files for certain plugins:

```
// In config/bootstrap.php,
// or in Application::bootstrap()

// Using loadAll()
Plugin::loadAll([
    'Blog' => ['routes' => true],
    'ContactManager' => ['bootstrap' => true],
    'WebmasterTools' => ['bootstrap' => true, 'routes' => true],
]);
```

Or you can load the plugins individually:

```
// Loading just the blog and include routes
Plugin::load('Blog', ['routes' => true]);

// Include bootstrap configuration/initializer file.
Plugin::load('ContactManager', ['bootstrap' => true]);
```

With either approach you no longer need to manually `include()` or `require()` a plugin's configuration or routes file – it happens automatically at the right time and place.

You can specify a set of defaults for `loadAll()` which will apply to every plugin that doesn't have a more specific configuration.

The following example will load the bootstrap file from all plugins, and additionally the routes from the Blog plugin:

```
Plugin::loadAll([
    ['bootstrap' => true],
    'Blog' => ['routes' => true]
]);
```

Note that all files specified should actually exist in the configured plugin(s) or PHP will give warnings for each file it cannot load. You can avoid potential warnings by using the `ignoreMissing` option:

```
Plugin::loadAll([
    ['ignoreMissing' => true, 'bootstrap' => true],
    'Blog' => ['routes' => true]
]);
```

When loading plugins, the plugin name used should match the namespace. For example, if you have a plugin with top level namespace `Users` you would load it using:

```
Plugin::load('User');
```

If you prefer to have your vendor name as top level and have a namespace like `AcmeCorp/Users`, then you would load the plugin as:

```
Plugin::load('AcmeCorp/Users');
```

This will ensure that classnames are resolved properly when using *plugin syntax*.

Most plugins will indicate the proper procedure for configuring them and setting up the database in their documentation. Some plugins will require more setup than others.

Using Plugins

You can reference a plugin's controllers, models, components, behaviors, and helpers by prefixing the name of the plugin before

For example, say you wanted to use the ContactManager plugin's ContactInfoHelper to output some pretty contact information in one of your views. In your controller, your `$helpers` array could look like this:

```
public $helpers = ['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);
```

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
            /Controller
                /Component
            /Model
                /Table
                /Entity
                /Behavior
            /View
                /Helper
            /Template
                /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. 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 'Template' 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 the bake shell.

In order to bake a plugin, use the following command:

```
bin/cake bake plugin ContactManager
```

Now you can bake using the same conventions which apply to the rest of your app. For example - baking controllers:

```
bin/cake bake controller --plugin ContactManager Contacts
```

Please refer to the chapter *Code Generation with Bake* 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 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;
use Cake\Routing\Router;

Router::plugin(
    'ContactManager',
    ['path' => '/contact-manager'],
    function ($routes) {
        $routes->get('/contacts', ['controller' => 'Contacts']);
        $routes->get('/contacts/:id', ['controller' => 'Contacts', 'action' => 'view
↪']);
        $routes->put('/contacts/:id', ['controller' => 'Contacts', 'action' => 'update
↪']);
    }
);
```

The above will connect default routes for your plugin. You can customize this file with more specific routes later on.

Before you can access your controllers, you'll need to ensure the plugin is loaded and the plugin routes are loaded. In your **config/bootstrap.php** add the following:

```
Plugin::load('ContactManager', ['routes' => true]);
```

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:

```
Router::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`.

New in version 3.5.0: `RouteBuilder::loadPlugin()` was added in 3.5.0

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 Configuration* for information on how to load plugin specific route files.

For plugins you did not create with bake, you will also need to edit the **composer.json** file to add your plugin to the autoload classes, this can be done as per the documentation *Autoloading Plugin Classes*.

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)
    {
        $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)
    {
        $this->hasMany('AltName', [
            'className' => 'ContactManager.AltName',
        ]);
    }
}
```

You can use `TableRegistry` to load your plugin tables using the familiar *plugin syntax*:

```
use Cake\ORM\TableRegistry;

$contacts = TableRegistry::get('ContactManager.Contacts');
```

Alternatively, from a controller context, you can use:

```
$this->loadModel('ContactsMangager.Contacts');
```

Plugin Views

Views behave exactly as they do in normal applications. Just place them in the right folder inside of the `plugins/[PluginName]/src/Template/` folder. For our ContactManager plugin, we'll need a view for our `ContactsController::index()` action, so let's include that as well:

```
// plugins/ContactManager/src/Template/Contacts/index.ctp:
<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]/src/Template/Layout`. To use a plugin layout in your controller you can do the following:

```
public $layout = '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 **src/Template/Plugin/[Plugin]/[Controller]/[view].ctp**. For the Contacts controller you could make the following file:

```
src/Template/Plugin/ContactManager/Contacts/index.ctp
```

Creating this file would allow you to override **plugins/ContactManager/src/Template/Contacts/index.ctp**.

If your plugin is in a composer dependency (i.e. 'TheVendor/ThePlugin'), the path to the 'index' view of the Custom controller will be:

```
src/Template/Plugin/TheVendor/ThePlugin/Custom/index.ctp
```

Creating this file would allow you to override **vendor/thevendor/theplugin/src/Template/Custom/index.ctp**.

If the plugin implements a routing prefix, you must include the routing prefix in your application template overrides.

If the 'ContactManager' plugin implemented an 'admin' prefix the overridng path would be:

```
src/Template/Plugin/ContactManager/Admin/ContactManager/index.ctp
```

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 *View\Helper\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');

// Generates a URL of /contact_manager/img/logo.jpg
echo $this->Html->image('ContactManager.logo');
```

Plugin assets are served using the `AssetFilter` dispatcher filter 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 regular 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()
{
    parent::initialize();
    $this->loadComponent('ContactManager.Example');
}
```

The same technique applies to Helpers and Behaviors.

Expand Your Plugin

This example created a good start for a plugin, but there is a lot more that you can do. As a general rule, anything you can do with your application you can do inside of a plugin as well.

Go ahead - include some third-party libraries in 'vendor', add some new shells to the cake console, and don't forget to create test cases so your plugin users can automatically test your plugin's functionality!

In our ContactManager example we might create add/remove/edit/delete actions in the ContactsController, implement validation in the Contact model, and implement the functionality one might expect when managing their contacts. It's up to you to decide what to implement in your plugins. Just don't forget to share your code with the community so that everyone can benefit from your awesome, reusable components!

Publish Your Plugin

Make sure you add your plugin to plugins.cakephp.org[151]. This way other people can use it as composer dependency. You can also propose your plugin to the awesome-cakephp list[152].

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.

[151] https://plugins.cakephp.org
[152] https://github.com/FriendsOfCake/awesome-cakephp

CHAPTER 25

REST

Many newer application programmers are realizing the need to open their core functionality to a greater audience. Providing easy, unfettered access to your core API can help get your platform accepted, and allows for mashups and easy integration with other systems.

While other solutions exist, REST is a great way to provide easy access to the logic you've created in your application. It's simple, usually XML-based (we're talking simple XML, nothing like a SOAP envelope), and depends on HTTP headers for direction. Exposing an API via REST in CakePHP is simple.

The Simple Setup

The fastest way to get up and running with REST is to add a few lines to setup *resource routes* in your config/routes.php file.

Once the router has been set up to map REST requests to certain controller actions, we can move on to creating the logic in our controller actions. A basic controller might look something like this:

```
// src/Controller/RecipesController.php
class RecipesController extends AppController
{

    public function initialize()
    {
        parent::initialize();
        $this->loadComponent('RequestHandler');
    }

    public function index()
    {
        $recipes = $this->Recipes->find('all');
        $this->set([
            'recipes' => $recipes,
            '_serialize' => ['recipes']
        ]);
    }

    public function view($id)
    {
```

```
        $recipe = $this->Recipes->get($id);
        $this->set([
            'recipe' => $recipe,
            '_serialize' => ['recipe']
        ]);
    }

    public function add()
    {
        $recipe = $this->Recipes->newEntity($this->request->getData());
        if ($this->Recipes->save($recipe)) {
            $message = 'Saved';
        } else {
            $message = 'Error';
        }
        $this->set([
            'message' => $message,
            'recipe' => $recipe,
            '_serialize' => ['message', 'recipe']
        ]);
    }

    public function edit($id)
    {
        $recipe = $this->Recipes->get($id);
        if ($this->request->is(['post', 'put'])) {
            $recipe = $this->Recipes->patchEntity($recipe, $this->request->getData());
            if ($this->Recipes->save($recipe)) {
                $message = 'Saved';
            } else {
                $message = 'Error';
            }
        }
        $this->set([
            'message' => $message,
            '_serialize' => ['message']
        ]);
    }

    public function delete($id)
    {
        $recipe = $this->Recipes->get($id);
        $message = 'Deleted';
        if (!$this->Recipes->delete($recipe)) {
            $message = 'Error';
        }
        $this->set([
            'message' => $message,
            '_serialize' => ['message']
        ]);
    }
}
```

RESTful controllers often use parsed extensions to serve up different views based on different kinds of requests. Since we're dealing with REST requests, we'll be making XML views. You can make JSON views using CakePHP's built-in *JSON and XML views*. By using the built in *XmlView* we can define a `_serialize` view variable. This special view variable is used to define which view variables `XmlView` should serialize into XML.

If we wanted to modify the data before it is converted into XML we should not define the `_serialize` view variable, and instead use template files. We place the REST views for our RecipesController inside **src/Template/Recipes/xml**. We can also use the `Xml` for quick-and-easy XML output in those views. Here's what our index view might look like:

```
// src/Template/Recipes/xml/index.ctp
// Do some formatting and manipulation on
// the $recipes array.
$xml = Xml::fromArray(['response' => $recipes]);
echo $xml->asXML();
```

When serving up a specific content type using *`Cake\Routing\Router::extensions()`*, CakePHP automatically looks for a view helper that matches the type. Since we're using XML as the content type, there is no built-in helper, however if you were to create one it would automatically be loaded for our use in those views.

The rendered XML will end up looking something like this:

```
<recipes>
    <recipe>
        <id>234</id>
        <created>2008-06-13</created>
        <modified>2008-06-14</modified>
        <author>
            <id>23423</id>
            <first_name>Billy</first_name>
            <last_name>Bob</last_name>
        </author>
        <comment>
            <id>245</id>
            <body>Yummy yummmy</body>
        </comment>
    </recipe>
    ...
</recipes>
```

Creating the logic for the edit action is a bit trickier, but not by much. Since you're providing an API that outputs XML, it's a natural choice to receive XML as input. Not to worry, the `Cake\Controller\Component\RequestHandler` and *`Cake\Routing\Router`* classes make things much easier. If a POST or PUT request has an XML content-type, then the input is run through CakePHP's `Xml` class, and the array representation of the data is assigned to `$this->request->getData()`. Because of this feature, handling XML and POST data in parallel is seamless: no changes are required to the controller or model code. Everything you need should end up in `$this->request->getData()`.

Accepting Input in Other Formats

Typically REST applications not only output content in alternate data formats, but also accept data in different formats. In CakePHP, the *`RequestHandlerComponent`* helps facilitate this. By default, it will decode any incoming JSON/XML input data for POST/PUT requests and supply the array version of that data in `$this->request->getData()`. You can also wire in additional deserializers for alternate formats if you need them, using `RequestHandler::addInputType()`.

RESTful Routing

CakePHP's Router makes connecting RESTful resource routes easy. See the section on *Creating RESTful Routes* for more information.

CHAPTER 26

Security

CakePHP provides you some tools to secure your application. The following sections cover those tools:

Security

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.

This method will use either openssl[154] or mcrypt[155] based on what is available on your system. Data encrypted in one implementation is portable to the other.

> **Warning:** The mcrypt[156] extension has been deprecated in PHP7.1
>
> [156] http://php.net/mcrypt

This method should **never** be used to store passwords. Instead you should use the one way hashing methods provided by *`Utility\Security::hash()`*. 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/3.x/class-Cake.Utility.Security.html
[154] http://php.net/openssl
[155] http://php.net/mcrypt

If you do not supply an HMAC salt, the `Security.salt` value will be used. Encrypted values can be decrypted using *`Cake\Utility\Security::decrypt()`*.

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.

Choosing a Specific Crypto Implementation

If you are upgrading an application from CakePHP 2.x, data encrypted in 2.x is not compatible with openssl. This is because the encrypted data is not fully AES compliant. If you don't want to go through the trouble of re-encrypting your data, you can force CakePHP to use `mcrypt` using the `engine()` method:

```
// In config/bootstrap.php
use Cake\Utility\Crypto\Mcrypt;

Security::engine(new Mcrypt());
```

The above will allow you to seamlessly read data from older versions of CakePHP, and encrypt new data to be compatible with OpenSSL.

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 algorithmn 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.

New in version 3.2.3: The randomBytes method was added in 3.2.3.

CHAPTER 27

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* before CakePHP's session handler expires the session.
- `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 'CAKEPHP'.
- `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
    ]
]);
```

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 without visiting
        // any page on the site.
        '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 on the site.

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 `Network\Session` namespace. For example, if you had an `AppSessionHandler` class, the file should be **src/Network/Session/AppSessionHandler.php**, and the class name should be `App\Network\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[157] 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'
    ]
]
```

[157] https://github.com/cakephp/app

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 APC, Memcached, or XCache. 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'`.

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/Network/Session/ComboSession.php**. The class should look something like:

```
namespace App\Network\Session;

use Cake\Cache\Cache;
use Cake\Core\Configure;
use Cake\Network\Session\DatabaseSession;
```

```
class ComboSession extends DatabaseSession
{
    public $cacheKey;

    public function __construct()
    {
        $this->cacheKey = Configure::read('Session.handler.cache');
        parent::__construct();
    }

    // Read data from the session.
    public function read($id)
    {
        $result = Cache::read($id, $this->cacheKey);
        if ($result) {
            return $result;
        }
        return parent::read($id);
    }

    // Write data into the session.
    public function write($id, $data)
    {
        Cache::write($id, $data, $this->cacheKey);
        return parent::write($id, $data);
    }

    // Destroy a session.
    public function destroy($id)
    {
        Cache::delete($id, $this->cacheKey);
        return parent::destroy($id);
    }

    // Removes expired sessions.
    public function gc($expires = null)
    {
        return Cache::gc($this->cacheKey) && 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. Using this session handler is also easy. In your **app.php** make the session block look like the following:

```
'Session' => [
    'defaults' => 'database',
    'handler' => [
        '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

In addition to the basic session object, you can also use the *`Cake\View\Helper\SessionHelper`* to interact with the session in your views. A basic example of session usage would be:

```
$name = $this->request->session()->read('User.name');

// If you are accessing the session multiple times,
// you will probably want a local variable.
$session = $this->request->session();
$name = $session->read('User.name');
```

Reading & Writing Session Data

`Session::`**`read`**(*$key*)

You can read values from the session using `Hash::extract()` compatible syntax:

```
$session->read('Config.language');
```

`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 `AuthComponent` 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 *Flash* and *Flash*

CHAPTER 28

Testing

CakePHP comes with comprehensive testing support built-in. CakePHP comes with integration for PHPUnit[158]. 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[159] or Composer[160].

Install PHPUnit with Composer

To install PHPUnit with Composer:

```
$ php composer.phar require --dev phpunit/phpunit:"^5.7|^6.0"

// Before CakePHP 3.4.1
$ php composer.phar require --dev phpunit/phpunit:"<6.0"
```

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
```

Using the PHAR File

After you have downloaded the **phpunit.phar** file, you can use it to run your tests:

[158] http://phpunit.de
[159] http://phpunit.de/#download
[160] http://getcomposer.org

```
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[161].

Test Database Setup

Remember to have debug enabled in your **config/app.php** file before running any tests. Before running any tests you should be sure to add a `test` datasource configuration to **config/app.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
```

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:

[161] http://phpunit.de/manual/current/en/installation.html#installation.phar.windows

```
$ 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 colours.

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.

New in version 3.4.1: Support for PHPUnit 6 was addded. If you're using a PHPUnit version lower than 5.7.0, your tests classes should either extends CakePHP's classes or `PHPUnit_Framework_TestCase`.

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()
    {

    }

    public function testBar()
    {

    }
}
```

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()
{
    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 *Core\Configure* and, storing the paths in *Core\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()
{
    $result = $this->Progress->bar(90);
    $this->assertContains('width: 90%', $result);
    $this->assertContains('progress-bar', $result);

    $result = $this->Progress->bar(33.3333333);
    $this->assertContains('width: 33%', $result);
}
```

The above test is a simple one but shows the potential benefit of using test cases. We use `assertContains()` 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 are easy to re-run whenever you make a change. This helps prevent the creation of new bugs.

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:

```
# composer installs
$ vendor/bin/phpunit

# phar file
php phpunit.phar
```

If you have cloned the CakePHP source from GitHub[162] 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 --dev
```

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

# Using composer installed phpunit
../vendor/bin/phpunit

# Using phar file
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.

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:

[162] https://github.com/cakephp/cakephp

```
$ 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`.

If you are using PHP 5.6.0 or greater, you can use phpdbg[163] 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 Test Suite">
        <directory>./tests/TestCase</directory>
    </testsuite>

    <!-- Add your plugin suites -->
    <testsuite name="Forum plugin">
        <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": {
    "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*.

[163] http://phpdbg.com/

Fixtures

When testing code that depends on models and the database, one can use **fixtures** as a way to generate temporary data tables loaded with sample data that can be used by the test. The benefit of using fixtures is that your test has no chance of disrupting live application data. In addition, you can begin testing your code prior to actually developing live content for an application.

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 fixture based test case:

1. Creates tables for each of the fixtures needed.
2. Populates tables with data, if data is provided in fixture.
3. Runs test methods.
4. Empties the fixture tables.
5. Removes fixture tables from database.

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'.

Creating Fixtures

When creating a fixture you will mainly define two things: how the table is created (which fields are part of the table), and which records will be initially populated to the table. 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 $fields = [
          'id' => ['type' => 'integer'],
          'title' => ['type' => 'string', 'length' => 255, 'null' => false],
          'body' => 'text',
          'published' => ['type' => 'integer', 'default' => '0', 'null' => false],
          'created' => 'datetime',
          'modified' => 'datetime',
          '_constraints' => [
            'primary' => ['type' => 'primary', 'columns' => ['id']]
          ]
      ];
```

```
    public $records = [
        [
            'title' => 'First Article',
            'body' => 'First Article Body',
            'published' => '1',
            'created' => '2007-03-18 10:39:23',
            'modified' => '2007-03-18 10:41:31'
        ],
        [
            'title' => 'Second Article',
            'body' => 'Second Article Body',
            'published' => '1',
            'created' => '2007-03-18 10:41:23',
            'modified' => '2007-03-18 10:43:31'
        ],
        [
            'title' => 'Third Article',
            'body' => 'Third Article Body',
            '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 use `$fields` to specify which fields will be part of this table, and how they are defined. The format used to define these fields is the same used with `Cake\Database\Schema\Table`. The keys available for table definition are:

type CakePHP internal data type. Currently supported:

- `string`: maps to `VARCHAR` or `CHAR`
- `uuid`: maps to `UUID`
- `text`: maps to `TEXT`
- `integer`: maps to `INT`
- `biginteger`: maps to `BIGINTEGER`
- `decimal`: maps to `DECIMAL`
- `float`: maps to `FLOAT`
- `datetime`: maps to `DATETIME`
- `timestamp`: maps to `TIMESTAMP`
- `time`: maps to `TIME`
- `date`: maps to `DATE`

- `binary`: maps to `BLOB`

fixed Used with string types to create CHAR columns in platforms that support them.

length Set to the specific length the field should take.

precision Set the number of decimal places used on float & decimal fields.

null Set to either `true` (to allow NULLs) or `false` (to disallow NULLs).

default Default value the field takes.

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 $records array must have a key for **every** field specified in the `$fields` array. If a field for a particular record needs to have a `null` value, just specify the value of that key as `null`.

Dynamic Data and Fixtures

Since records for a fixture are declared as a class property, you cannot use functions or other dynamic data to define fixtures. To solve this problem, you can define `$records` in the `init()` function of your fixture. For example if you wanted all the created and modified timestamps to reflect today's date you could do the following:

```
namespace App\Test\Fixture;

use Cake\TestSuite\Fixture\TestFixture;

class ArticlesFixture extends TestFixture
{

    public $fields = [
        'id' => ['type' => 'integer'],
        'title' => ['type' => 'string', 'length' => 255, 'null' => false],
        'body' => 'text',
        'published' => ['type' => 'integer', 'default' => '0', 'null' => false],
        'created' => 'datetime',
        'modified' => 'datetime',
        '_constraints' => [
            'primary' => ['type' => 'primary', 'columns' => ['id']],
        ]
    ];

    public function init()
    {
        $this->records = [
            [
                'title' => 'First Article',
                'body' => 'First Article Body',
                'published' => '1',
                'created' => date('Y-m-d H:i:s'),
                'modified' => date('Y-m-d H:i:s'),
            ],
        ];
        parent::init();
    }
}
```

When overriding `init()` remember to always call `parent::init()`.

Importing Table Information

Defining the schema in fixture files can be really handy when creating plugins or libraries or if you are creating an application that needs to be portable between database vendors. Redefining the schema in fixtures can become difficult to maintain in larger applications. Because of this CakePHP provides the ability to import the schema from an existing connection and use the reflected table definition to create the table definition used in the test suite.

Let's start with an example. Assuming you have a table named articles available in your application, change the example fixture given in the previous section (**tests/Fixture/ArticlesFixture.php**) to:

```
class ArticlesFixture extends TestFixture
{
    public $import = ['table' => 'articles'];
}
```

If you want to use a different connection use:

```
class ArticlesFixture extends TestFixture
{
    public $import = ['table' => 'articles', 'connection' => 'other'];
}
```

New in version 3.1.7.

Usually, you have a Table class along with your fixture, as well. You can also use that to retrieve the table name:

```
class ArticlesFixture extends TestFixture
{
    public $import = ['model' => 'Articles'];
}
```

Since this uses `TableRegistry::get()`, it also supports plugin syntax.

You can naturally import your table definition from an existing model/table, but have your records defined directly on the fixture as it was shown on previous section. For example:

```
class ArticlesFixture extends TestFixture
{
    public $import = ['table' => 'articles'];
    public $records = [
        [
          'title' => 'First Article',
          'body' => 'First Article Body',
          'published' => '1',
          'created' => '2007-03-18 10:39:23',
          'modified' => '2007-03-18 10:41:31'
        ],
        [
          'title' => 'Second Article',
          'body' => 'Second Article Body',
          'published' => '1',
          'created' => '2007-03-18 10:41:23',
          'modified' => '2007-03-18 10:43:31'
        ],
        [
          'title' => 'Third Article',
          'body' => 'Third Article Body',
          'published' => '1',
          'created' => '2007-03-18 10:43:23',
```

```
            'modified' => '2007-03-18 10:45:31'
        ]
    ];
}
```

Finally, it's possible to not load/create any schema in a fixture. This is useful if you already have a test database setup with all the empty tables created. By defining neither `$fields` nor `$import`, a fixture will only insert its records and truncate the records on each test method.

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
{
    public $fixtures = ['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
{
    public $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 control when your fixtures are loaded by setting `Cake\TestSuite\TestCase::$autoFixtures` to `false` and later load them using `Cake\TestSuite\TestCase::loadFixtures()`:

```
class ArticlesTest extends TestCase
{
    public $fixtures = ['app.articles', 'app.comments'];
    public $autoFixtures = false;

    public function testMyFunction()
    {
        $this->loadFixtures('Articles', 'Comments');
    }
}
```

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
{
    public $fixtures = ['app.blog/articles', 'app.blog/comments'];
}
```

In the above example, both fixtures would be loaded from `tests/Fixture/blog/`.

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;

class ArticlesTable extends Table
{

    public function findPublished(Query $query, array $options)
    {
        $query->where([
            $this->alias() . '.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\ORM\TableRegistry;
use Cake\TestSuite\TestCase;

class ArticlesTableTest extends TestCase
{
    public $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\ORM\TableRegistry;
use Cake\TestSuite\TestCase;

class ArticlesTableTest extends TestCase
{
    public $fixtures = ['app.articles'];

    public function setUp()
    {
        parent::setUp();
```

```
        $this->Articles = TableRegistry::get('Articles');
    }

    public function testFindPublished()
    {
        $query = $this->Articles->find('published');
        $this->assertInstanceOf('Cake\ORM\Query', $query);
        $result = $query->hydrate(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.

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()
{
    $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:

```
TableRegistry::clear();
```

Controller Integration Testing

While you can test controller classes in a similar fashion to Helpers, Models, and Components, CakePHP offers a specialized `IntegrationTestCase` class. Using this class as the base class for 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 makes it easy 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 $this->redirect(['action' => 'index']);
            }
        }
        if (!empty($short)) {
            $result = $this->Articles->find('all', [
                'fields' => ['id', 'title']
            ]);
        } else {
            $result = $this->Articles->find();
        }

        $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\ORM\TableRegistry;
use Cake\TestSuite\IntegrationTestCase;

class ArticlesControllerTest extends IntegrationTestCase
{
    public $fixtures = ['app.articles'];

    public function testIndex()
    {
        $this->get('/articles');

        $this->assertResponseOk();
        // More asserts.
    }

    public function testIndexQueryData()
    {
        $this->get('/articles?page=1');

        $this->assertResponseOk();
```

```
        // More asserts.
    }

    public function testIndexShort()
    {
        $this->get('/articles/index/short');

        $this->assertResponseOk();
        $this->assertResponseContains('Articles');
        // More asserts.
    }

    public function testIndexPostData()
    {
        $data = [
            'user_id' => 1,
            'published' => 1,
            'slug' => 'new-article',
            'title' => 'New Article',
            'body' => 'New Body'
        ];
        $this->post('/articles', $data);

        $this->assertResponseSuccess();
        $articles = TableRegistry::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 `IntegrationTestCase` 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 `IntegrationTestCase` or PHPUnit to ensure your request had the correct side-effects.

New in version 3.5.0: `options()` and `head()` were added in 3.5.0.

Setting up the Request

The `IntegrationTestCase` class comes with a number of helpers to make it easy 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 That Require Authentication

If you are using `AuthComponent` you will need to stub out the session data that AuthComponent uses to validate a user's identity. You can use helper methods in `IntegrationTestCase` to do this. Assuming you had an `ArticlesController` that contained an add method, and that add method required authentication, you could write the following tests:

```
public function testAddUnauthenticatedFails()
{
    // No session data set.
    $this->get('/articles/add');

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

public function testAddAuthenticated()
{
    // Set session data
    $this->session([
        'Auth' => [
            'User' => [
                'id' => 1,
                'username' => 'testing',
                // other keys.
            ]
        ]
    ]);
    $this->get('/articles/add');

    $this->assertResponseOk();
    // Other assertions.
}
```

Testing Stateless Authentication and APIs

To test APIs that use stateless authentication, such as Basic authentication, you can configure the request to inject environment conditions or headers that simulate actual authentication request headers.

When testing Basic or Digest Authentication, you can add the environment variables that PHP creates[164] automatically. These environment variables used in the authentication adapter outlined in *Using Basic Authentication*:

[164] http://php.net/manual/en/features.http-auth.php

```
public function testBasicAuthentication()
{
    $this->configRequest([
        'environment' => [
            'PHP_AUTH_USER' => 'username',
            'PHP_AUTH_PW' => 'password',
        ]
    ]);

    $this->get('/api/posts');
    $this->assertResponseOk();
}
```

If you are testing other forms of authentication, such as OAuth2, you can set the Authorization header directly:

```
public function testOauthToken()
{
    $this->configRequest([
        'headers' => [
            'authorization' => 'Bearer: oauth-token'
        ]
    ]);

    $this->get('/api/posts');
    $this->assertResponseOk();
}
```

The headers key in `configRequest()` can be used to configure any additional HTTP headers needed for an action.

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()
{
    $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']
]);
```

New in version 3.1.2: The `enableCsrfToken()` and `enableSecurityToken()` methods were added in 3.1.2

Integration Testing PSR-7 Middleware

Integration testing can also be used to test your entire PSR-7 application and /controllers/middleware. By default `IntegrationTestCase` will auto-detect the presence of an `App\Application` class and automatically enable integration testing of your Application. You can toggle this behavior with the `useHttpServer()` method:

```
public function setUp()
{
    // Enable PSR-7 integration testing.
    $this->useHttpServer(true);

    // Disable PSR-7 integration testing.
    $this->useHttpServer(false);
}
```

You can customize the application class name used, and the constructor arguments, by using the `configApplication()` method:

```
public function setUp()
{
    $this->configApplication('App\App', [CONFIG]);
}
```

After enabling the PSR-7 mode, and possibly configuring your application class, you can use the remaining `IntegrationTestCase` features as normal.

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.

New in version 3.3.0: PSR-7 Middleware and the `useHttpServer()` method were added in 3.3.0.

Testing with Encrypted Cookies

If you use the *Cake\Controller\Component\CookieComponent* in your controllers, your cookies are likely encrypted. As of 3.1.7, CakePHP provides helper methods for interacting with 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('/bookmarks/index');

$this->assertCookieEncrypted('An updated value', 'my_cookie');
```

New in version 3.1.7: `assertCookieEncrypted` and `cookieEncrypted` were added in 3.1.7.

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:

```
$this->enableRetainFlashMessages();
$this->get('/bookmarks/delete/9999');

$this->assertSession('That bookmark does not exist', 'Flash.flash.0.message');
```

New in version 3.4.7: `enableRetainFlashMessages()` was added in 3.4.7

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:

```
class MarkersController extends AppController
{
    public function initialize()
    {
        parent::initialize();
        $this->loadComponent('RequestHandler');
    }

    public function view($id)
    {
        $marker = $this->Markers->get($id);
        $this->set([
            '_serialize' => ['marker'],
            'marker' => $marker,
        ]);
    }
}
```

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
{

    public function testGet()
    {
        $this->configRequest([
            'headers' => ['Accept' => 'application/json']
        ]);
        $result = $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, $this->_response->body());
    }
}
```

We use the `JSON_PRETTY_PRINT` option as CakePHP's built in JsonView will use that option when `debug` is enabled.

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()
{
    $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.

New in version 3.5.0.

Assertion methods

The `IntegrationTestCase` class 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, e.g. 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 not empty response content
$this->assertResponseNotEmpty();

// Assert empty response content
$this->assertResponseEmpty();

// Assert response content
$this->assertResponseEquals('Yeah!');

// Assert partial response content
$this->assertResponseContains('You won!');
$this->assertResponseNotContains('You lost!');

// 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');

// Assert view variables
$user =  $this->viewVariable('user');
$this->assertEquals('jose', $user->username);

// Assert cookies in the response
$this->assertCookie('1', 'thingid');

// 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[165] and those found in PHPUnit[166].

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()
    {
        $this->_compareBasePath = APP . 'tests' . DS . 'comparisons' . DS;
        parent::setUp();
    }

    public function testExample()
    {
        $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

[165] https://api.cakephp.org/3.x/class-Cake.TestSuite.TestCase.html
[166] https://phpunit.de/manual/current/en/appendixes.assertions.html

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

To make testing console applications easier, CakePHP comes with a `ConsoleIntegrationTestCase` class that can be used to test console applications and assert against their results.

New in version 3.5.0: The `ConsoleIntegrationTestCase` was added.

To get started testing your console application, create a test case that extends `Cake\TestSuite\ConsoleIntegrationTestCase`. This class 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.

Let's start with a very simple shell, located in **src/Shell/MyConsoleShell.php**:

```
namespace App\Shell;

use Cake\Console\ConsoleOptionParser;
use Cake\Console\Shell;

class MyConsoleShell extends Shell
{
    public function getOptionParser()
    {
        $parser = new ConsoleOptionParser();
        $parser->setDescription('My cool console app');

        return $parser;
    }
}
```

To write an integration test for this shell, we would create a test case in **tests/TestCase/Shell/MyConsoleShellTest.php** that extends `Cake\TestSuite\ConsoleIntegrationTestCase`. This shell doesn't do much at the moment, but let's just test that our shell's description is displayed in `stdout`:

```
namespace App\Test\TestCase\Shell;

use Cake\TestSuite\ConsoleIntegrationTestCase;

class MyConsoleShellTest extends ConsoleIntegrationTestCase
```

```
{
    public function testDescriptionOutput()
    {
        $this->exec('my_console');
        $this->assertOutputContains('My cool console app');
    }
}
```

Our test passes! While this is very trivial example, it shows that creating an integration test case for console applications is quite easy. Let's continue by adding some subcommands and options to our shell:

```
namespace App\Shell;

use Cake\Console\ConsoleOptionParser;
use Cake\I18n\FrozenTime;

class MyConsoleShell extends Shell
{
    public function getOptionParser()
    {
        $parser = new ConsoleOptionParser();

        $updateModifiedParser = new ConsoleOptionParser();
        $updateModifiedParser->addArgument('table', [
            'help' => 'Table to update',
            'required' => true
        ]);

        $parser
            ->setDescription('My cool console app')
            ->addSubcommand('updateModified', [
                'parser' => $updateModifiedParser
            ]);

        return $parser;
    }

    public function updateModified()
    {
        $table = $this->args[0];
        $this->loadModel($table);
        $this->{$table}->query()
            ->update()
            ->set([
                'modified' => new FrozenTime()
            ])
            ->execute();
    }
}
```

This is a more complete shell that has a subcommand with its own parser. Let's test the `updateModified` subcommand. Modify your test case to the following snippet of code:

```
namespace Cake\Test\TestCase\Shell;

use Cake\Console\Shell;
use Cake\I18n\FrozenTime;
use Cake\ORM\TableRegistry;
```

```
use Cake\TestSuite\ConsoleIntegrationTestCase;

class MyConsoleShellTest extends ConsoleIntegrationTestCase
{

    public $fixtures = [
        // assumes you have a UsersFixture
        'app.users'
    ];

    public function testDescriptionOutput()
    {
        $this->exec('my_console');
        $this->assertOutputContains('My cool console app');
    }

    public function testUpdateModified()
    {
        $now = new FrozenTime('2017-01-01 00:00:00');
        FrozenTime::setTestNow($now);

        $this->loadFixtures('Users');

        $this->exec('my_console update_modified Users');
        $this->assertExitCode(Shell::CODE_SUCCESS);

        $user = TableRegistry::get('Users')->get(1);
        $this->assertSame($user->modified->timestamp, $now->timestamp);

        FrozenTime::setTestNow(null);
    }
}
```

As you can see from the `testUpdateModified` method, we are testing that our `update_modified` subcommand updates the table that we are passing as the first argument. First, we assert that the shell exited with the proper status code, `0`. Then we check that our subcommand 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 shells with the `Cake\TestSuite\ConsoleIntegrationTestCase` class 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 shell, let's add an interactive subcommand. Update the shell class to the following:

```
namespace App\Shell;

use Cake\Console\ConsoleOptionParser;
use Cake\Console\Shell;
use Cake\I18n\FrozenTime;

class MyConsoleShell extends Shell
{
```

```
    public function getOptionParser()
    {
        $parser = new ConsoleOptionParser();

        $updateModifiedParser = new ConsoleOptionParser();
        $updateModifiedParser->addArgument('table', [
            'help' => 'Table to update',
            'required' => true
        ]);

        $parser
            ->setDescription('My cool console app')
            ->addSubcommand('updateModified', [
                'parser' => $updateModifiedParser
            ])
            // add a new subcommand
            ->addSubcommand('bestFramework');

        return $parser;
    }

    public function updateModified()
    {
        $table = $this->args[0];
        $this->loadModel($table);
        $this->{$table}->query()
            ->update()
            ->set([
                'modified' => new FrozenTime()
            ])
            ->execute();
    }

    // create this interactive subcommand
    public function bestFramework()
    {
        $this->out('Hi there!');

        $framework = $this->in('What is the best PHP framework?');
        if ($framework !== 'CakePHP') {
            $this->err("I disagree that '$framework' is the best.");
            $this->_stop(Shell::CODE_ERROR);
        }

        $this->out('I agree!');
    }
}
```

Now that we have an interactive subcommand, we can add a test case that tests that we receive the proper response, and one that tests that we receive an incorrect response. Add the following methods to **tests/TestCase/Shell/MyConsoleShellTest.php**:

```
public function testBestFramework()
{
    $this->exec('my_console best_framework', [
        'CakePHP'
    ]);
    $this->assertExitCode(Shell::CODE_SUCCESS);
```

```
    $this->assertOutputContains('I agree!');
}

public function testBestFrameworkWrongAnswer()
{
    $this->exec('my_console best_framework', [
        'my homemade framework'
    ]);
    $this->assertExitCode(Shell::CODE_ERROR);
    $this->assertErrorRegExp("/I disagree that \'(.+)\' is the best\./");
}
```

As you can see from the `testBestFramework`, it responds to the first input request with "CakePHP". Since this is the correct answer according to our subcommand, the shell will exit successfully after outputting a response.

The second test case, `testBestFrameworkWrongAnswer`, provides an incorrect answer which causes our shell to fail and exit with `1`. We also assert that `stderr` was given our error, which includes the name of the incorrect answer.

Testing the CommandRunner

To test shells that are dispatched using the `CommandRunner` class, enable it in your test case with the following method:

```
$this->useCommandRunner();
```

New in version 3.5.0: The `CommandRunner` class was added.

Assertion methods

The `Cake\TestSuite\ConsoleIntegrationTestCase` class provides a number of assertion methods that make it easy to assert against console output:

```
// assert that the shell 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);
```

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 `IntegrationTestCase` you can inspect the rendered view content by setting the `return` option to 'view'. While it is possible to test view content

using IntegrationTestCase, a more robust and maintainable integration/view testing can be accomplished using tools like Selenium webdriver[167].

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(Event $event)
    {
        $this->setController($event->getSubject());
    }

    public function adjust($length = 'short')
    {
        switch ($length) {
            case 'long':
                $this->controller->paginate['limit'] = 100;
            break;
            case 'medium':
                $this->controller->paginate['limit'] = 50;
            break;
            default:
                $this->controller->paginate['limit'] = 20;
            break;
        }
    }
}
```

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;
```

[167] http://seleniumhq.org

```
use Cake\TestSuite\TestCase;

class PagematronComponentTest extends TestCase
{

    public $component = null;
    public $controller = null;

    public function setUp()
    {
        parent::setUp();
        // Setup our component and fake test controller
        $request = new ServerRequest();
        $response = new Response();
        $this->controller = $this->getMockBuilder('Cake\Controller\Controller')
            ->setConstructorArgs([$request, $response])
            ->setMethods(null)
            ->getMock();
        $registry = new ComponentRegistry($this->controller);
        $this->component = new PagematronComponent($registry);
        $event = new Event('Controller.startup', $this->controller);
        $this->component->startup($event);
    }

    public function testAdjust()
    {
        // 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()
    {
        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($amount)
    {
        return 'USD ' . number_format($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;
use Cake\View\View;

class CurrencyRendererHelperTest extends TestCase
{

    public $helper = null;

    // Here we instantiate our helper
    public function setUp()
    {
        parent::setUp();
        $View = new View();
        $this->helper = new CurrencyRendererHelper($View);
    }

    // Testing the usd() function
    public function testUsd()
    {
        $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($order)
    {
        if ($this->save($order)) {
            // moved cart removal to CartsTable
            $event = new Event('Model.Order.afterPlace', $this, [
                'order' => $order
            ]);
            $this->eventManager()->dispatch($event);
            return true;
        }
        return false;
    }
}

class CartsTable extends Table
{

    public function implementedEvents()
    {
        return [
            'Model.Order.afterPlace' => 'removeFromCart'
        ];
    }

    public function removeFromCart(Event $event)
    {
        $order = $event->getData('order');
        $this->delete($order->cart_id);
    }
}
```

Note: 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\ORM\TableRegistry;
use Cake\TestSuite\TestCase;

class OrdersTableTest extends TestCase
```

```
{

    public $fixtures = ['app.orders'];

    public function setUp()
    {
        parent::setUp();
        $this->Orders = TableRegistry::get('Orders');
        // enable event tracking
        $this->Orders->eventManager()->setEventList(new EventList());
    }

    public function testPlace()
    {
        $order = new Order([
            'user_id' => 1,
            'item' => 'Cake',
            'quantity' => 42,
        ]);

        $this->assertTrue($this->Orders->place($order));

        $this->assertEventFired('Model.Order.afterPlace', $this->Orders->
↪eventManager());
        $this->assertEventFiredWith('Model.Order.afterPlace', 'order', $order, $this->
↪Orders->eventManager());
    }
}
```

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);
```

New in version 3.2.11: Event tracking, `assertEventFired()`, and `assertEventFiredWith` were added.

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.blog_posts`:

```
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/
    public $fixtures = ['plugin.blog.blog_posts'];

    public function testSomething()
    {
        // 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.

Before you can use fixtures you should double check that your `phpunit.xml` contains the fixture listener:

```
<!-- Setup a listener for fixtures -->
<listeners>
    <listener
    class="\Cake\TestSuite\Fixture\FixtureInjector"
    file="./vendor/cakephp/cakephp/src/TestSuite/Fixture/FixtureInjector.php">
        <arguments>
            <object class="\Cake\TestSuite\Fixture\FixtureManager" />
        </arguments>
    </listener>
</listeners>
```

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. Cell

While `<name>` should be the name of the object you want to bake a test skeleton for.

Integration with Jenkins

Jenkins[168] is a continuous integration server, that can help you automate the running of your test cases. This helps ensure that all your tests stay passing and your application is always ready.

Integrating a CakePHP application with Jenkins is fairly straightforward. The following assumes you've already installed Jenkins on *nix system, and are able to administer it. You also know how to create jobs, and run builds. If you are unsure of any of these, refer to the Jenkins documentation[169] .

Create a Job

Start off by creating a job for your application, and connect your repository so that jenkins can access your code.

Add Test Database Config

Using a separate database just for Jenkins is generally a good idea, as it stops bleed through and avoids a number of basic problems. Once you've created a new database in a database server that jenkins can access (usually localhost). Add a *shell script step* to the build that contains the following:

```
cat > config/app_local.php <<'CONFIG'
<?php
return [
    'Datasources' => [
        'test' => [
            'datasource' => 'Database/Mysql',
            'host'       => 'localhost',
```

[168] http://jenkins-ci.org
[169] http://jenkins-ci.org/

```
            'database'   => 'jenkins_test',
            'username'      => 'jenkins',
            'password'   => 'cakephp_jenkins',
            'encoding'   => 'utf8'
        ]
    ]
];
CONFIG
```

Then uncomment the following line in your **config/bootstrap.php** file:

```
//Configure::load('app_local', 'default');
```

By creating an **app_local.php** file, you have an easy way to define configuration specific to Jenkins. You can use this same configuration file to override any other configuration files you need on Jenkins.

It's often a good idea to drop and re-create the database before each build as well. This insulates you from chained failures, where one broken build causes others to fail. Add another *shell script step* to the build that contains the following:

```
mysql -u jenkins -pcakephp_jenkins -e 'DROP DATABASE IF EXISTS jenkins_test; CREATE␣
↪DATABASE jenkins_test';
```

Add your Tests

Add another *shell script step* to your build. In this step install your dependencies and run the tests for your application. Creating a junit log file, or clover coverage is often a nice bonus, as it gives you a nice graphical view of your testing results:

```
# Download Composer if it is missing.
test -f 'composer.phar' || curl -sS https://getcomposer.org/installer | php
# Install dependencies
php composer.phar install
vendor/bin/phpunit --log-junit junit.xml --coverage-clover clover.xml
```

If you use clover coverage, or the junit results, make sure to configure those in Jenkins as well. Failing to configure those steps will mean you won't see the results.

Run a Build

You should be able to run a build now. Check the console output and make any necessary changes to get a passing build.

CHAPTER 29

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[170].

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')
    ->notEmpty('title', 'Please fill this field')
    ->add('title', [
        'length' => [
            'rule' => ['minLength', 10],
            'message' => 'Titles need to be at least 10 characters long',
        ]
    ])
    ->allowEmpty('published')
    ->add('published', 'boolean', [
        'rule' => 'boolean'
    ])
    ->requirePresence('body')
    ->add('body', 'length', [
        'rule' => ['minLength', 50],
        'message' => 'Articles must have a substantial body.'
    ]);
```

[170] https://api.cakephp.org/3.x/class-Cake.Validation.Validation.html

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.',
    ]
]);
```

New in version 3.3.0: `requirePresence()` accepts an array of fields as of 3.3.0

Allowing Empty Fields

The `allowEmpty()` and `notEmpty()` methods allow you to control which fields are allowed to be 'empty'. By using the `notEmpty()` method, the given field will be marked invalid when it is empty. You can use `allowEmpty()` to allow a field to be empty. Both `allowEmpty()` and `notEmpty()` support a mode 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.

The values `''`, `null` and `[]` (empty array) will cause validation errors when fields are not allowed to be empty. When fields are allowed to be empty, the values `''`, `null`, `false`, `[]`, `0`, `'0'` are accepted.

An example of these methods in action is:

```
$validator->allowEmpty('published')
    ->notEmpty('title', 'Title cannot be empty')
    ->notEmpty('body', 'Body cannot be empty', 'create')
    ->allowEmpty('header_image', 'update');
```

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. However, 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.

Validation Methods Less Verbose

Since 3.2, the Validator object has a number of new methods that make building validators less verbose. For example adding validation rules to a username field can now look like:

```
$validator = new Validator();
$validator
    ->email('username')
    ->ascii('username')
    ->lengthBetween('username', [4, 8]);
```

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\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.

New in version 3.5.0.

You can use the Localized plugin[171] 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)
    {
        // add the provider to the validator
        $validator->setProvider('fr', '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.

[171] https://github.com/cakephp/localized

There are a few methods that are common to all classes, defined through the ValidationInterface interface[172]:

```
phone() to check a phone number
postal() to check a postal code
personId() to check a country specific person ID
```

Custom Validation Rules

In addition to using methods 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.

If you need to pass additional data to your validation methods such as the current user's id, you can use a custom dynamic provider from your controller.

```
$this->Examples->validator('default')->provider('passed', [
    'count' => $countFromController,
    'userid' => $this->Auth->user('id')
]);
```

[172] https://github.com/cakephp/localized/blob/master/src/Validation/ValidationInterface.php

Then ensure that your validation method has the second context parameter.

```
public function customValidationMethod($check, array $context)
{
    $userid = $context['providers']['passed']['userid'];
}
```

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:

```
$validator->allowEmpty('tax', function ($context) {
    return !$context['data']['is_taxable'];
});
```

Likewise, a field can be required to be populated when certain conditions are met:

```
$validator->notEmpty('email_frequency', 'This field is required', function ($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, since it would also be needed when canceling a subscription.

New in version 3.1.1: The callable support for `requirePresence()` was added in 3.1.1

Nesting Validators

New in version 3.0.5.

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.
$validator->errors($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.

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()
    {
        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')
    ->notEmpty('name', 'We need your name.')
    ->requirePresence('comment')
    ->notEmpty('comment', 'You need to give a comment.');

$errors = $validator->errors($this->request->getData());
if (empty($errors)) {
    // Send an email.
}
```

The `errors()` 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 `errors()` method applies rules for the 'create' mode. If you'd like to apply 'update' rules you can do the following:

```
$errors = $validator->errors($this->request->getData(), false);
if (empty($errors)) {
    // Send an email.
}
```

Note: If you need to validate entities you should use methods like `ORM\Table::newEntity()`, `ORM\Table::newEntities()`, `ORM\Table::patchEntity()`, `ORM\Table::patchEntities()` or *`ORM\Table::save()`* as they are designed for that.

Validating Entities

While entities are validated as they are saved, you may also want to validate entities before attempting to do any saving. Validating entities before saving is done automatically when using the `newEntity()`, `newEntities()`, `patchEntity()` or `patchEntities()`:

```
// In the ArticlesController class
$article = $this->Articles->newEntity($this->request->getData());
if ($article->errors()) {
```

```
    // Do work to show error messages.
}
```

Similarly, when you need to pre-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->errors()) {
            $this->Articles->save($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',
    ]
  ]
]);
```

Validation is commonly used for user-facing forms or interfaces, and thus it is not limited to only validating columns in the table schema. However, 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[173] 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:

```
$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.

[173] https://api.cakephp.org/3.x/class-Cake.Validation.Validation.html

CHAPTER 30

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 classnames throughout CakePHP. It resolves the short form names CakePHP uses and returns the fully resolved classname:

```
// Resolve a short classname with the namespace + suffix.
App::classname('Auth', 'Controller/Component', 'Component');
// Returns Cake\Controller\Component\AuthComponent

// 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 classnames do not exist, false will be returned.

Finding Paths to Namespaces

static Cake\Core\App::**path**(*string $package*, *string $plugin* = *null*)

Used to get locations for paths based on conventions:

```
// Get the path to Controller/ in your application
App::path('Controller');
```

This can be done for all namespaces that are part of your application. You can also fetch paths for a plugin:

```
// Returns the component paths in DebugKit
App::path('Component', 'DebugKit');
```

`App::path()` 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 Plugins

static `Cake\Core\Plugin::path`(*string $plugin*)

Plugins can be located with Plugin. Using `Plugin::path('DebugKit');` for example, will give you the full path to the DebugKit plugin:

```
$path = Plugin::path('DebugKit');
```

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\\": "App",
        "App\\Test\\": "Test",
        "": "./Plugin"
    },
    "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\\": "App",
        "App\\Test\\": "Test",
        "": "./Plugin"
    },
    "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 31

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.
$collectionA = new Collection($items);
$collectionB = collection($items);
```

The benefit of the helper method is that it is easier to chain off of than `(new Collection($items))`.

The `Collection\CollectionTrait` allows you to integrate collection-like features into any `Traversable` object you have in your application as well.

List of Methods

append	*avg*	*buffered*	*chunk*
chunkWithKeys	*combine*	*compile*	*contains*
countBy	*each*	*every*	*extract*
filter	*first*	*groupBy*	*indexBy*
insert	*isEmpty*	*last*	*listNested*
map	*match*	*max*	*median*
min	*nest*	*reduce*	*reject*
sample	*shuffle*	*skip*	*some*
sortBy	*stopWhen*	*sumOf*	*take*
through	*transpose*	*unfold*	*zip*

Iterating

`Cake\Collection\Collection::**each**(`*callable $c*`)`

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**(`*callable $c*`)`

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);

$new = $collection->map(function ($value, $key) {
    return $value * 2;
});

// $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**(`*$matcher*`)`

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->toArray();
```

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->toArray();
```

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',
        'phone_numbers' => [
            ['number' => 'number-4'],
            ['number' => 'number-5'],
        ]
    ]
];

$numbers = (new Collection($data))->extract('phone_numbers.{*}.number');
$numbers->toList();
// Returns ['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 will look like this when converted to array
[
    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 will look like this when converted to array
[
    '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 `Cake/Time` instances by the ORM) you may want to group results by date:

```
$combined = (new Collection($entities))->combine(
    'id',
    function ($entity) { return $entity; },
    function ($entity) { return $entity->date->toDateString(); }
);

// Result will look like this when converted to array
[
    '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 false 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->toArray();
```

`Cake\Collection\Collection::`**`unfold`**`(`*callable $c*`)`

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)->toArray();
});

$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)->toList();
// Creates
[
    ['a' => 1, 'b' => 2],
    ['c' => 3, 'd' => [4, 5]]
]
```

New in version 3.4.0: `chunkWithKeys()` was added in 3.4.0

Filtering

`Cake\Collection\Collection::filter`(*callable $c*)

Collections make it easy 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`(*callable $c*)

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`(*callable $c*)

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**(*array $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**(*array $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**(*callable $c*)

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->min('dad.age');
```

Cake\Collection\Collection::**sumOf**(*string|callable $callback*)

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**(*$matcher = 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: 150
$average = (new Collection($items))->avg('invoice.total');
```

New in version 3.5.0.

Cake\Collection\Collection::**median**(*$matcher = 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: 333
$median = (new Collection($items))->median('invoice.total');
```

New in version 3.5.0.

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 will look like this when converted to array:
[
  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(`*$elements*`)`

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)->toList();
// Returns:
[
    [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));
$firstYear->zip($data[0], $data[1])->toList();

// Or $firstYear->zip(...$data) in PHP >= 5.6

// Returns
[
```

```
    [100, 300, 400],
    [200, 500, 600]
]
```

Sorting

`Cake\Collection\Collection::sortBy(*$callback*)`

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*)`

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 is easy 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'],
]);

$collection->nest('id', 'parent_id')->toArray();
// Returns
[
    [
        '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(*$dir = 'desc'*, *$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:

```
$nested->listNested()->toList();

// Returns
[
    ['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:

```
$nested->listNested()->toArray();

// Returns
[
    ['id' => 3, 'parent_id' => 1, 'name' => 'Eagle'],
    ['id' => 4, 'parent_id' => 1, 'name' => 'Seagull'],
    ['id' => 5, 'parent_id' => 6, 'name' => 'Clown Fish']
]
```

Once you have converted a tree into a nested list, you can use the `printer()` method to configure how the list output should be formatted:

```
$nested->listNested()->printer('name', 'id', '--')->toArray();

// Returns
[
    3 => 'Eagle',
    4 => 'Seagull',
    5 -> '--Clown Fish',
]
```

The `printer()` method also lets you use a callback to generate the keys and or values:

```
$nested->listNested()->printer(
    function ($el) {
        return $el->name;
    },
    function ($el) {
        return $el->id;
    }
);
```

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()->toArray();
```

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'],
]
$transpose = (new Collection($items))->transpose()->toList();

// Returns
[
    ['Products', 'Product A', 'Product B', 'Product C'],
    ['2012', '200', '300', '400'],
    ['2013', '100', '200', '300'],
    ['2014', '50', '100', '200'],
]
```

New in version 3.3.0: `Collection::transpose()` was added in 3.3.0.

Withdrawing Elements

Cake\Collection\Collection::**sample**(*int $size*)

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**(*int $size*, *int $from*)

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**(*int $positions*)

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');
});
```

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` (*string $path*, *array|Traversable $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);
```

When converted to an array, the `$merged` collection will look like this:

```
[
    ['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 be filled with `null` values:

```
$languages = [
    ['PHP', 'Python', 'Ruby'],
    ['Bash', 'PHP', 'Javascript']
];

$merged = (new Collection($users))->insert('skills', $languages);

// Will yield
[
    ['username' => 'mark', 'skills' => ['PHP', 'Python', 'Ruby']],
    ['username' => 'juan', 'skills' => ['Bash', 'PHP', 'Javascript']],
    ['username' => 'jose', 'skills' => null]
];
```

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`(*callable $c*)

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');

$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`**(*bool $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 32

Folder & File

The Folder and File utilities are convenience classes to help you read from and write/append to files; list files within a folder and other common directory related tasks.

Basic Usage

Ensure the classes are loaded:

```
use Cake\Filesystem\Folder;
use Cake\Filesystem\File;
```

Then we can setup a new folder instance:

```
$dir = new Folder('/path/to/folder');
```

and search for all *.ctp* files within that folder using regex:

```
$files = $dir->find('.*\.ctp');
```

Now we can loop through the files and read from or write/append to the contents or simply delete the file:

```
foreach ($files as $file) {
    $file = new File($dir->pwd() . DS . $file);
    $contents = $file->read();
    // $file->write('I am overwriting the contents of this file');
    // $file->append('I am adding to the bottom of this file.');
    // $file->delete(); // I am deleting this file
    $file->close(); // Be sure to close the file when you're done
}
```

Folder API

class Cake\Filesystem**Folder**(*string $path = false*, *boolean $create = false*, *string|boolean $mode = false*)

```
// Create a new folder with 0755 permissions
$dir = new Folder('/path/to/folder', true, 0755);
```

property Cake\Filesystem\Folder::$**path**
Path of the current folder. *Folder::pwd()* will return the same information.

property Cake\Filesystem\Folder::$**sort**
Whether or not the list results should be sorted by name.

property Cake\Filesystem\Folder::$**mode**
Mode to be used when creating folders. Defaults to 0755. Does nothing on Windows machines.

static Cake\Filesystem\Folder::**addPathElement**(*string $path*, *string $element*)
Returns $path with $element added, with correct slash in-between:

```
$path = Folder::addPathElement('/a/path/for', 'testing');
// $path equals /a/path/for/testing
```

$element can also be an array:

```
$path = Folder::addPathElement('/a/path/for', ['testing', 'another']);
// $path equals /a/path/for/testing/another
```

Cake\Filesystem\Folder::**cd**(*$path*)
Change directory to $path. Returns false on failure:

```
$folder = new Folder('/foo');
echo $folder->path; // Prints /foo
$folder->cd('/bar');
echo $folder->path; // Prints /bar
$false = $folder->cd('/non-existent-folder');
```

Cake\Filesystem\Folder::**chmod**(*string $path*, *integer $mode = false*, *boolean $recursive = true*, *array $exceptions = []*)
Change the mode on a directory structure recursively. This includes changing the mode on files as well:

```
$dir = new Folder();
$dir->chmod('/path/to/folder', 0755, true, ['skip_me.php']);
```

Cake\Filesystem\Folder::**copy**(*array|string $options = []*)
Recursively copy a directory. The only parameter $options can either be a path into copy to or an array of options:

```
$folder1 = new Folder('/path/to/folder1');
$folder1->copy('/path/to/folder2');
// Will put folder1 and all its contents into folder2

$folder = new Folder('/path/to/folder');
$folder->copy([
    'to' => '/path/to/new/folder',
    'from' => '/path/to/copy/from', // Will cause a cd() to occur
    'mode' => 0755,
    'skip' => ['skip-me.php', '.git'],
    'scheme' => Folder::SKIP  // Skip directories/files that already exist.
]);
```

There are 3 supported schemes:

- Folder::SKIP skip copying/moving files & directories that exist in the destination directory.

- `Folder::MERGE` merge the source/destination directories. Files in the source directory will replace files in the target directory. Directory contents will be merged.
- `Folder::OVERWRITE` overwrite existing files & directories in the target directory with those in the source directory. If both the target and destination contain the same subdirectory, the target directory's contents will be removed and replaced with the source's.

static `Cake\Filesystem\Folder::correctSlashFor`(*string $path*)
Returns a correct set of slashes for given $path ('\' for Windows paths and '/' for other paths).

`Cake\Filesystem\Folder::create`(*string $pathname*, *integer $mode = false*)
Create a directory structure recursively. Can be used to create deep path structures like */foo/bar/baz/shoe/horn*:

```
$folder = new Folder();
if ($folder->create('foo' . DS . 'bar' . DS . 'baz' . DS . 'shoe' . DS . 'horn'))
↪{
    // Successfully created the nested folders
}
```

`Cake\Filesystem\Folder::delete`(*string $path = null*)
Recursively remove directories if the system allows:

```
$folder = new Folder('foo');
if ($folder->delete()) {
    // Successfully deleted foo and its nested folders
}
```

`Cake\Filesystem\Folder::dirsize`()
Returns the size in bytes of this Folder and its contents.

`Cake\Filesystem\Folder::errors`()
Get the error from latest method.

`Cake\Filesystem\Folder::find`(*string $regexpPattern = '.*'*, *boolean $sort = false*)
Returns an array of all matching files in the current directory:

```
// Find all .png in your webroot/img/ folder and sort the results
$dir = new Folder(WWW_ROOT . 'img');
$files = $dir->find('.*\.png', true);
/*
Array
(
    [0] => cake.icon.png
    [1] => test-error-icon.png
    [2] => test-fail-icon.png
    [3] => test-pass-icon.png
    [4] => test-skip-icon.png
)
*/
```

Note: The folder find and findRecursive methods will only find files. If you would like to get folders and files see *Folder::read()* or *Folder::tree()*

`Cake\Filesystem\Folder::findRecursive`(*string $pattern = '.*'*, *boolean $sort = false*)
Returns an array of all matching files in and below the current directory:

```
// Recursively find files beginning with test or index
$dir = new Folder(WWW_ROOT);
```

```
$files = $dir->findRecursive('(test|index).*');
/*
Array
(
    [0] => /var/www/cake/webroot/index.php
    [1] => /var/www/cake/webroot/test.php
    [2] => /var/www/cake/webroot/img/test-skip-icon.png
    [3] => /var/www/cake/webroot/img/test-fail-icon.png
    [4] => /var/www/cake/webroot/img/test-error-icon.png
    [5] => /var/www/cake/webroot/img/test-pass-icon.png
)
*/
```

Cake\Filesystem\Folder::**inCakePath**(*string $path = ''*)
Returns `true` if the file is in a given CakePath.

Cake\Filesystem\Folder::**inPath**(*string $path = ''*, *boolean $reverse = false*)
Returns `true` if the file is in the given path:

```
$Folder = new Folder(WWW_ROOT);
$result = $Folder->inPath(APP);
// $result = true, /var/www/example/ is in /var/www/example/webroot/

$result = $Folder->inPath(WWW_ROOT . 'img' . DS, true);
// $result = true, /var/www/example/webroot/ is in /var/www/example/webroot/img/
```

static Cake\Filesystem\Folder::**isAbsolute**(*string $path*)
Returns `true` if the given $path is an absolute path.

static Cake\Filesystem\Folder::**isSlashTerm**(*string $path*)
Returns `true` if given $path ends in a slash (i.e. is slash-terminated):

```
$result = Folder::isSlashTerm('/my/test/path');
// $result = false
$result = Folder::isSlashTerm('/my/test/path/');
// $result = true
```

static Cake\Filesystem\Folder::**isWindowsPath**(*string $path*)
Returns `true` if the given $path is a Windows path.

Cake\Filesystem\Folder::**messages**()
Get the messages from the latest method.

Cake\Filesystem\Folder::**move**(*array $options*)
Recursive directory move.

static Cake\Filesystem\Folder::**normalizePath**(*string $path*)
Returns a correct set of slashes for given $path ('\' for Windows paths and '/' for other paths).

Cake\Filesystem\Folder::**pwd**()
Return current path.

Cake\Filesystem\Folder::**read**(*boolean $sort = true*, *array|boolean $exceptions = false*, *boolean $fullPath = false*)
Returns an array of the contents of the current directory. The returned array holds two sub arrays: One of directories and one of files:

```
$dir = new Folder(WWW_ROOT);
$files = $dir->read(true, ['files', 'index.php']);
```

```
/*
Array
(
    [0] => Array // Folders
        (
            [0] => css
            [1] => img
            [2] => js
        )
    [1] => Array // Files
        (
            [0] => .htaccess
            [1] => favicon.ico
            [2] => test.php
        )
)
*/
```

Cake\Filesystem\Folder::**realpath**(*string $path*)
Get the real path (taking ".." and such into account).

static Cake\Filesystem\Folder::**slashTerm**(*string $path*)
Returns $path with added terminating slash (corrected for Windows or other OS).

Cake\Filesystem\Folder::**tree**(*null|string $path = null*, *array|boolean $exceptions = true*, *null|string $type = null*)
Returns an array of nested directories and files in each directory.

File API

class Cake\Filesystem**File**(*string $path*, *boolean $create = false*, *integer $mode = 755*)

```
// Create a new file with 0644 permissions
$file = new File('/path/to/file.php', true, 0644);
```

property Cake\Filesystem\File::$**Folder**
The Folder object of the file.

property Cake\Filesystem\File::$**name**
The name of the file with the extension. Differs from *File::name()* which returns the name without the extension.

property Cake\Filesystem\File::$**info**
An array of file info. Use *File::info()* instead.

property Cake\Filesystem\File::$**handle**
Holds the file handler resource if the file is opened.

property Cake\Filesystem\File::$**lock**
Enable locking for file reading and writing.

property Cake\Filesystem\File::$**path**
The current file's absolute path.

Cake\Filesystem\File::**append**(*string $data*, *boolean $force = false*)
Append the given data string to the current file.

Cake\Filesystem\File::**close**()
Closes the current file if it is opened.

Cake\Filesystem\File::**copy**(*string $dest*, *boolean $overwrite = true*)
Copy the file to $dest.

Cake\Filesystem\File::**create**()
Creates the file.

Cake\Filesystem\File::**delete**()
Deletes the file.

Cake\Filesystem\File::**executable**()
Returns `true` if the file is executable.

Cake\Filesystem\File::**exists**()
Returns `true` if the file exists.

Cake\Filesystem\File::**ext**()
Returns the file extension.

Cake\Filesystem\File::**Folder**()
Returns the current folder.

Cake\Filesystem\File::**group**()
Returns the file's group, or `false` in case of an error.

Cake\Filesystem\File::**info**()
Returns the file info.

Cake\Filesystem\File::**lastAccess**()
Returns last access time.

Cake\Filesystem\File::**lastChange**()
Returns last modified time, or `false` in case of an error.

Cake\Filesystem\File::**md5**(*integer|boolean $maxsize = 5*)
Get the MD5 Checksum of file with previous check of filesize, or `false` in case of an error.

Cake\Filesystem\File::**name**()
Returns the file name without extension.

Cake\Filesystem\File::**offset**(*integer|boolean $offset = false*, *integer $seek = 0*)
Sets or gets the offset for the currently opened file.

Cake\Filesystem\File::**open**(*string $mode = 'r'*, *boolean $force = false*)
Opens the current file with the given $mode.

Cake\Filesystem\File::**owner**()
Returns the file's owner.

Cake\Filesystem\File::**perms**()
Returns the "chmod" (permissions) of the file.

static Cake\Filesystem\File::**prepare**(*string $data*, *boolean $forceWindows = false*)
Prepares a ascii string for writing. Converts line endings to the correct terminator for the current platform. For Windows "\r\n" will be used, "\n" for all other platforms.

Cake\Filesystem\File::**pwd**()
Returns the full path of the file.

Cake\Filesystem\File::**read**(*string $bytes = false*, *string $mode = 'rb'*, *boolean $force = false*)
Return the contents of the current file as a string or return `false` on failure.

Cake\Filesystem\File::**readable**()
Returns `true` if the file is readable.

Cake\Filesystem\File::**safe**(*string $name = null*, *string $ext = null*)
Makes filename safe for saving.

Cake\Filesystem\File::**size**()
Returns the filesize in bytes.

Cake\Filesystem\File::**writable**()
Returns `true` if the file is writable.

Cake\Filesystem\File::**write**(*string $data*, *string $mode = 'w'*, *boolean$force = false*)
Write given data to the current file.

Cake\Filesystem\File::**mime**()
Get the file's mimetype, returns `false` on failure.

Cake\Filesystem\File::**replaceText**(*$search*, *$replace*)
Replaces text in a file. Returns `false` on failure and `true` on success.

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']],
        ['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');
/* $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
        ]
    ]
*/
```

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'
        ]
    ],
    [
        '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'],
    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. You callback should return `false` to remove elements from the resulting array:

```
$data = [
    '0',
    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,
    '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] => [
            [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:
    [
        [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']];
$res = Hash::mergeDiff($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*)

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 null 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
                ]
        ]
    ]
*/

$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
        ]
    */
```

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]],
    ['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 basic but powerful HTTP client which can be used for making requests. It is a great way to communicate with webservices, and remote APIs.

Changed in version 3.3.0: Prior to 3.3.0 you should use `Cake\Network\Http\Client`.

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();
$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(...);
```

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.

Warning: For compatibility reasons, strings beginning with `@` will be evaluated as local or remote file paths.

This functionality is deprecated as of CakePHP 3.0.5 and will be removed in a future version. Until that happens, user data being passed to the Http Client must be sanitized as follows:

```
$response = $http->post('http://example.com/api', [
    'search' => ltrim($this->request->getData('search'), '@'),
]);
```

If it is necessary to preserve leading `@` characters in query strings, you can pass a pre-encoded query string from `http_build_query()`:

```
$response = $http->post('http://example.com/api', http_build_query([
    'search' => $this->request->getData('search'),
]));
```

Building Multipart Request Bodies by Hand

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);
// 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 API's 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`.

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.

OAuth 1 Authentication

Many modern web-services require OAuth authentication to access their API's. 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()[174].

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:

[174] http://php.net/manual/en/function.stream-context-create.php

```
// 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');
```

The following information can be used when creating a scoped client:

- host
- 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'));
```

New in version 3.5.0: `addCookie()` was added in 3.5.0

Response Objects

class `Cake\Http\Client\`**Response**

Response objects have a number of methods for inspecting the response data.

Changed in version 3.3.0: As of 3.3.0 `Cake\Http\Client\Response` implements the PSR-7 ResponseInterface[175].

Reading Response Bodies

You read the entire response body as a string:

```
// Read the entire response as a string.
$response->body();

// As a property
$response->body;
```

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 easy to use accessors to read decoded data. JSON data is decoded into an array, while XML data is decoded into a `SimpleXMLElement` tree:

[175] http://www.php-fig.org/psr/psr-7/#3-3-psr-http-message-responseinterface

```
// Get some XML
$http = new Client();
$response = $http->get('http://example.com/test.xml');
$xml = $response->xml;

// Get some JSON
$http = new Client();
$response = $http->get('http://example.com/test.json');
$json = $response->json;
```

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();

// Get an array of key=>value for all headers
$response->headers;
```

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');

// Access the complete data for all cookies.
$response->cookies;
```

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();

// __get() helper
$response->code;
```

CHAPTER 35

Inflector

class `Cake\Utility\`**`Inflector`**

The Inflector class takes a string and can manipulate it to handle word variations such as pluralizations or camelizing and is normally accessed statically. Example: `Inflector::pluralize('example')` returns "examples".

You can try out the inflections online at inflector.cakephp.org[176].

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:

Method	Argument	Output
`pluralize()`	BigApple	BigApples
	big_apple	big_apples
`singularize()`	BigApples	BigApple
	big_apples	big_apple
`camelize()`	big_apples	BigApples
	big apple	BigApple
`underscore()`	BigApples	big_apples
	Big Apples	big apples
`humanize()`	big_apples	Big Apples
	bigApple	BigApple
`classify()`	big_apples	BigApple
	big apple	BigApple
`dasherize()`	BigApples	big-apples
	big apple	big apple
`tableize()`	BigApple	big_apples
	Big Apple	big apples
`variable()`	big_apple	bigApple
	big apples	bigApples
`slug()`	Big Apple	big-apple
	BigApples	BigApples

[176] https://inflector.cakephp.org/

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()` may not always correctly convert a noun that is already in its plural form.

```
// Person
echo Inflector::singularize('People');
```

Note: `singularize()` may not always correctly convert 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');
```

Creating URL Safe Strings

static Cake\Utility\Inflector::**slug**(*$word*, *$replacement* = '-')

Slug converts special characters into latin versions and converting unmatched characters and spaces to dashes. The slug method expects UTF-8 encoding:

```
// apple-puree
Inflector::slug('apple purée');
```

Note: `Inflector::slug()` has been deprecated since 3.2.7. Use `Text::slug()` instead.

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()
    {
        parent::initialize();
        $this->loadComponent('Auth');
    }

    public function afterLogin()
    {
        $storageUsed = $this->Auth->user('storage_used');
        if ($storageUsed > 5000000) {
            // Notify users of quota
            $this->Flash->success(__('You are using {0} storage', Number::
↪toReadableSize($storageUsed)));
        }
    }
}
```

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). 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()*

Setting the Default Currency

`Cake\I18n\Number::`**`defaultCurrency`**(*$currency*)

Setter/getter 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 `false`, it will clear the currently stored value. By default, it will retrieve the `intl.default_locale` 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->Auth = $this->loadComponent('Auth', ['className' => 'MyCustomAuth']);
$this->Auth->user(); // Actually using MyCustomAuth::user();
```

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 Auth from callbacks.
$this->eventManager()->off($this->Auth);

// Re-enable Auth for callbacks.
$this->eventManager()->on($this->Auth);
```

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()
    {
        parent::initialize();
        $this->loadComponent('Auth')
    };

    public function afterLogin()
    {
        $message = $this->Users->find('new_message');
        if (!empty($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[177]:

```
// 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 e.g. 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' => '.']);
```

Generating UUIDs

static `Cake\Utility\Text::uuid`

The UUID method is used to generate unique identifiers as per **RFC 4122**[178]. The UUID is a 128-bit string in the format of `485fc381-e790-47a3-9794-1337c0a8fe68`.

```
Text::uuid(); // 485fc381-e790-47a3-9794-1337c0a8fe68
```

[177] http://userguide.icu-project.org/transforms/general#TOC-Transliterator-Identifiers
[178] https://tools.ietf.org/html/rfc4122.html

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:

```
$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 identation, you need to use `wrapBlock()` instead of `wrap()`. This is particulary 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:

```
Highlights $needle in $haystack <span class="highlight">using</span> the
$options['format'] string specified  or a default string.
```

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,
    [
        '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\`**`Time`**

If you need `TimeHelper` functionalities outside of a `View`, use the `Time` class:

```
use Cake\I18n\Time;

class UsersController extends AppController
{

    public function initialize()
    {
        parent::initialize();
        $this->loadComponent('Auth');
    }

    public function afterLogin()
    {
        $time = new Time($this->Auth->user('date_of_birth'));
        if ($time->isToday()) {
            // Greet user with a happy birthday message
            $this->Flash->success(__('Happy birthday to you...'));
        }
    }
}
```

Under the hood, CakePHP uses Chronos[179] to power its `Time` utility. Anything you can do with `Chronos` and `DateTime`, you can do with `Time` and `Date`.

Note: Prior to 3.2.0 CakePHP used Carbon[180].

For more details on Chronos please see the API documentation[181].

[179] https://github.com/cakephp/chronos
[180] https://github.com/briannesbitt/Carbon
[181] https://api.cakephp.org/chronos/1.0/

Creating Time Instances

There are a few ways to create `Time` instances:

```
use Cake\I18n\Time;

// Create from a string datetime.
$time = Time::createFromFormat(
    'Y-m-d H:i:s',
    $datetime,
    'America/New_York'
);

// Create from a timestamp
$time = Time::createFromTimestamp($ts);

// Get the current time.
$time = Time::now();

// Or just use 'new'
$time = new Time('2014-01-10 11:11', 'America/New_York');

$time = new Time('2 hours ago');
```

The `Time` class constructor can take any parameter that the internal `DateTime` 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.
$now = new Time('2014-04-12 12:22:30');
Time::setTestNow($now);

// Returns '2014-04-12 12:22:30'
$now = Time::now();

// Returns '2014-04-12 12:22:30'
$now = Time::parse('now');
```

Manipulation

Once created, you can manipulate `Time` instances using setter methods:

```
$now = Time::now();
$now->year(2013)
    ->month(10)
    ->day(31);
```

You can also use the methods provided by PHP's built-in `DateTime` class:

```
$now->setDate(2013, 10, 31);
```

Dates can be modified through subtraction and addition of their components:

```
$now = Time::now();
$now->subDays(5);
$now->addMonth(1);

// Using strtotime strings.
$now->modify('+5 days');
```

You can get the internal components of a date by accessing its properties:

```
$now = Time::now();
echo $now->year; // 2014
echo $now->month; // 5
echo $now->day; // 10
echo $now->timezone; // America/New_York
```

It is also allowed to directly assign those properties to modify the date:

```
$time->year = 2015;
$time->timezone = 'Europe/Paris';
```

Formatting

static `Cake\I18n\Time::setJsonEncodeFormat` (*$format*)

This method sets the default format used when converting an object to json:

```
Time::setJsonEncodeFormat('yyyy-MM-dd HH:mm:ss');  // For any mutable DateTime
FrozenTime::setJsonEncodeFormat('yyyy-MM-dd HH:mm:ss');  // For any immutable DateTime
Date::setJsonEncodeFormat('yyyy-MM-dd HH:mm:ss');  // For any mutable Date
FrozenDate::setJsonEncodeFormat('yyyy-MM-dd HH:mm:ss');  // For any immutable Date
```

Note: This method must be called statically.

`Cake\I18n\Time::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:

```
$now = Time::parse('2014-10-31');

// Prints a localized datetime stamp.
echo $now;

// Outputs '10/31/14, 12:00 AM' for the en-US locale
$now->i18nFormat();

// Use the full date and time format
$now->i18nFormat(\IntlDateFormatter::FULL);

// Use full date but short time format
$now->i18nFormat([\IntlDateFormatter::FULL, \IntlDateFormatter::SHORT]);

// Outputs '2014-10-31 00:00:00'
$now->i18nFormat('yyyy-MM-dd HH:mm:ss');
```

It is possible to specify the desired format for the string to be displayed. You can either pass IntlDateFormatter constants[182] as the first argument of this function, or pass a full ICU date formatting string as specified in the following resource: http://www.icu-project.org/apiref/icu4c/classSimpleDateFormat.html#details.

You can also format dates with non-gregorian calendars:

```
// Outputs 'Friday, Aban 9, 1393 AP at 12:00:00 AM GMT'
$result = $now->i18nFormat(\IntlDateFormatter::FULL, null, 'en-IR@calendar=persian');
```

The following calendar types are supported:

- japanese
- buddhist
- chinese
- persian
- indian
- islamic
- hebrew
- coptic
- ethiopic

New in version 3.1: Non-gregorian calendar support was added in 3.1

Note: For constant strings i.e. IntlDateFormatter::FULL Intl uses ICU library that feeds its data from CLDR (http://cldr.unicode.org/) which version may vary depending on PHP installation and give different results.

`Cake\I18n\Time::nice()`

Print out a predefined 'nice' format:

```
$now = Time::parse('2014-10-31');

// Outputs 'Oct 31, 2014 12:00 AM' in en-US
echo $now->nice();
```

You can alter the timezone in which the date is displayed without altering the `Time` object itself. This is useful when you store dates in one timezone, but want to display them in a user's own timezone:

```
$now->i18nFormat(\IntlDateFormatter::FULL, 'Europe/Paris');
```

Leaving the first parameter as `null` will use the default formatting string:

```
$now->i18nFormat(null, 'Europe/Paris');
```

Finally, it is possible to use a different locale for displaying a date:

```
echo $now->i18nFormat(\IntlDateFormatter::FULL, 'Europe/Paris', 'fr-FR');

echo $now->nice('Europe/Paris', 'fr-FR');
```

[182] http://www.php.net/manual/en/class.intldateformatter.php

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[183]. You can, however, modify this default at runtime:

```
Time::setDefaultLocale('es-ES'); // For any mutable DateTime
FrozenTime::setDefaultLocale('es-ES'); // For any immutable DateTime
Date::setDefaultLocale('es-ES'); // For any mutable Date
FrozenDate::setDefaultLocale('es-ES'); // For any immutable Date
```

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`:

```
Time::setToStringFormat(\IntlDateFormatter::SHORT); // For any mutable DateTime
FrozenTime::setToStringFormat(\IntlDateFormatter::SHORT); // For any immutable␣
↪DateTime
Date::setToStringFormat(\IntlDateFormatter::SHORT); // For any mutable Date
FrozenDate::setToStringFormat(\IntlDateFormatter::SHORT); // For any immutable Date

// The same method exists on Date, FrozenDate and FrozenTime
Time::setToStringFormat([
    \IntlDateFormatter::FULL,
    \IntlDateFormatter::SHORT
]);

// The same method exists on Date, FrozenDate and FrozenTime
Time::setToStringFormat('yyyy-MM-dd HH:mm:ss');
```

It is recommended to always use the constants instead of directly passing a date format string.

Formatting Relative Times

`Cake\I18n\Time::`**`timeAgoInWords`**(*array $options = []*)

Often it is useful to print times relative to the present:

```
$now = new Time('Aug 22, 2011');
echo $now->timeAgoInWords(
    ['format' => 'MMM d, YYY', 'end' => '+1 year']
);
// On Nov 10th, 2011 this would display: 2 months, 2 weeks, 6 days ago
```

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:

```
// If $timestamp is 1 month, 1 week, 5 days and 6 hours ago
echo $timestamp->timeAgoInWords([
    'accuracy' => ['month' => 'month'],
    'end' => '1 year'
]);
// Outputs '1 month ago'
```

By setting `accuracy` to a string, you can specify what is the maximum level of detail you want output:

[183] http://www.php.net/manual/en/intl.configuration.php#ini.intl.default-locale

```
$time = new Time('+23 hours');
// Outputs 'in about a day'
$result = $time->timeAgoInWords([
    'accuracy' => 'day'
]);
```

Conversion

`Cake\I18n\Time::toQuarter()`

Once created, you can convert `Time` instances into timestamps or quarter values:

```
$time = new Time('2014-06-15');
$time->toQuarter();
$time->toUnixString();
```

Comparing With the Present

`Cake\I18n\Time::isYesterday()`

`Cake\I18n\Time::isThisWeek()`

`Cake\I18n\Time::isThisMonth()`

`Cake\I18n\Time::isThisYear()`

You can compare a `Time` instance with the present in a variety of ways:

```
$time = new Time('2014-06-15');

echo $time->isYesterday();
echo $time->isThisWeek();
echo $time->isThisMonth();
echo $time->isThisYear();
```

Each of the above methods will return `true`/`false` based on whether or not the `Time` instance matches the present.

Comparing With Intervals

`Cake\I18n\Time::isWithinNext(`*$interval*`)`

You can see if a `Time` instance falls within a given range using `wasWithinLast()` and `isWithinNext()`:

```
$time = new Time('2014-06-15');

// Within 2 days.
echo $time->isWithinNext(2);

// Within 2 next weeks.
echo $time->isWithinNext('2 weeks');
```

`Cake\I18n\Time::wasWithinLast(`*$interval*`)`

You can also compare a `Time` instance within a range in the past:

```
// Within past 2 days.
echo $time->wasWithinLast(2);

// Within past 2 weeks.
echo $time->wasWithinLast('2 weeks');
```

Dates

New in version 3.2.

The `Date` class in CakePHP implements the same API and methods as *`Cake\I18n\Time`* does. The main difference between `Time` and `Date` is that `Date` does not track time components, and is always in UTC. As an example:

```
use Cake\I18n\Date;
$date = new Date('2015-06-15');

$date->modify('+2 hours');
// Outputs 2015-06-15 00:00:00
echo $date->format('Y-m-d H:i:s');

$date->modify('+36 hours');
// Outputs 2015-06-15 00:00:00
echo $date->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('2015-06-15');
$date->setTimezone(new \DateTimeZone('America/New_York'));

// Outputs UTC
echo $date->format('e');
```

Immutable Dates and Times

class `Cake\I18n\`**`FrozenTime`**

class `Cake\I18n\`**`FrozenDate`**

CakePHP offers immutable date and time classes that implement the same interface as their mutable 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\FrozenTime;
$time = new FrozenTime('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[184].

[184] http://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.

Importing Data to Xml Class

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:

```
// Local file
$xml = Xml::build('/home/awesome/unicorns.xml');
```

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[185] and SimpleXML[186] implement different API's. Be sure to use the correct methods on the object you request from Xml.

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.

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:

[185] http://php.net/domdocument
[186] http://php.net/simplexml

```
$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);

$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.

__ (*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('PluginName','This is my plugin');`

__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*)
Changed in version 3.3.0: Calling this method will return passed `$var`, so that you can, for instance, place it in return statements.

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 $var*)
Changed in version 3.3.0: Calling this method will return passed `$var`, so that you can, for instance, place it in return statements.

Convenience wrapper for `print_r()`, with the addition of wrapping `<pre>` tags around the output.

pj (*mixed $var*)
Changed in version 3.3.0: Calling this method will return passed `$var`, so that you can, for instance, place it in return statements.

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 $default = null*)
Changed in version 3.1.1: The `$default` parameter has been added.

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 APP
Absolute path to your application directory, including a trailing slash.

constant APP_DIR
Equals `app` or the name of your application directory.

constant CACHE
Path to the cache files directory. It can be shared between hosts in a multi-server setup.

constant CAKE
Path to the cake directory.

constant CAKE_CORE_INCLUDE_PATH
Path to the root lib directory.

constant CONFIG
Path to the config directory.

constant CORE_PATH
Path to the root directory with ending directory slash.

constant DS
Short for PHP's `DIRECTORY_SEPARATOR`, which is `/` on Linux and `\` on Windows.

constant LOGS
Path to the logs directory.

constant ROOT
Path to the root directory.

constant TESTS
Path to the tests directory.

constant TMP
Path to the temporary files directory.

constant WWW_ROOT
Full path to the webroot.

Timing Definition Constants

constant TIME_START
Unix timestamp in microseconds as a float from when the application started.

constant `SECOND`
Equals 1

constant `MINUTE`
Equals 60

constant `HOUR`
Equals 3600

constant `DAY`
Equals 86400

constant `WEEK`
Equals 604800

constant `MONTH`
Equals 2592000

constant `YEAR`
Equals 31536000

CHAPTER 42

Debug Kit

DebugKit is a plugin supported by the core team that provides a toolbar to help make debugging CakePHP applications easier.

Installation

By default DebugKit is installed with the default application skeleton. If you've removed it and want to re-install it, you can do so by running the following from your application's ROOT directory (where composer.json file is located):

```
php composer.phar require --dev cakephp/debug_kit "~3.0"
```

Then, you need to enable the plugin by executing the following line:

```
bin/cake plugin load DebugKit
```

DebugKit Storage

By default, DebugKit uses a small SQLite database in your application's `/tmp` directory to store the panel data. If you'd like DebugKit to store its data elsewhere, you should define a `debug_kit` connection.

Database Configuration

By default DebugKit will store panel data into a SQLite database in your application's `tmp` directory. If you cannot install pdo_sqlite, you can configure DebugKit to use a different database by defining a `debug_kit` connection in your **config/app.php** file. For example:

```
/**
 * The debug_kit connection stores DebugKit meta-data.
 */
'debug_kit' => [
    'className' => 'Cake\Database\Connection',
    'driver' => 'Cake\Database\Driver\Mysql',
    'persistent' => false,
    'host' => 'localhost',
```

```
    //'port' => 'nonstandard_port_number',
    'username' => 'dbusername',    // Your DB username here
    'password' => 'dbpassword',    // Your DB password here
    'database' => 'debug_kit',
    'encoding' => 'utf8',
    'timezone' => 'UTC',
    'cacheMetadata' => true,
    'quoteIdentifiers' => false,
    //'init' => ['SET GLOBAL innodb_stats_on_metadata = 0'],
],
```

Toolbar Usage

The DebugKit Toolbar is comprised of several panels, which are shown by clicking the CakePHP icon in the bottom right-hand corner of your browser window. Once the toolbar is open, you should see a series of buttons. Each of these buttons expands into a panel of related information.

Each panel lets you look at a different aspect of your application:

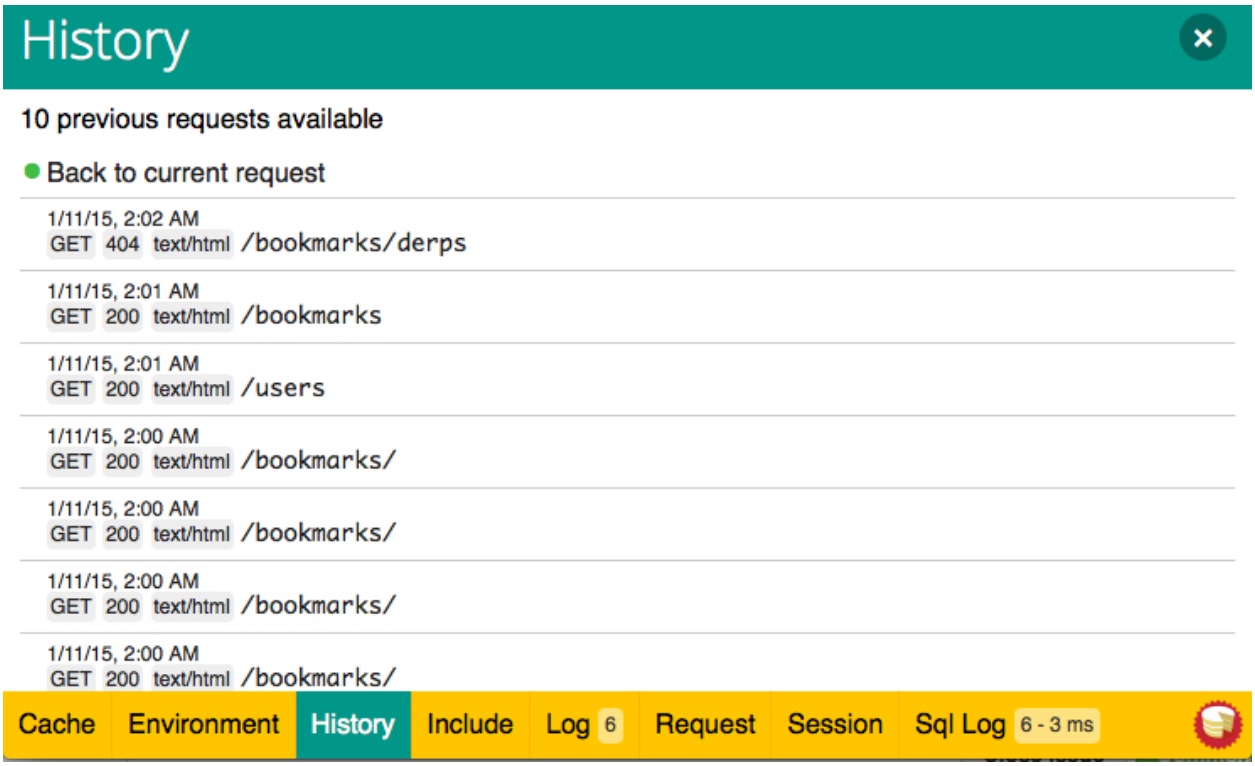
- **Cache** See cache usage during a request and clear caches.
- **Environment** Display environment variables related to PHP + CakePHP.
- **History** Displays a list of previous requests, and allows you to load and view toolbar data from previous requests.
- **Include** View the included files grouped by type.
- **Log** Display any entries made to the log files this request.
- **Packages** Display the list of packages dependencies with their actual version and allow you to check for outdated packages.
- **Mail** Display all emails sent during the request and allow to preview emails during development without sending them.
- **Request** Displays information about the current request, GET, POST, Cake Parameters, Current Route information and Cookies.
- **Session** Display the information currently in the Session.
- **Sql Logs** Displays SQL logs for each database connection.
- **Timer** Display any timers that were set during the request with `DebugKit\DebugTimer`, and memory usage collected with `DebugKit\DebugMemory`.
- **Variables** Display View variables set in controller.

Typically, a panel handles the collection and display of a single type of information such as Logs or Request information. You can choose to view panels from the toolbar or add your own custom panels.

Using the History Panel

The history panel is one of the most frequently misunderstood features of DebugKit. It provides a way to view toolbar data from previous requests, including errors and redirects.

As you can see, the panel contains a list of requests. On the left you can see a dot marking the active request. Clicking any request data will load the panel data for that request. When historical data is loaded the panel titles will transition to indicate that alternative data has been loaded.



Using The Mail Panel

The mail panel allow you to track all emails sent during a request.

The mailer preview allows you to easily check emails during development.

Developing Your Own Panels

You can create your own custom panels for DebugKit to help in debugging your applications.

Creating a Panel Class

Panel Classes simply need to be placed in the **src/Panel** directory. The filename should match the classname, so the class `MyCustomPanel` would be expected to have a filename of **src/Panel/MyCustomPanel.php**:

```
namespace App\Panel;

use DebugKit\DebugPanel;

/**
 * My Custom Panel
 */
class MyCustomPanel extends DebugPanel
{
    ...
}
```

Notice that custom panels are required to extend the `DebugPanel` class.

Callbacks

By default Panel objects have two callbacks, allowing them to hook into the current request. Panels subscribe to the `Controller.initialize` and `Controller.shutdown` events. If your panel needs to subscribe to additional events, you can use the `implementedEvents()` method to define all of the events your panel is interested in.

You should refer to the built-in panels for some examples on how you can build panels.

Panel Elements

Each Panel is expected to have a view element that renders the content from the panel. The element name must be the underscored inflection of the class name. For example `SessionPanel` has an element named **session_panel.ctp**, and SqllogPanel has an element named **sqllog_panel.ctp**. These elements should be located in the root of your **src/Template/Element** directory.

Custom Titles and Elements

Panels should pick up their title and element name by convention. However, if you need to choose a custom element name or title, you can define methods to customize your panel's behavior:

- `title()` - Configure the title that is displayed in the toolbar.
- `elementName()` - Configure which element should be used for a given panel.

Panel Hook Methods

You can also implement the following hook methods to customize how your panel behaves and appears:

- `shutdown(Event $event)` This method typically collects and prepares the data for the panel. Data is generally stored in `$this->_data`.
- `summary()` Can return a string of summary data to be displayed in the toolbar even when a panel is collapsed. Often this is a counter, or short summary information.
- `data()` Returns the panel's data to be used as element context. This hook method lets you further manipulate the data collected in the `shutdown()` method. This method **must** return data that can be serialized.

Panels in Other Plugins

Panels provided by *Plugins* work almost entirely the same as other plugins, with one minor difference: You must set `public $plugin` to be the name of the plugin directory, so that the panel's Elements can be located at render time:

```
namespace MyPlugin\Panel;

use DebugKit\DebugPanel;

class MyCustomPanel extends DebugPanel
{
    public $plugin = 'MyPlugin';
        ...
}
```

To use a plugin or app panel, update your application's DebugKit configuration to include the panel:

```
// in config/bootstrap.php
Configure::write('DebugKit.panels', ['App', 'MyPlugin.MyCustom']);
Plugin::load('DebugKit', ['bootstrap' => true]);
```

The above would load all the default panels as well as the `AppPanel`, and `MyCustomPanel` panel from `MyPlugin`.

CHAPTER 43

Migrations

Migrations is a plugin supported by the core team that helps you do schema changes in your database by writing PHP files that can be tracked using your version control system.

It allows you to evolve your database tables over time. Instead of writing schema modifications in SQL, this plugin allows you to use an intuitive set of methods to implement your database changes.

This plugin is a wrapper for the database migrations library Phinx[187].

Installation

By default Migrations is installed with the default application skeleton. If you've removed it and want to re-install it, you can do so by running the following from your application's ROOT directory (where composer.json file is located):

```
$ php composer.phar require cakephp/migrations "@stable"

// Or if composer is installed globally

$ composer require cakephp/migrations "@stable"
```

To use the plugin you'll need to load it in your application's **config/bootstrap.php** file. You can use *CakePHP's Plugin shell* to load and unload plugins from your **config/bootstrap.php**:

```
$ bin/cake plugin load Migrations
```

Or you can load the plugin by editing your **config/bootstrap.php** file and adding the following statement:

```
Plugin::load('Migrations');
```

Additionally, you will need to configure the default database configuration for your application in your **config/app.php** file as explained in the *Database Configuration section*.

[187] https://phinx.org/

Overview

A migration is basically a single PHP file that describes the changes to operate to the database. A migration file can create or drop tables, add or remove columns, create indexes and even insert data into your database.

Here's an example of a migration:

```
<?php
use Migrations\AbstractMigration;

class CreateProducts extends AbstractMigration
{
    /**
     * Change Method.
     *
     * More information on this method is available here:
     * http://docs.phinx.org/en/latest/migrations.html#the-change-method
     * @return void
     */
    public function change()
    {
        $table = $this->table('products');
        $table->addColumn('name', 'string', [
            'default' => null,
            'limit' => 255,
            'null' => false,
        ]);
        $table->addColumn('description', 'text', [
            'default' => null,
            'null' => false,
        ]);
        $table->addColumn('created', 'datetime', [
            'default' => null,
            'null' => false,
        ]);
        $table->addColumn('modified', 'datetime', [
            'default' => null,
            'null' => false,
        ]);
        $table->create();
    }
}
```

This migration will add a table to your database named `products` with the following column definitions:

- `id` column of type `integer` as primary key
- `name` column of type `string`
- `description` column of type `text`
- `created` column of type `datetime`
- `modified` column of type `datetime`

Tip: The primary key column named `id` will be added **implicitly**.

Note: Note that this file describes how the database will look **after** applying the migration. At this point no `products` table exists in your database, we have merely created a file that is able to both create the `products` table with the specified columns as well as drop it when a `rollback` operation of the migration is performed.

Once the file has been created in the **config/Migrations** folder, you will be able to execute the following `migrations` command to create the table in your database:

```
bin/cake migrations migrate
```

The following `migrations` command will perform a `rollback` and drop the table from your database:

```
bin/cake migrations rollback
```

Creating Migrations

Migration files are stored in the **config/Migrations** directory of your application. The name of the migration files are prefixed with the date in which they were created, in the format **YYYYMMDDHHMMSS_MigrationName.php**. Here are examples of migration filenames:

- 20160121163850_CreateProducts.php
- 20160210133047_AddRatingToProducts.php

The easiest way to create a migrations file is by using the *Code Generation with Bake* CLI command.

Please make sure you read the official Phinx documentation[188] in order to know the complete list of methods you can use for writing migration files.

Note: When using the `bake` option, you can still modify the migration before running them if so desired.

Syntax

The `bake` command syntax follows the form below:

```
$ bin/cake bake migration CreateProducts name:string description:text created modified
```

When using `bake` to create tables, add columns and so on, to your database, you will usually provide two things:

- the name of the migration you will generate (`CreateProducts` in our example)
- the columns of the table that will be added or removed in the migration (`name:string description:text created modified` in our example)

Due to the conventions, not all schema changes can be performed via these shell commands.

Additionally you can create an empty migrations file if you want full control over what needs to be executed, by omitting to specify a columns definition:

```
$ bin/cake migrations create MyCustomMigration
```

[188] http://docs.phinx.org/en/latest/migrations.html

Migrations file name

Migration names can follow any of the following patterns:

- (`/^(Create)(.*)/`) Creates the specified table.
- (`/^(Drop)(.*)/`) Drops the specified table. Ignores specified field arguments
- (`/^(Add).*(?:To)(.*)/`) Adds fields to the specified table
- (`/^(Remove).*(?:From)(.*)/`) Removes fields from the specified table
- (`/^(Alter)(.*)/`) Alters the specified table. An alias for CreateTable and AddField.

You can also use the `underscore_form` as the name for your migrations i.e. `create_products`.

New in version cakephp/migrations: 1.5.2

As of v1.5.2 of the migrations plugin[189], the migration filename will be automatically camelized. This version of the plugin is only available with a release of CakePHP >= to 3.1. Prior to this version of the plugin the migration name would be in the underscore form, i.e. `20160121164955_create_products.php`.

Warning: Migration names are used as migration class names, and thus may collide with other migrations if the class names are not unique. In this case, it may be necessary to manually override the name at a later date, or simply change the name you are specifying.

Columns definition

When using columns in the command line, it may be handy to remember that they follow the following pattern:

```
fieldName:fieldType?[length]:indexType:indexName
```

For instance, the following are all valid ways of specifying an email field:

- `email:string?`
- `email:string:unique`
- `email:string?[50]`
- `email:string:unique:EMAIL_INDEX`
- `email:string[120]:unique:EMAIL_INDEX`

The question mark following the fieldType will make the column nullable.

The `length` parameter for the `fieldType` is optional and should always be written between bracket.

Fields named `created` and `modified`, as well as any field with a `_at` suffix, will automatically be set to the type `datetime`.

Field types are those generically made available by the `Phinx` library. Those can be:

- string
- text
- integer
- biginteger

[189] https://github.com/cakephp/migrations/

- float
- decimal
- datetime
- timestamp
- time
- date
- binary
- boolean
- uuid

There are some heuristics to choosing fieldtypes when left unspecified or set to an invalid value. Default field type is `string`:

- id: integer
- created, modified, updated: datetime

Creating a table

You can use `bake` to create a table:

```
$ bin/cake bake migration CreateProducts name:string description:text created modified
```

The command line above will generate a migration file that resembles:

```
<?php
use Migrations\AbstractMigration;

class CreateProducts extends AbstractMigration
{
    /**
     * Change Method.
     *
     * More information on this method is available here:
     * http://docs.phinx.org/en/latest/migrations.html#the-change-method
     * @return void
     */
    public function change()
    {
        $table = $this->table('products');
        $table->addColumn('name', 'string', [
            'default' => null,
            'limit' => 255,
            'null' => false,
        ]);
        $table->addColumn('description', 'text', [
            'default' => null,
            'null' => false,
        ]);
        $table->addColumn('created', 'datetime', [
            'default' => null,
            'null' => false,
        ]);
```

```
        $table->addColumn('modified', 'datetime', [
            'default' => null,
            'null' => false,
        ]);
        $table->create();
    }
}
```

Adding columns to an existing table

If the migration name in the command line is of the form "AddXXXToYYY" and is followed by a list of column names and types then a migration file containing the code for creating the columns will be generated:

```
$ bin/cake bake migration AddPriceToProducts price:decimal
```

Executing the command line above will generate:

```
<?php
use Migrations\AbstractMigration;

class AddPriceToProducts extends AbstractMigration
{
    public function change()
    {
        $table = $this->table('products');
        $table->addColumn('price', 'decimal')
              ->update();
    }
}
```

Adding a column as index to a table

It is also possible to add indexes to columns:

```
$ bin/cake bake migration AddNameIndexToProducts name:string:index
```

will generate:

```
<?php
use Migrations\AbstractMigration;

class AddNameIndexToProducts extends AbstractMigration
{
    public function change()
    {
        $table = $this->table('products');
        $table->addColumn('name', 'string')
              ->addIndex(['name'])
              ->update();
    }
}
```

Specifying field length

New in version cakephp/migrations: 1.4

If you need to specify a field length, you can do it within brackets in the field type, ie:

```
$ bin/cake bake migration AddFullDescriptionToProducts full_description:string[60]
```

Executing the command line above will generate:

```
<?php
use Migrations\AbstractMigration;

class AddFullDescriptionToProducts extends AbstractMigration
{
    public function change()
    {
        $table = $this->table('products');
        $table->addColumn('full_description', 'string', [
            'default' => null,
            'limit' => 60,
            'null' => false,
        ])
        ->update();
    }
}
```

If no length is specified, lengths for certain type of columns are defaulted:

- string: 255
- integer: 11
- biginteger: 20

Removing a column from a table

In the same way, you can generate a migration to remove a column by using the command line, if the migration name is of the form "RemoveXXXFromYYY":

```
$ bin/cake bake migration RemovePriceFromProducts price
```

creates the file:

```
<?php
use Migrations\AbstractMigration;

class RemovePriceFromProducts extends AbstractMigration
{
    public function up()
    {
        $table = $this->table('products');
        $table->removeColumn('price')
              ->save();
    }
}
```

Note: The *removeColumn* command is not reversible, so must be called in the *up* method. A corresponding *addColumn* call should be added to the *down* method.

Generating migrations from an existing database

If you are dealing with a pre-existing database and want to start using migrations, or to version control the initial schema of your application's database, you can run the `migration_snapshot` command:

```
$ bin/cake bake migration_snapshot Initial
```

It will generate a migration file called **YYYYMMDDHHMMSS_Initial.php** containing all the create statements for all tables in your database.

By default, the snapshot will be created by connecting to the database defined in the `default` connection configuration. If you need to bake a snapshot from a different datasource, you can use the `--connection` option:

```
$ bin/cake bake migration_snapshot Initial --connection my_other_connection
```

You can also make sure the snapshot includes only the tables for which you have defined the corresponding model classes by using the `--require-table` flag:

```
$ bin/cake bake migration_snapshot Initial --require-table
```

When using the `--require-table` flag, the shell will look through your application `Table` classes and will only add the model tables in the snapshot .

The same logic will be applied implicitly if you wish to bake a snapshot for a plugin. To do so, you need to use the `--plugin` option:

```
$ bin/cake bake migration_snapshot Initial --plugin MyPlugin
```

Only the tables which have a `Table` object model class defined will be added to the snapshot of your plugin.

Note: When baking a snapshot for a plugin, the migration files will be created in your plugin's **config/Migrations** directory.

Be aware that when you bake a snapshot, it is automatically added to the phinx log table as migrated.

Generating a diff between two database states

New in version cakephp/migrations: 1.6.0

You can generate a migrations file that will group all the differences between two database states using the `migration_diff` bake template. To do so, you can use the following command:

```
$ bin/cake bake migration_diff NameOfTheMigrations
```

In order to have a point of comparison from your current database state, the migrations shell will generate a "dump" file after each `migrate` or `rollback` call. The dump file is a file containing the full schema state of your database at a given point in time.

Once a dump file is generated, every modifications you do directly in your database management system will be added to the migration file generated when you call the `bake migration_diff` command.

By default, the diff will be created by connecting to the database defined in the `default` connection configuration. If you need to bake a diff from a different datasource, you can use the `--connection` option:

```
$ bin/cake bake migration_diff NameOfTheMigrations --connection my_other_connection
```

If you want to use the diff feature on an application that already has a migrations history, you need to manually create the dump file that will be used as comparison:

```
$ bin/cake migrations dump
```

The database state must be the same as it would be if you just migrated all your migrations before you create a dump file. Once the dump file is generated, you can start doing changes in your database and use the `bake migration_diff` command whenever you see fit.

Note: The migrations shell can not detect column renamings.

The commands

`migrate` : Applying Migrations

Once you have generated or written your migration file, you need to execute the following command to apply the changes to your database:

```
# Run all the migrations
$ bin/cake migrations migrate

# Migrate to a specific version using the ``--target`` option
# or ``-t`` for short.
# The value is the timestamp that is prefixed to the migrations file name::
$ bin/cake migrations migrate -t 20150103081132

# By default, migration files are looked for in the **config/Migrations**
# directory. You can specify the directory using the ``--source`` option
# or ``-s`` for short.
# The following example will run migrations in the **config/Alternate**
# directory
$ bin/cake migrations migrate -s Alternate

# You can run migrations to a different connection than the ``default`` one
# using the ``--connection`` option or ``-c`` for short
$ bin/cake migrations migrate -c my_custom_connection

# Migrations can also be run for plugins. Simply use the ``--plugin`` option
# or ``-p`` for short
$ bin/cake migrations migrate -p MyAwesomePlugin
```

`rollback` : Reverting Migrations

The Rollback command is used to undo previous migrations executed by this plugin. It is the reverse action of the `migrate` command:

```
# You can rollback to the previous migration by using the
# ``rollback`` command::
$ bin/cake migrations rollback

# You can also pass a migration version number to rollback
# to a specific version::
$ bin/cake migrations rollback -t 20150103081132
```

You can also use the `--source`, `--connection` and `--plugin` options just like for the `migrate` command.

`status` : Migrations Status

The Status command prints a list of all migrations, along with their current status. You can use this command to determine which migrations have been run:

```
$ bin/cake migrations status
```

You can also output the results as a JSON formatted string using the `--format` option (or `-f` for short):

```
$ bin/cake migrations status --format json
```

You can also use the `--source`, `--connection` and `--plugin` options just like for the `migrate` command.

`mark_migrated` : Marking a migration as migrated

New in version 1.4.0.

It can sometimes be useful to mark a set of migrations as migrated without actually running them. In order to do this, you can use the `mark_migrated` command. The command works seamlessly as the other commands.

You can mark all migrations as migrated using this command:

```
$ bin/cake migrations mark_migrated
```

You can also mark all migrations up to a specific version as migrated using the `--target` option:

```
$ bin/cake migrations mark_migrated --target=20151016204000
```

If you do not want the targeted migration to be marked as migrated during the process, you can use the `--exclude` flag with it:

```
$ bin/cake migrations mark_migrated --target=20151016204000 --exclude
```

Finally, if you wish to mark only the targeted migration as migrated, you can use the `--only` flag:

```
$ bin/cake migrations mark_migrated --target=20151016204000 --only
```

You can also use the `--source`, `--connection` and `--plugin` options just like for the `migrate` command.

Note: When you bake a snapshot with the `cake bake migration_snapshot` command, the created migration will automatically be marked as migrated.

Deprecated since version 1.4.0: The following way of using the command has been deprecated. Use it only if you are using a version of the plugin < 1.4.0.

This command expects the migration version number as argument:

```
$ bin/cake migrations mark_migrated 20150420082532
```

If you wish to mark all migrations as migrated, you can use the `all` special value. If you use it, it will mark all found migrations as migrated:

```
$ bin/cake migrations mark_migrated all
```

`seed` : Seeding your database

As of 1.5.5, you can use the `migrations` shell to seed your database. This leverages the Phinx library seed feature[190]. By default, seed files will be looked for in the `config/Seeds` directory of your application. Please make sure you follow Phinx instructions to build your seed files[191].

As for migrations, a `bake` interface is provided for seed files:

```
# This will create a ArticlesSeed.php file in the directory config/Seeds of your␣
↪application
# By default, the table the seed will try to alter is the "tableized" version of the␣
↪seed filename
$ bin/cake bake seed Articles

# You specify the name of the table the seed files will alter by using the ``--
↪table`` option
$ bin/cake bake seed Articles --table my_articles_table

# You can specify a plugin to bake into
$ bin/cake bake seed Articles --plugin PluginName

# You can specify an alternative connection when generating a seeder.
$ bin/cake bake seed Articles --connection connection
```

New in version cakephp/migrations: 1.6.4

Options `--data`, `--limit` and `--fields` were added to export data from your database.

As of 1.6.4, the `bake seed` command allows you to create a seed file with data exported from your database by using the `--data` flag:

```
$ bin/cake bake seed --data Articles
```

By default, it will export all the rows found in your table. You can limit the number of rows exported by using the `--limit` option:

```
# Will only export the first 10 rows found
$ bin/cake bake seed --data --limit 10 Articles
```

[190] http://docs.phinx.org/en/latest/seeding.html
[191] http://docs.phinx.org/en/latest/seeding.html#creating-a-new-seed-class

If you only want to include a selection of fields from the table in your seed file, you can use the `--fields` option. It takes the list of fields to include as a comma separated value string:

```
# Will only export the fields `id`, `title` and `excerpt`
$ bin/cake bake seed --data --fields id,title,excerpt Articles
```

Tip: Of course you can use both the `--limit` and `--fields` options in the same command call.

To seed your database, you can use the `seed` subcommand:

```
# Without parameters, the seed subcommand will run all available seeders
# in the target directory, in alphabetical order.
$ bin/cake migrations seed

# You can specify only one seeder to be run using the `--seed` option
$ bin/cake migrations seed --seed ArticlesSeed

# You can run seeders from an alternative directory
$ bin/cake migrations seed --source AlternativeSeeds

# You can run seeders from a plugin
$ bin/cake migrations seed --plugin PluginName

# You can run seeders from a specific connection
$ bin/cake migrations seed --connection connection
```

Be aware that, as opposed to migrations, seeders are not tracked, which means that the same seeder can be applied multiple times.

Calling a Seeder from another Seeder

New in version cakephp/migrations: 1.6.2

Usually when seeding, the order in which to insert the data must be respected to not encounter constraints violations. Since Seeders are executed in the alphabetical order by default, you can use the `\Migrations\AbstractSeed::call()` method to define your own sequence of seeders execution:

```
use Migrations\AbstractSeed;

class DatabaseSeed extends AbstractSeed
{
    public function run()
    {
        $this->call('AnotherSeed');
        $this->call('YetAnotherSeed');

        // You can use the plugin dot syntax to call seeders from a plugin
        $this->call('PluginName.FromPluginSeed');
    }
}
```

Note: Make sure to extend the Migrations plugin `AbstractSeed` class if you want to be able to use the `call()` method. This class was added with release 1.6.2.

`dump` : Generating a dump file for the diff baking feature

The Dump command creates a file to be used with the `migration_diff` bake template:

```
$ bin/cake migrations dump
```

Each generated dump file is specific to the Connection it is generated from (and is suffixed as such). This allows the `bake migration_diff` command to properly compute diff in case your application is dealing with multiple database possibly from different database vendors.

Dump files are created in the same directory as your migrations files.

You can also use the `--source`, `--connection` and `--plugin` options just like for the `migrate` command.

Using Migrations In Plugins

Plugins can also provide migration files. This makes plugins that are intended to be distributed much more portable and easy to install. All commands in the Migrations plugin support the `--plugin` or `-p` option that will scope the execution to the migrations relative to that plugin:

```
$ bin/cake migrations status -p PluginName

$ bin/cake migrations migrate -p PluginName
```

Running Migrations in a non-shell environment

New in version cakephp/migrations: 1.2.0

Since the release of version 1.2 of the migrations plugin, you can run migrations from a non-shell environment, directly from an app, by using the new `Migrations` class. This can be handy in case you are developing a plugin installer for a CMS for instance. The `Migrations` class allows you to run the following commands from the migrations shell:

- migrate
- rollback
- markMigrated
- status
- seed

Each of these commands has a method defined in the `Migrations` class.

Here is how to use it:

```
use Migrations\Migrations;

$migrations = new Migrations();

// Will return an array of all migrations and their status
$status = $migrations->status();

// Will return true if success. If an error occurred, an exception will be thrown
$migrate = $migrations->migrate();
```

```
// Will return true if success. If an error occurred, an exception will be thrown
$rollback = $migrations->rollback();

// Will return true if success. If an error occurred, an exception will be thrown
$markMigrated = $migrations->markMigrated(20150804222900);

// Will return true if success. If an error occurred, an exception will be thrown
$seeded = $migrations->seed();
```

The methods can accept an array of parameters that should match options from the commands:

```
use Migrations\Migrations;

$migrations = new Migrations();

// Will return an array of all migrations and their status
$status = $migrations->status(['connection' => 'custom', 'source' =>
↪'MyMigrationsFolder']);
```

You can pass any options the shell commands would take. The only exception is the `markMigrated` command which is expecting the version number of the migrations to mark as migrated as first argument. Pass the array of parameters as the second argument for this method.

Optionally, you can pass these parameters in the constructor of the class. They will be used as default and this will prevent you from having to pass them on each method call:

```
use Migrations\Migrations;

$migrations = new Migrations(['connection' => 'custom', 'source' =>
↪'MyMigrationsFolder']);

// All the following calls will be done with the parameters passed to the Migrations␣
↪class constructor
$status = $migrations->status();
$migrate = $migrations->migrate();
```

If you need to override one or more default parameters for one call, you can pass them to the method:

```
use Migrations\Migrations;

$migrations = new Migrations(['connection' => 'custom', 'source' =>
↪'MyMigrationsFolder']);

// This call will be made with the "custom" connection
$status = $migrations->status();
// This one with the "default" connection
$migrate = $migrations->migrate(['connection' => 'default']);
```

Tips and tricks

Creating Custom Primary Keys

If you need to avoid the automatic creation of the `id` primary key when adding new tables to the database, you can use the second argument of the `table()` method:

```
<?php
use Migrations\AbstractMigration;

class CreateProductsTable extends AbstractMigration
{
    public function change()
    {
        $table = $this->table('products', ['id' => false, 'primary_key' => ['id']]);
        $table
              ->addColumn('id', 'uuid')
              ->addColumn('name', 'string')
              ->addColumn('description', 'text')
              ->create();
    }
}
```

The above will create a `CHAR(36) id` column that is also the primary key.

Note: When specifying a custom primary key on the command line, you must note it as the primary key in the id field, otherwise you may get an error regarding duplicate id fields, i.e.:

```
$ bin/cake bake migration CreateProducts id:uuid:primary name:string description:text␣
↪created modified
```

Additionally, since Migrations 1.3, a new way to deal with primary key was introduced. To do so, your migration class should extend the new `Migrations\AbstractMigration` class. You can specify a `autoId` property in the Migration class and set it to `false`, which will turn off the automatic `id` column creation. You will need to manually create the column that will be used as a primary key and add it to the table declaration:

```
<?php
use Migrations\AbstractMigration;

class CreateProductsTable extends AbstractMigration
{

    public $autoId = false;

    public function up()
    {
        $table = $this->table('products');
        $table
            ->addColumn('id', 'integer', [
                'autoIncrement' => true,
                'limit' => 11
            ])
            ->addPrimaryKey('id')
            ->addColumn('name', 'string')
            ->addColumn('description', 'text')
            ->create();
    }
}
```

Compared to the previous way of dealing with primary key, this method gives you the ability to have more control over the primary key column definition: unsigned or not, limit, comment, etc.

All baked migrations and snapshot will use this new way when necessary.

Warning: Dealing with primary key can only be done on table creation operations. This is due to limitations for some database servers the plugin supports.

Collations

If you need to create a table with a different collation than the database default one, you can define it with the `table()` method, as an option:

```
<?php
use Migrations\AbstractMigration;

class CreateCategoriesTable extends AbstractMigration
{
    public function change()
    {
        $table = $this
            ->table('categories', [
                'collation' => 'latin1_german1_ci'
            ])
            ->addColumn('title', 'string', [
                'default' => null,
                'limit' => 255,
                'null' => false,
            ])
            ->create();
    }
}
```

Note however this can only be done on table creation : there is currently no way of adding a column to an existing table with a different collation than the table or the database. Only `MySQL` and `SqlServer` supports this configuration key for the time being.

Updating columns name and using Table objects

If you use a CakePHP ORM Table object to manipulate values from your database along with renaming or removing a column, make sure you create a new instance of your Table object after the `update()` call. The Table object registry is cleared after an `update()` call in order to refresh the schema that is reflected and stored in the Table object upon instantiation.

Migrations and Deployment

If you use the plugin when deploying your application, be sure to clear the ORM cache so it renews the column metadata of your tables. Otherwise, you might end up having errors about columns not existing when performing operations on those new columns. The CakePHP Core includes a *ORM Cache Shell* that you can use to perform this operation:

```
$ bin/cake orm_cache clear
```

Be sure to read the *ORM Cache Shell* section of the cookbook if you want to know more about this shell.

Renaming a table

The plugin gives you the ability to rename a table, using the `rename()` method. In your migration file, you can do the following:

```
public function up()
{
    $this->table('old_table_name')
        ->rename('new_table_name');
}
```

Skipping the `schema.lock` file generation

New in version cakephp/migrations: 1.6.5

In order for the diff feature to work, a **.lock** file is generated everytime you migrate, rollback or bake a snapshot, to keep track of the state of your database schema at any given point in time. You can skip this file generation, for instance when deploying on your production environment, by using the `--no-lock` option for the aforementioned command:

```
$ bin/cake migrations migrate --no-lock

$ bin/cake migrations rollback --no-lock

$ bin/cake bake migration_snapshot MyMigration --no-lock
```

CHAPTER 44

Appendices

Appendices contain information regarding the new features introduced in each version and the migration path between versions.

3.x Migration Guide

3.x Migration Guide

Migration guides contain information regarding the new features introduced in each version and the migration path between 2.x and 3.x. If you are currently using 1.x you should first upgrade to 2.x. See the 2.x documentation for the relevant upgrade guides.

3.5 Migration Guide

3.5 Migration Guide

CakePHP 3.5 is an API compatible upgrade from 3.4. This page outlines the changes and improvements made in 3.5.

To upgrade to 3.5.x run the following composer command:

```
php composer.phar require --update-with-dependencies "cakephp/cakephp:3.5.*"
```

Deprecations

The following is a list of deprecated methods, properties and behaviors. These features will continue to function until 4.0.0 after which they will be removed.

- `Cake\Http\Client\CookieCollection` is deprecated. Use `Cake\Http\Cookie\CookieCollection` instead.
- `Cake\View\Helper\RssHelper` is deprecated. Due to infrequent use the RssHelper is deprecated.
- `Cake\Controller\Component\CsrfComponent` is deprecated. Use csrf-middleware instead.
- `Cake\Datasource\TableSchemaInterface` is deprecated. Use `Cake\Database\TableSchemaAwareInterface` instead.

- `Cake\Console\ShellDispatcher` is deprecated. Applications should update to use `Cake\Console\CommandRunner` instead.
- `Cake\Database\Schema\TableSchema::column()` is deprecated. Use `Cake\Database\Schema\TableSchema::getColumn()` instead.
- `Cake\Database\Schema\TableSchema::constraint()` is deprecated. Use `Cake\Database\Schema\TableSchema::getConstraint()` instead.
- `Cake\Database\Schema\TableSchema::index()` is deprecated. Use `Cake\Database\Schema\TableSchema::getIndex()` instead.

Deprecated Combined Get/Set Methods

In the past CakePHP has leveraged 'modal' methods that provide both a get and set mode. These methods complicate IDE autocompletion and our ability to add stricter return types in the future. For these reasons, combined get/set methods are being split into separate get and set methods.

The following is a list of methods that are deprecated and replaced with `getX()` and `setX()` methods:

Cake\Cache\Cache

- `config()`
- `registry()`

Cake\Console\Shell

- `io()`

Cake\Console\ConsoleIo

- `outputAs()`

Cake\Console\ConsoleOutput

- `outputAs()`

Cake\Database\Connection

- `logger()`

Cake\Database\TypedResultInterface

- `returnType()`

Cake\Database\TypedResultTrait

- `returnType()`

Cake\Database\Log\LoggingStatement

- `logger()`

Cake\Datasource\ModelAwareTrait

- `modelType()`

Cake\Database\Query

- getter part of `valueBinder()` (now `getValueBinder()`)

Cake\Database\Schema\TableSchema

- `columnType()`

Cake\Datasource\QueryTrait

- getter part of `eagerLoaded()` (now `isEagerLoaded()`)

Cake\Event\EventDispatcherInterface

- `eventManager()`

Cake\Event\EventDispatcherTrait

- `eventManager()`

Cake\Error\Debugger

- `outputAs()` (now `getOutputFormat()` / `setOutputFormat()`)

Cake\Http\ServerRequest

- `env()` (now `getEnv()` / `withEnv()`)
- `charset()` (now `getCharset()` / `withCharset()`)

Cake\I18n\I18n

- `locale()`
- `translator()`
- `defaultLocale()`
- `defaultFormatter()`

Cake\ORM\Association\BelongsToMany

- `sort()`

Cake\ORM\LocatorAwareTrait

- `tableLocator()`

Cake\ORM\EntityTrait

- `invalid()` (now `getInvalid()`, `setInvalid()`, `setInvalidField()`, and `getInvalidField()`)

Cake\ORM\Table

- `validator()`

Cake\Routing\RouteBuilder

- `extensions()`
- `routeClass()`

Cake\Routing\RouteCollection

- `extensions()`

Cake\TestSuite\TestFixture

- `schema()`

Cake\Utility\Security

- `salt()`

Cake\View\View

- `template()`
- `layout()`

- `theme()`
- `templatePath()`
- `layoutPath()`
- `autoLayout()` (now `isAutoLayoutEnabled()` / `enableAutoLayout()`)

Behavior Changes

While these changes are API compatible, they represent minor variances in behavior that may affect your application:

- `BehaviorRegistry`, `HelperRegistry` and `ComponentRegistry` will now raise exceptions when `unload()` is called with an unknown object name. This change should help find errors easier by making possible typos more visible.
- `HasMany` associations now gracefully handle empty values set for the association property, similar to `BelongsToMany` associations - that is they treat `false`, `null`, and empty strings the same way as empty arrays. For `HasMany` associations this now results in all associated records to be deleted/unlinked when the `replace` save strategy is being used. As a result this allows you to use forms to delete/unlink all associated records by passing an empty string. Previously this would have required custom marshalling logic.
- `ORM\Table::newEntity()` now only marks association properties dirty if the marshalled association record is dirty. In scenarios where an association entity is created that contains no properties the empty record will not be flagged for persistence.
- `Http\Client` no longer uses the `cookie()` method results when building requests. Instead the `Cookie` header and internal CookieCollection are used. This should only effect applications that have a custom HTTP adapter in their clients.
- Multi-word subcommand names previouly required camelBacked names to be used when invoking shells. Now subcommands can be invoked with underscored_names. For example: `cake tool initMyDb` can now be called with `cake tool init_my_db`. If your shells previously bound two subcommands with different inflections, only the last bound command will function.
- `SecurityComponent` will blackhole post requests that have no request data now. This change helps protect actions that create records using database defaults alone.
- `Cake\ORM\Table::addBehavior()` and `removeBehavior()` now return `$this` to assist in defining table objects in a fluent fashion.
- Cache engines no longer throw an exception when they fail or are misconfigured, but instead fall back to the noop `NullEngine`. Fallbacks can also be *configured* on a per-engine basis.
- `Cake\Database\Type\DateTimeType` will now marshal ISO-8859-1 formatted datetime strings (e.g. 2017-07-09T12:33:00+00:02) in addition to the previously accepted format. If you have a subclass of DateTimeType you may need to update your code.

New Features

Scoped Middleware

Middleware can now be conditionally applied to routes in specific URL scopes. This allows you to build specific stacks of middleware for different parts of your application without having to write URL checking code in your middleware. See the *Connecting Scoped Middleware* section for more information.

New Console Runner

3.5.0 adds `Cake\Console\CommandRunner`. This class alongside `Cake\Console\CommandCollection` integrate the CLI environment with the new `Application` class. Application classes can now implement a `console()` hook that allows them to have full control over which CLI commands are exposed, how they are named and how the shells get their dependencies. Adopting this new class requires replacing the contents of your `bin/cake.php` file with the following file[192].

Cache Engine Fallbacks

Cache engines can now be configured with a `fallback` key that defines a cache configuration to fall back to if the engine is misconfigured (or unavailable). See *Configuring Cache Fallbacks* for more information on configuring fallbacks.

dotenv Support added to Application Skeleton

The application skeleton now features a 'dotenv' integration making it easier to use environment variables to configure your application. See the *Environment Variables* section for more information.

Console Integration Testing

The `Cake\TestSuite\ConsoleIntegrationTestCase` class was added to make integration testing console applications easier. For more information, visit the *Console Integration Testing* section. This test class is fully compatible with the current shell dispatcher as well as the new `Cake\Console\CommandRunner`.

Collection

- `Cake\Collection\Collection::avg()` was added.
- `Cake\Collection\Collection::median()` was added.

Core

- `Cake\Core\Configure::read()` now supports default values if the desired key does not exist.
- `Cake\Core\ObjectRegistry` now implements the `Countable` and `IteratorAggregate` interfaces.

Console

- `Cake\Console\ConsoleOptionParser::setHelpAlias()` was added. This method allows you to set the command name used when generating help output. Defaults to `cake`.
- `Cake\Console\CommandRunnner` was added replacing `Cake\Console\ShellDispatcher`.
- `Cake\Console\CommandCollection` was added to provide an interface for applications to define the command line tools they offer.

[192] https://github.com/cakephp/app/tree/3.next/bin/cake.php

Database

- SQLite driver had the `mask` option added. This option lets you set the file permissions on the SQLite database file when it is created.

Datasource

- `Cake\Datasource\SchemaInterface` was added.
- New abstract types were added for `smallinteger` and `tinyinteger`. Existing `SMALLINT` and `TINYINT` columns will now be reflected as these new abstract types. `TINYINT(1)` columns will continue to be treated as boolean columns in MySQL.
- `Cake\Datasource\PaginatorInterface` was added. The `PaginatorComponent` now uses this interface to interact with paginators. This allows other ORM-like implementations to be paginated by the component.
- `Cake\Datasource\Paginator` was added to paginate ORM/Database Query instances.

Event

- `Cake\Event\EventManager::on()` and `off()` methods are now chainable making it simpler to set multiple events at once.

Http

- New `Cookie` & `CookieCollection` classes have been added. These classes allow you to work with cookies in an object-orientated way, and are available on `Cake\Http\ServerRequest`, `Cake\Http\Response`, and `Cake\Http\Client\Response`. See the *Cookies* and *Setting Cookies* for more information.
- New middleware has been added to make applying security headers easier. See security-header-middleware for more information.
- New middleware has been added to transparently encrypt cookie data. See encrypted-cookie-middleware for more information.
- New middleware has been added to make protecting against CSRF easier. See csrf-middleware for more information.
- `Cake\Http\Client::addCookie()` was added to make it easy to add cookies to a client instance.

InstanceConfigTrait

- `InstanceConfigTrait::getConfig()` now takes a 2nd parameter `$default`. If no value is available for the specified `$key`, the `$default` value will be returned.

ORM

- `Cake\ORM\Query::contain()` now allows you to call it without the wrapping array when containing a single association. `contain('Comments',function () { ... });` will now work. This makes `contain()` consistent with other eagerloading related methods like `leftJoinWith()` and `matching()`.

Routing

- `Cake\Routing\Router::reverseToArray()` was added. This method allow you to convert a request object into an array that can be used to generate URL strings.
- `Cake\Routing\RouteBuilder::resources()` had the `path` option added. This option lets you make the resource path and controller name not match.
- `Cake\Routing\RouteBuilder` now has methods to create routes for specific HTTP methods. e.g `get()` and `post()`.
- `Cake\Routing\RouteBuilder::loadPlugin()` was added.
- `Cake\Routing\Route` now has fluent methods for defining options.

TestSuite

- `TestCase::loadFixtures()` will now load all fixtures when no arguments are provided.
- `IntegrationTestCase::head()` was added.
- `IntegrationTestCase::options()` was added.
- `IntegrationTestCase::disableErrorHandlerMiddleware()` was added to make debugging errors easier in integration tests.

Validation

- `Cake\Validation\Validator::scalar()` was added to ensure that fields do not get non-scalar data.
- `Cake\Validation\Validator::regex()` was added for a more convenient way to validate data against a regex pattern.
- `Cake\Validation\Validator::addDefaultProvider()` was added. This method lets you inject validation providers into all the validators created in your application.
- `Cake\Validation\ValidatorAwareInterface` was added to define the methods implemented by `Cake\Validation\ValidatorAwareTrait`.

View

- `Cake\View\Helper\PaginatorHelper::limitControl()` was added. This method lets you create a form with a select box for updating the limit value on a paginated result set.

3.4 Migration Guide

3.4 Migration Guide

CakePHP 3.4 is an API compatible upgrade from 3.3. This page outlines the changes and improvements made in 3.4.

Minimum PHP 5.6 Required

CakePHP 3.4 requires at least PHP 5.6.0 as PHP 5.5 is no longer supported and won't receive any security fixes anymore.

Deprecations

The following is a list of deprecated methods, properties and behaviors. These features will continue to function until 4.0.0 after which they will be removed.

Request & Response Deprecations

The bulk of deprecations for 3.4 are in the `Request` and `Response` objects. The existing methods that modify objects in-place are now deprecated, and superseded by methods that follow the immutable object patterns described in the PSR-7 standard.

Several properties on `Cake\Network\Request` have been deprecated:

- `Request::$params` is deprecated. Use `Request::getAttribute('params')` instead.
- `Request::$data` is deprecated. Use `Request::getData()` instead.
- `Request::$query` is deprecated. Use `Request::getQueryParams()` instead.
- `Request::$cookies` is deprecated. Use `Request::getCookie()` instead.
- `Request::$base` is deprecated. Use `Request::getAttribute('base')` instead.
- `Request::$webroot` is deprecated. Use `Request::getAttribute('webroot')` instead.
- `Request::$here` is deprecated. Use `Request::getRequestTarget()` instead.
- `Request::$_session` was renamed to `Request::$session`.

A number of methods on `Cake\Network\Request` have been deprecated:

- `__get()` & `__isset()` methods are deprecated. Use `getParam()` instead.
- `method()` is deprecated. Use `getMethod()` instead.
- `setInput()` is deprecated. Use `withBody()` instead.
- The `ArrayAccess` methods have all been deprecated.
- `Request::param()` is deprecated. Use `Request::getParam()` instead.
- `Request::data()` is deprecated. Use `Request::getData()` instead.
- `Request::query()` is deprecated. Use `Request::getQuery()` instead.
- `Request::cookie()` is deprecated. Use `Request::getCookie()` instead.

Several methods on `Cake\Network\Response` have been deprecated because they either overlap the PSR-7 methods, or are made obsolete by the PSR-7 stack:

- `Response::header()` is deprecated. Use `getHeaderLine()`, `hasHeader()` or `Response::getHeader()` instead.
- `Response::body()` is deprecated. Use `Response::withBody()` instead.
- `Response::statusCode()` is deprecated. Use `Response::getStatusCode()` instead.
- `Response::httpCodes()` This method should no longer be used. CakePHP now supports all standards recommended status codes.
- `Response::protocol()` is deprecated. Use `Response::getProtocolVersion()` instead.
- `send()`, `sendHeaders()`, `_sendHeader()`, `_sendContent()`, `_setCookies()`, `_setContentType()`, and `stop()` are deprecated and made obsolete by the PSR-7 HTTP stack.

With responses heading towards immutable object patterns as recommended by the PSR-7 standards, a number of 'helper' methods in `Response` have been deprecated and immutable variants are now recommended:

- `Response::location()` would become `Response::withLocation()`
- `Response::disableCache()` would become `Response::withDisabledCache()`
- `Response::type()` would become `Response::withType()`
- `Response::charset()` would become `Response::withCharset()`
- `Response::cache()` would become `Response::withCache()`
- `Response::modified()` would become `Response::withModified()`
- `Response::expires()` would become `Response::withExpires()`
- `Response::sharable()` would become `Response::withSharable()`
- `Response::maxAge()` would become `Response::withMaxAge()`
- `Response::vary()` would become `Response::withVary()`
- `Response::etag()` would become `Response::withEtag()`
- `Response::compress()` would become `Response::withCompression()`
- `Response::length()` would become `Response::withLength()`
- `Response::mustRevalidate()` would become `Response::withMustRevalidate()`
- `Response::notModified()` would become `Response::withNotModified()`
- `Response::cookie()` would become `Response::withCookie()`
- `Response::file()` would become `Response::withFile()`
- `Response::download()` would become `Response::withDownload()`

Please see the *Adopting Immutable Responses* section for more information before updating your code as using responses through the immutable methods will require additional changes.

Other Deprecations

- The public properties on `Cake\Event\Event` are deprecated, new methods have been added to read/write the relevant properties.
- `Event::name()` is deprecated. Use `Event::getName()` instead.
- `Event::subject()` is deprecated. Use `Event::getSubject()` instead.
- `Event::result()` is deprecated. Use `Event::getResult()` instead.
- `Event::data()` is deprecated. Use `Event::getData()` instead.
- The `Auth.redirect` session variable is no longer used. Instead a query string parameter is used to store the redirect URL. This has the additional effect of removing the ability to store a redirect URL in the session outside of login scenarios.
- `AuthComponent` no longer stores redirect URLs when the unauthorized URL is not a `GET` action.
- The `ajaxLogin` option for `AuthComponent` is deprecated. You should use the `403` status code to trigger the correct behavior in clientside code now.
- The `beforeRedirect` method of `RequestHandlerComponent` is now deprecated.

- The `306` status code in `Cake\Network\Response` is now deprecated and has its status phrase changed to 'Unused' as this status code is non-standard.
- `Cake\Database\Schema\Table` has been renamed to `Cake\Database\Schema\TableSchema`. The previous name was confusing to a number of users.
- The `fieldList` option for `Cake\ORM\Table::newEntity()` and `patchEntity()` has been renamed to `fields` to be more consistent with other parts of the ORM.
- `Router::parse()` is deprecated. `Router::parseRequest()` should be used instead as it accepts a request and gives more control/flexibility in handling incoming requests.
- `Route::parse()` is deprecated. `Route::parseRequest()` should be used instead as it accepts a request and gives more control/flexibility in handling incoming requests.
- `FormHelper::input()` is deprecated. Use `FormHelper::control()` instead.
- `FormHelper::inputs()` is deprecated. Use `FormHelper::controls()` instead.
- `FormHelper::allInputs()` is deprecated. Use `FormHelper::allControls()` instead.
- `Mailer::layout()` is deprecated. Use `Mailer::setLayout()` provided by `Mailer::__call()` instead.

Deprecated Combined Get/Set Methods

In the past CakePHP has leveraged 'modal' methods that provide both a get/set mode. These methods complicate IDE autocompletion and our ability to add stricter return types in the future. For these reasons, combined get/set methods are being split into separate get and set methods.

The following is a list of methods that are deprecated and replaced with `getX()` and `setX()` methods:

`Cake\Core\InstanceConfigTrait`

- `config()`

`Cake\Core\StaticConfigTrait`

- `config()`
- `dsnClassMap()`

`Cake\Console\ConsoleOptionParser`

- `command()`
- `description()`
- `epilog()`

`Cake\Database\Connection`

- `driver()`
- `schemaCollection()`
- `useSavePoints()` (now `enableSavePoints()`/`isSavePointsEnabled()`)

`Cake\Database\Driver`

- `autoQuoting` (now `enableAutoQuoting()`/`isAutoQuotingEnabled()`)

`Cake\Database\Expression\FunctionExpression`

- `name()`

Cake\Database\Expression\QueryExpression

- `tieWith()` (now `setConjunction()`/`getConjunction()`)

Cake\Database\Expression\ValuesExpression

- `columns()`
- `values()`
- `query()`

Cake\Database\Query

- `connection()`
- `selectTypeMap()`
- `bufferResults()` (now `enableBufferedResults()`/`isBufferedResultsEnabled()`)

Cake\Database\Schema\CachedCollection

- `cacheMetadata()`

Cake\Database\Schema\TableSchema

- `options()`
- `temporary()` (now `setTemporary()`/`isTemporary()`)

Cake\Database\TypeMap

- `defaults()`
- `types()`

Cake\Database\TypeMapTrait

- `typeMap()`
- `defaultTypes()`

Cake\ORM\Association

- `name()`
- `cascadeCallbacks()`
- `source()`
- `target()`
- `conditions()`
- `bindingKey()`
- `foreignKey()`
- `dependent()`
- `joinType()`
- `property()`
- `strategy()`
- `finder()`

Cake\ORM\Association\BelongsToMany

- `targetForeignKey()`

- saveStrategy()
- conditions()

Cake\ORM\Association\HasMany

- saveStrategy()
- foreignKey()
- sort()

Cake\ORM\Association\HasOne

- foreignKey()

Cake\ORM\EagerLoadable

- config()
- setter part of canBeJoined() (now setCanBeJoined())

Cake\ORM\EagerLoader

- matching() (getMatching() will have to be called after setMatching() to keep the old behavior)
- autoFields() (now enableAutoFields()/isAutoFieldsEnabled())

Cake\ORM\Locator\TableLocator

- config()

Cake\ORM\Query

- eagerLoader()
- hydrate() (now enableHydration()/isHydrationEnabled())
- autoFields() (now enableAutoFields()/isAutoFieldsEnabled())

Cake\ORM\Table

- table()
- alias()
- registryAlias()
- connection()
- schema()
- primaryKey()
- displayField()
- entityClass()

Cake\Mailer\Email

- from()
- sender()
- replyTo()
- readReceipt()
- returnPath()

- `to()`
- `cc()`
- `bcc()`
- `charset()`
- `headerCharset()`
- `emailPattern()`
- `subject()`
- `template()` (now `setTemplate()`/`getTemplate()` and `setLayout()`/`getLayout()`)
- `viewRender()` (now `setViewRenderer()`/`getViewRenderer()`)
- `viewVars()`
- `theme()`
- `helpers()`
- `emailFormat()`
- `transport()`
- `messageId()`
- `domain()`
- `attachments()`
- `configTransport()`
- `profile()`

Cake\Validation\Validator

- `provider()`

Cake\View\StringTemplateTrait

- `templates()`

Cake\View\ViewBuilder

- `templatePath()`
- `layoutPath()`
- `plugin()`
- `helpers()`
- `theme()`
- `template()`
- `layout()`
- `options()`
- `name()`
- `className()`
- `autoLayout()` (now `enableAutoLayout()`/`isAutoLayoutEnabled()`)

Adopting Immutable Responses

Before you migrate your code to use the new response methods you should be aware of the conceptual differences the new methods have. The immutable methods are generally indicated using a `with` prefix. For example, `withLocation()`. Because these methods operate in an immutable context, they return *new* instances which you need to assign to variables or properties. If you had controller code that looked like:

```
$response = $this->response;
$response->location('/login')
$response->header('X-something', 'a value');
```

If you were to simply find & replace method names your code would break. Instead you must now use code that looks like:

```
$this->response = $this->response
    ->withLocation('/login')
    ->withHeader('X-something', 'a value');
```

There are a few key differences:

1. The result of your changes is re-assigned to `$this->response`. This is critical to preserving the intent of the above code.
2. The setter methods can all be chained together. This allows you to skip storing all the intermediate objects.

Component Migration Tips

In previous versions of CakePHP, Components often held onto references to both the request and response, in order to make changes later. Before you adopt the immutable methods you should use the response attached to the Controller:

```
// In a component method (not a callback)
$this->response->header('X-Rate-Limit', $this->remaining);

// Should become
$controller = $this->getController();
$controller->response = $controller->response->withHeader('X-Rate-Limit', $this->
↪remaining);
```

In component callbacks you can use the event object to access the response/controller:

```
public function beforeRender($event)
{
    $controller = $event->getSubject();
    $controller->response = $controller->response->withHeader('X-Teapot', 1);
}
```

Tip: Instead of holding onto references of Responses, always get the current response from the controller, and re-assign the response property when you are done.

Behavior Changes

While these changes are API compatible, they represent minor variances in behavior that may affect your application:

- `ORM\Query` results will not typecast aliased columns based on the original column's type. For example if you alias `created` to `created_time` you will now get a `Time` object back instead of a string.
- Internal ORM traits used to build Association classes have been removed and replaced with new internal APIs. This shouldn't impact your applications, but may if you have created custom association types.
- `AuthComponent` now uses a query string to store the redirect URL when an unauthenticated user is redirected to the login page. Previously, this redirect was stored in the session. Using the query string allows for better multi-browser experience.
- Database Schema reflection now treats unknown column types as `string` and not `text`. A visible impact of this is that `FormHelper` will generate text inputs instead of textarea elements for unknown column types.
- `AuthComponent` no longer stores the flash messages it creates under the 'auth' key. They are now rendered with the 'error' template under the 'default' flash message key. This simplifies using `AuthComponent`.
- `Mailer\Email` will now autodetect attachment content types using `mime_content_type` if a content-type is not provided. Previously attachments would have defaulted to 'application/octet-stream'.
- CakePHP now uses the `...` operator in place of `call_user_func_array()`. If you are passing associative arrays, you should update your code to pass a numerically indexed array using `array_values()` for the following methods:
  - `Cake\Mailer\Mailer::send()`
  - `Cake\Controller\Controller::setAction()`
  - `Cake\Http\ServerRequest::is()`

Visibility Changes

- `MailerAwareTrait::getMailer()` will now become protected.
- `CellTrait::cell()` will now become protected.

If the above traits are used in controllers, their public methods could be accessed by default routing as actions. These changes help protect your controllers. If you need the methods to remain public you will need to update your `use` statement to look like:

```
use CellTrait {
    cell as public;
}
use MailerAwareTrait {
    getMailer as public;
}
```

Collection

- `CollectionInterface::chunkWithKeys()` was added. User land implementations of the `CollectionInterface` will need to implement this method now.
- `Collection::chunkWithKeys()` was added.

Error

- `Debugger::setOutputMask()` and `Debugger::outputMask()` were added. These methods allow you to configure properties/array keys that should be masked from output generated by Debugger (for instance,

when calling `debug()`).

Event

- `Event::getName()` was added.
- `Event::getSubject()` was added.
- `Event::getData()` was added.
- `Event::setData()` was added.
- `Event::getResult()` was added.
- `Event::setResult()` was added.

I18n

- You can now customize the behavior of the fallback message loader. See *Creating Generic Translators* for more information.

Routing

- `RouteBuilder::prefix()` now accepts an array of defaults to add to each connected route.
- Routes can now match only specific hosts through the `_host` option.

Email

- `Email::setPriority()`/`Email::getPriority()` have been added.

HtmlHelper

- `HtmlHelper::scriptBlock()` no longer wraps the JavaScript code in `<![CDATA[ ]]` tag by default. The `safe` option which controls this behavior now defaults to `false`. Using `<![CDATA[ ]]` tag was only required for XHTML which is no longer the dominant doctype used for HTML pages.

BreadcrumbsHelper

- `BreadcrumbsHelper::reset()` was added. This method lets you clear out existing crumbs.

PaginatorHelper

- `PaginatorHelper::numbers()` now uses an HTML elipsis instead of '...' in the default templates.
- `PaginatorHelper::total()` was added to enable reading the total number of pages for the currently paginated results.
- `PaginatorHelper::generateUrlParams()` was added as a lower level URL building method.
- `PaginatorHelper::meta()` can now create links for 'first', 'last'.

FormHelper

- You can now configure the sources which FormHelper reads from. This makes creating GET forms simpler. See *Getting form values from the query string* for more information.
- `FormHelper::control()` was added.
- `FormHelper::controls()` was added.
- `FormHelper::allControls()` was added.

Validation

- `Validation::falsey()` and `Validation::truthy()` were added.

TranslateBehavior

- `TranslateBehavior::translationField()` was added.

PluginShell

- `cake plugin load` and `cake plugin unload` now support a `--cli` option, which updates the `bootstrap_cli.php` instead.

TestSuite

- Support to `PHPUnit 6` was added. With this framework version requiring PHP 5.6.0 as a minimum version, the supported version of PHPUnit are now `^5.7|^6.0`

3.3 Migration Guide

3.3 Migration Guide

CakePHP 3.3 is an API compatible upgrade from 3.2. This page outlines the changes and improvements made in 3.3.

Deprecations

- `Router::mapResources()` is deprecated. Use routing scopes and `$routes->resources()` instead.
- `Router::redirect()` is deprecated. Use routing scopes and `$routes->redirect()` instead.
- `Router::parseNamedParams()` is deprecated. Named parameter backwards compatibility will be removed in 4.0.0
- `Cake\Http\Client\Response` has had the following methods deprecated because they overlap with PSR-7 interface methods:
  - `statusCode()` use `getStatusCode()` instead.
  - `encoding()` use `getEncoding()` instead.
  - `header()` use `getHeaderLine()` instead.

  - `cookie()` use `getCookie()` instead.
  - `version()` use `getProtocolVersion()` instead.
- Dispatcher Filters are now deprecated. Use /controllers/middleware instead.
- `RequestActionTrait` has been deprecated. Refactor your code to use *View Cells* instead.
- `Cake\\Utility\\Crypto\\Mcrypt` engine has been deprecated as the `mcrypt` extension is deprecated in PHP 7.1. Use the `openssl` and `Cake\Utility\Crypto\Openssl` instead.

Behavior Changes

While these changes are API compatible, they represent minor variances in behavior that may effect your application:

- The default JSON encode format for Date and DateTime instances is now ISO-8601. This means that the timezone value contains a `:`. For example `2015-11-06T00:00:00+03:00`
- `Controller::referer()` now consistently omits the application base path when generating application local URLs. Previously string URLs would have the base path prepended to them, while array URLs would not.
- The default `ErrorController` no longer disables `Auth` and `Security` components, as it does not extend `AppController`. If you are enabling these components through events, you will need to update your code.
- `Entity::clean` now cleans original values, clearing them on save. This behavior was a bug as the entity's original state should not be retained after a save, but instead reflect the new state of the entity.

PSR-7 Middleware Support Added

In tandem with the deprecation of Dispatcher Filters, support for PSR-7 middleware has been added. Middleware is part of the new HTTP stack that is an opt-in component of CakePHP 3.3.0. By using the new HTTP stack, you can take advantage of features like:

- Using middleware from plugins, and libraries outside of CakePHP.
- Leverage the same response object methods in both the responses you get from `Http\Client` and the responses your application generates.
- Be able to augment the response objects emitted by error handling and asset delivery.

See the /controllers/middleware chapter and adding-http-stack sections for more information and how to add the new HTTP stack to an existing application.

Http Client is now PSR-7 Compatible

`Cake\Network\Http\Client` has been moved to `Cake\Http\Client`. Its request and response objects now implement the PSR-7 interfaces[193]. Several methods on `Cake\Http\Client\Response` are now deprecated, see above for more information.

ORM Improvements

- Additional support has been added for mapping complex data types. This makes it easier to work with geospatial types, and data that cannot be represented by strings in SQL queries. See the *Mapping Custom Datatypes to SQL Expressions* for more information.

[193] http://www.php-fig.org/psr/psr-7/

- A new `JsonType` was added. This new type lets you use the native JSON types available in MySQL and Postgres. In other database providers the `json` type will map to `TEXT` columns.
- `Association::unique()` was added. This method proxies the target table's `unique()` method, but ensures that association conditions are applied.
- `isUnique` rules now apply association conditions.
- When entities are converted into JSON, the associated objects are no longer converted to an array first using `toArray()`. Instead, the `jsonSerialize()` method will be invoked on all associated entities. This gives you more flexibility and control on which properties are exposed in JSON representations of your entities.
- `Table::newEntity()` and `Table::patchEntity()` will now raise an exception when an unknown association is in the 'associated' key.
- `RulesChecker::validCount()` was added. This new method allows to apply rules to the number of associated records an entity has.
- The `allowNullableNulls` option was added to the `existsIn` rule. This option allows rules to pass when some columns are null.
- Saving translated records is now simpler. See the *Saving Multiple Translations* for more information.

Multiple Pagination Support Added

You can now paginate multiple queries in a single controller action/view template. See the *Paginating Multiple Queries* section for more details.

Cache Shell Added

To help you better manage cached data from the CLI environment, a shell command has been added that exposes methods for clearing cached data:

```
// Clear one cache config
bin/cake cache clear <configname>

// Clear all cache configs
bin/cake cache clear_all
```

FormHelper

- FormHelper will now automatically set the default value of fields to the default value defined in your database columns. You can disable this behavior by setting `schemaDefault` option to false.

Validation

- `Validator::requirePresence()`, `Validator::allowEmpty()` and `Validator::notEmpty()` now accept a list of fields. This allows you to more concisely define the fields that are required.

StringTemplate

`StringTemplate::format()` now throws an exception instead of returning `null` when requested template is not found.

Other Enhancements

- `Collection::transpose()` was added. This method allows you to tranpose the rows and columns of a matrix with equal length rows.
- The default `ErrorController` now loads `RequestHandlerComponent` to enable `Accept` header based content-type negotiation for error pages.

Routing

- `Router::parse()`, `RouteCollection::parse()` and `Route::parse()` had a `$method` argument added. It defaults to 'GET'. This new parameter reduces reliance on global state, and necessary for the PSR-7 work integration to be done.
- When building resource routes, you can now define a prefix. This is useful when defining nested resources as you can create specialized controllers for nested resources.
- Dispatcher Filters are now deprecated. Use /controllers/middleware instead.

Console

- Shell tasks that are invoked directly from the CLI no longer have their `_welcome` method invoked. They will also have the `requested` parameter set now.
- `Shell::err()` will now apply the 'error' style to text. The default styling is red text.

Request

- `Request::is()` and `Request::addDetector()` now supports additional arguments in detectors. This allows detector callables to operate on additional parameters.

Debugging Functions

- The `pr()`, `debug()`, and `pj()` functions now return the value being dumped. This makes them easier to use when values are being returned.
- *`dd()`* has been added to completely halt execution.

3.2 Migration Guide

3.2 Migration Guide

CakePHP 3.2 is an API compatible upgrade from 3.1. This page outlines the changes and improvements made in 3.2.

Minimum PHP 5.5 Required

CakePHP 3.2 requires at least PHP 5.5.9. By adopting PHP 5.5 we can provide better Date and Time libraries and remove dependencies on password compatibility libraries.

Deprecations

As we continue to improve CakePHP, certain features are deprecated as they are replaced with better solutions. Deprecated features will not be removed until 4.0:

- `Shell::error()` is deprecated because its name does not clearly indicate that it both outputs a message and stops execution. Use `Shell::abort()` instead.
- `Cake\Database\Expression\QueryExpression::type()` is deprecated. Use `tieWith()` instead.
- `Cake\Database\Type\DateTimeType::$dateTimeClass` is deprecated. Use DateTimeType::useMutable() or DateTimeType::useImmutable() instead.
- `Cake\Database\Type\DateType::$dateTimeClass` is deprecated. Use `DateTimeType::useMutable()` or `DateType::useImmutable()` instead.
- `Cake\ORM\ResultSet::_calculateTypeMap()` is now unused and deprecated.
- `Cake\ORM\ResultSet::_castValues()` is now unused and deprecated.
- The `action` key for `FormHelper::create()` has been deprecated. You should use the `url` key directly.

Disabling Deprecation Warnings

Upon upgrading you may encounter several deprecation warnings. These warnings are emitted by methods, options and functionality that will be removed in CakePHP 4.x, but will continue to exist throughout the lifetime of 3.x. While we recommend addressing deprecation issues as they are encountered, that is not always possible. If you'd like to defer fixing deprecation notices, you can disable them in your **config/app.php**:

```
'Error' => [
    'errorLevel' => E_ALL & ~E_DEPRECATED & ~E_USER_DEPRECATED,
]
```

The above error level will suppress deprecation warnings from CakePHP.

New Enhancements

Carbon Replaced with Chronos

The Carbon library has been replaced with cakephp/chronos. This new library is a fork of Carbon with no additional dependencies. It also offer a calendar date object, and immutable versions of both date and datetime objects.

New Date Object

The `Date` class allows you to cleanly map `DATE` columns into PHP objects. Date instances will always fix their time to `00:00:00 UTC`. By default the ORM creates instances of `Date` when mapping `DATE` columns now.

New Immutable Date and Time Objects

The `FrozenTime`, and `FrozenDate` classes were added. These classes offer the same API as the `Time` object has. The frozen classes provide immutable variants of `Time` and `Date`. By using immutable objects, you can prevent accidental mutations. Instead of in-place modifications, modifier methods return *new* instances:

```
use Cake\I18n\FrozenTime;

$time = new FrozenTime('2016-01-01 12:23:32');
$newTime = $time->modify('+1 day');
```

In the above code `$time` and `$newTime` are different objects. The `$time` object retains its original value, while `$newTime` has the modified value. See the *Immutable Dates and Times* section for more information. As of 3.2, the ORM can map date/datetime columns into immutable objects. See the *Enabling Immutable DateTime Objects* section for more information.

CorsBuilder Added

In order to make setting headers related to Cross Origin Requests (CORS) easier, a new `CorsBuilder` has been added. This class lets you define CORS related headers with a fluent interface. See *Setting Cross Origin Request Headers (CORS)* for more information.

RedirectRoute raises an exception on redirect

`Router::redirect()` now raises `Cake\Network\Routing\RedirectException` when a redirect condition is reached. This exception is caught by the routing filter and converted into a response. This replaces calls to `response->send()` and allows dispatcher filters to interact with redirect responses.

ORM Improvements

- Containing the same association multiple times now works as expected, and the query builder functions are now stacked.
- Function expressions now correctly cast their results. This means that expressions like `$query->func()->current_date()` will return datetime instances.
- Field data that fails validation can now be accessed in entities via the `invalid()` method.
- Entity accessor method lookups are now cached and perform better.

Improved Validator API

The Validator object has a number of new methods that make building validators less verbose. For example adding validation rules to a username field can now look like:

```
$validator->email('username')
    ->ascii('username')
    ->lengthBetween('username', [4, 8]);
```

Console Improvements

- `Shell::info()`, `Shell::warn()` and `Shell::success()` were added. These helper methods make using commonly used styling simpler.
- `Cake\Console\Exception\StopException` was added.
- `Shell::abort()` was added to replace `error()`.

StopException Added

`Shell::_stop()` and `Shell::error()` no longer call `exit()`. Instead they raise `Cake\Console\Exception\StopException`. If your shells/tasks are catching `\Exception` where these methods would have been called, those catch blocks will need to be updated so they don't catch the `StopException`. By not calling `exit()` testing shells should be easier and require fewer mocks.

Helper initialize() added

Helpers can now implement an `initialize(array $config)` hook method like other class types.

Fatal Error Memory Limit Handling

A new configuration option `Error.extraFatalErrorMemory` can be 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.

Migration Steps

Updating setToStringFormat()

Before CakePHP 3.2 using Time::setToStringFormat() was working on Date Objects as well. After upgrading you will need to add Date::setToStringFormat() in addition to see the formatted Date again.

3.1 Migration Guide

3.1 Migration Guide

CakePHP 3.1 is a fully API compatible upgrade from 3.0. This page outlines the changes and improvements made in 3.1.

Routing

- The default route class has been changed to `DashedRoute` in the `cakephp/app` repo. Your current code base is not affected by this, but it is recommended to use this route class from now on.
- Name prefix options were added to the various route builder methods. See the *Using Named Routes* section for more information.

Console

- `Shell::dispatchShell()` no longer outputs the welcome message from the dispatched shell.
- The `breakpoint()` helper function has been added. This function provides a snippet of code that can be put into `eval()` to trigger an interactive console. This is very helpful when debugging in test cases, or other CLI scripts.
- The `--verbose` and `--quiet` console options now control stdout/stderr logging output levels.

Shell Helpers Added

- Console applications can now create helper classes that encapsulate re-usable blocks of output logic. See the *Shell Helpers* section for more information.

RoutesShell

- RoutesShell has been added and now provides you a simple to use CLI interface for testing and debugging routes. See the *Routes Shell* section for more information.

Controller

- The following Controller properties are now deprecated:
  - layout
  - view - replaced with `template`
  - theme
  - autoLayout
  - viewPath - replaced with `templatePath`
  - viewClass - replaced with `className`
  - layoutPath

  Instead of setting these properties on your controllers, you should set them on the view using methods with matching names:

```
// In a controller, instead of
$this->layout = 'advanced';

// You should use
$this->viewBuilder()->layout('advanced');
```

These methods should be called after you've determined which view class will be used by a controller/action.

AuthComponent

- New config option `storage` has been added. It contains the storage class name that `AuthComponent` uses to store user record. By default `SessionStorage` is used. If using a stateless authenticator you should configure `AuthComponent` to use `MemoryStorage` instead.

- New config option `checkAuthIn` has been added. It contains the name of the event for which auth checks should be done. By default `Controller.startup` is used, but you can set it to `Controller.initialize` if you want authentication to be checked before you controller's `beforeFilter()` method is run.
- The options `scope` and `contain` for authenticator classes have been deprecated. Instead, use the new `finder` option to configure a custom finder method and modify the query used to find a user there.
- The logic for setting `Auth.redirect` session variable, which is used to get the URL to be redirected to after login, has been changed. It is now set only when trying to access a protected URL without authentication. So `Auth::redirectUrl()` returns the protected URL after login. Under normal circumstances, when a user directly accesses the login page, `Auth::redirectUrl()` returns the value set for `loginRedirect` config.

FlashComponent

- `FlashComponent` now stacks Flash messages when set with the `set()` or `__call()` method. This means that the structure in the Session for stored Flash messages has changed.

CsrfComponent

- CSRF cookie expiry time can now be set as a `strtotime()` compatible value.
- Invalid CSRF tokens will now throw a `Cake\Network\Exception\InvalidCsrfTokenException` instead of the `Cake\Network\Exception\ForbiddenException`.

RequestHandlerComponent

- `RequestHandlerComponent` now switches the layout and template based on the parsed extension or `Accept` header in the `beforeRender()` callback instead of `startup()`.
- `addInputType()` and `viewClassMap()` are deprecated. You should use `config()` to modify this configuration data at runtime.
- When `inputTypeMap` or `viewClassMap` are defined in the component settings, they will *overwrite* the default values. This change makes it possible to remove the default configuration.

Network

HttpClient

- The default mime type used when sending requests has changed. Previously `multipart/form-data` would always be used. In 3.1, `multipart/form-data` is only used when file uploads are present. When there are no file uploads, `application/x-www-form-urlencoded` is used instead.

ORM

You can now *Lazily Eager Load Associations*. This feature allows you to conditionally load additional associations into a result set, entity or collection of entities.

The `patchEntity()` and `newEntity()` method now support the `onlyIds` option. This option allows you to restrict hasMany/belongsToMany association marshalling to only use the `_ids` list. This option defaults to `false`.

Query

- `Query::notMatching()` was added.
- `Query::leftJoinWith()` was added.
- `Query::innerJoinWith()` was added.
- `Query::select()` now supports `Table` and `Association` objects as parameters. These parameter types will select all the columns on the provided table or association instance's target table.
- `Query::distinct()` now accepts a string to distinct on a single column.
- `Table::loadInto()` was added.
- `EXTRACT`, `DATE_ADD` and `DAYOFWEEK` raw SQL functions have been abstracted to `extract()`, `dateAdd()` and `dayOfWeek()`.

View

- You can now set `_serialized` to `true` for `JsonView` and `XmlView` to serialize all view variables instead of explicitly specifying them.
- `View::$viewPath` is deprecated. You should use `View::templatePath()` instead.
- `View::$view` is deprecated. You should use `View::template()` instead.
- `View::TYPE_VIEW` is deprecated. You should use `View::TYPE_TEMPLATE` instead.

Helper

SessionHelper

- The `SessionHelper` has been deprecated. You can use `$this->request->session()` directly.

FlashHelper

- `FlashHelper` can render multiple messages if multiple messages where set with the `FlashComponent`. Each message will be rendered in its own element. Messages will be rendered in the order they were set.

FormHelper

- New option `templateVars` has been added. `templateVars` allows you to pass additional variables to your custom form control templates.

Email

- `Email` and `Transport` classes have been moved under the `Cake\Mailer` namespace. Their former namespaces are still usable as class aliases have been set for them.
- The `default` email profile is now automatically set when an `Email` instance is created. This behavior is similar to what is done in 2.x.

Mailer

- The `Mailer` class was added. This class helps create reusable emails in an application.

I18n

Time

- `Time::fromNow()` has been added. This method makes it easier to calculate differences from 'now'.
- `Time::i18nFormat()` now supports non-gregorian calendars when formatting dates.

Validation

- `Validation::geoCoordinate()` was added.
- `Validation::latitude()` was added.
- `Validation::longitude()` was added.
- `Validation::isInteger()` was added.
- `Validation::ascii()` was added.
- `Validation::utf8()` was added.

Testing

TestFixture

`model` key is now supported to retrieve the table name for importing.

3.0 Migration Guide

New ORM Upgrade Guide

CakePHP 3.0 features a new ORM that has been re-written from the ground up. While the ORM used in 1.x and 2.x has served us well for a long time it had a few issues that we wanted to fix.

- Frankenstein - Is it a record, or a table? Currently it's both.
- Inconsistent API - Model::read() for example.
- No query object - Queries are always defined as arrays, this has some limitations and restrictions. For example it makes doing unions and sub-queries much harder.
- Returns arrays - This is a common complaint about CakePHP, and has probably reduced adoption at some levels.
- No record object - This makes attaching formatting methods difficult/impossible.
- Containable - Should be part of the ORM, not a crazy hacky behavior.
- Recursive - This should be better controlled as defining which associations are included, not a level of recursiveness.

- DboSource - It is a beast, and Model relies on it more than datasource. That separation could be cleaner and simpler.
- Validation - Should be separate, it's a giant crazy function right now. Making it a reusable bit would make the framework more extensible.

The ORM in CakePHP 3.0 solves these and many more problems. The new ORM focuses on relational data stores right now. In the future and through plugins we will add non relational stores like ElasticSearch and others.

Design of the New ORM

The new ORM solves several problems by having more specialized and focused classes. In the past you would use `Model` and a Datasource for all operations. Now the ORM is split into more layers:

- `Cake\Database\Connection` - Provides a platform independent way to create and use connections. This class provides a way to use transactions, execute queries and access schema data.
- `Cake\Database\Dialect` - The classes in this namespace provide platform specific SQL and transform queries to work around platform specific limitations.
- `Cake\Database\Type` - Is the gateway class to CakePHP database type conversion system. It is a pluggable framework for adding abstract column types and providing mappings between database, PHP representations and PDO bindings for each data type. For example datetime columns are represented as `DateTime` instances in your code now.
- `Cake\ORM\Table` - The main entry point into the new ORM. Provides access to a single table. Handles the definition of association, use of behaviors and creation of entities and query objects.
- `Cake\ORM\Behavior` - The base class for behaviors, which act very similar to behaviors in previous versions of CakePHP.
- `Cake\ORM\Query` - A fluent object based query builder that replaces the deeply nested arrays used in previous versions of CakePHP.
- `Cake\ORM\ResultSet` - A collection of results that gives powerful tools for manipulating data in aggregate.
- `Cake\ORM\Entity` - Represents a single row result. Makes accessing data and serializing to various formats a snap.

Now that you are more familiar with some of the classes you'll interact with most frequently in the new ORM it is good to look at the three most important classes. The `Table`, `Query` and `Entity` classes do much of the heavy lifting in the new ORM, and each serves a different purpose.

Table Objects

Table objects are the gateway into your data. They handle many of the tasks that `Model` did in previous releases. Table classes handle tasks like:

- Creating queries.
- Providing finders.
- Validating and saving entities.
- Deleting entities.
- Defining and accessing associations.
- Triggering callback events.
- Interacting with behaviors.

The documentation chapter on *Table Objects* provides far more detail on how to use table objects than this guide can. Generally when moving existing model code over it will end up in a table object. Table objects don't contain any platform dependent SQL. Instead they collaborate with entities and the query builder to do their work. Table objects also interact with behaviors and other interested parties through published events.

Query Objects

While these are not classes you will build yourself, your application code will make extensive use of the *Query Builder* which is central to the new ORM. The query builder makes it easy to build simple or complex queries including those that were previously very difficult in CakePHP like `HAVING`, `UNION` and sub-queries.

The various find() calls your application has currently will need to be updated to use the new query builder. The Query object is responsible for containing the data to make a query without executing the query itself. It collaborates with the connection/dialect to generate platform specific SQL which is executed creating a `ResultSet` as the output.

Entity Objects

In previous versions of CakePHP the `Model` class returned dumb arrays that could not contain any logic or behavior. While the community made this short-coming less painful with projects like CakeEntity, the array results were often a short coming that caused many developers trouble. For CakePHP 3.0, the ORM always returns object result sets unless you explicitly disable that feature. The chapter on *Entities* covers the various tasks you can accomplish with entities.

Entities are created in one of two ways. Either by loading data from the database, or converting request data into entities. Once created, entities allow you to manipulate the data they contain and persist their data by collaborating with table objects.

Key Differences

The new ORM is a large departure from the existing `Model` layer. There are many important differences that are important in understanding how the new ORM operates and how to update your code.

Inflection Rules Updated

You may have noticed that table classes have a pluralized name. In addition to tables having pluralized names, associations are also referred in the plural form. This is in contrast to `Model` where class names and association aliases were singular. There are a few reasons for this change:

- Table classes represent **collections** of data, not single rows.
- Associations link tables together, describing the relations between many things.

While the conventions for table objects are to always use plural forms, your entity association properties will be populated based on the association type.

Note: BelongsTo and HasOne associations will use the singular form in entity properties, while HasMany and BelongsToMany (HABTM) will use plural forms.

The convention change for table objects is most apparent when building queries. Instead of expressing queries like:

```
// Wrong
$query->where(['User.active' => 1]);
```

You need to use the plural form:

```
// Correct
$query->where(['Users.active' => 1]);
```

Find returns a Query Object

One important difference in the new ORM is that calling `find` on a table will not return the results immediately, but will return a Query object; this serves several purposes.

It is possible to alter queries further, after calling `find`:

```
$articles = TableRegistry::get('Articles');
$query = $articles->find();
$query->where(['author_id' => 1])->order(['title' => 'DESC']);
```

It is possible to stack custom finders to append conditions, sorting, limit and any other clause to the same query before it is executed:

```
$query = $articles->find('approved')->find('popular');
$query->find('latest');
```

You can compose queries one into the other to create subqueries easier than ever:

```
$query = $articles->find('approved');
$favoritesQuery = $article->find('favorites', ['for' => $user]);
$query->where(['id' => $favoritesQuery->select(['id'])]);
```

You can decorate queries with iterators and call methods without even touching the database. This is great when you have parts of your view cached and having the results taken from the database is not actually required:

```
// No queries made in this example!
$results = $articles->find()
    ->order(['title' => 'DESC'])
    ->formatResults(function (\Cake\Collection\CollectionInterface $results) {
        return $results->extract('title');
    });
```

Queries can be seen as the result object, trying to iterate the query, calling `toArray()` or any method inherited from *collection*, will result in the query being executed and results returned to you.

The biggest difference you will find when coming from CakePHP 2.x is that `find('first')` does not exist anymore. There is a trivial replacement for it, and it is the `first()` method:

```
// Before
$article = $this->Article->find('first');

// Now
$article = $this->Articles->find()->first();

// Before
$article = $this->Article->find('first', [
    'conditions' => ['author_id' => 1]
```

```
]);

// Now
$article = $this->Articles->find('all', [
    'conditions' => ['author_id' => 1]
])->first();

// Can also be written
$article = $this->Articles->find()
    ->where(['author_id' => 1])
    ->first();
```

If you are loading a single record by its primary key, it will be better to just call `get()`:

```
$article = $this->Articles->get(10);
```

Finder Method Changes

Returning a query object from a find method has several advantages, but comes at a cost for people migrating from 2.x. If you had some custom find methods in your models, they will need some modifications. This is how you create custom finder methods in 3.0:

```
class ArticlesTable
{

    public function findPopular(Query $query, array $options)
    {
        return $query->where(['times_viewed' > 1000]);
    }

    public function findFavorites(Query $query, array $options)
    {
        $for = $options['for'];
        return $query->matching('Users.Favorites', function ($q) use ($for) {
            return $q->where(['Favorites.user_id' => $for]);
        });
    }
}
```

As you can see, they are pretty straightforward, they get a Query object instead of an array and must return a Query object back. For 2.x users that implemented afterFind logic in custom finders, you should check out the *Modifying Results with Map/Reduce* section, or use the features found on the *collection objects*. If in your models you used to rely on having an afterFind for all find operations you can migrate this code in one of a few ways:

1. Override your entity constructor method and do additional formatting there.
2. Create accessor methods in your entity to create the virtual properties.
3. Redefine `findAll()` and use `formatResults`.

In the 3rd case above your code would look like:

```
public function findAll(Query $query, array $options)
{
    return $query->formatResults(function (\Cake\Collection\CollectionInterface
↪$results) {
        return $results->map(function ($row) {
```

```
            // Your afterfind logic
        });
    })
}
```

You may have noticed that custom finders receive an options array. You can pass any extra information to your finder using this parameter. This is great news for people migrating from 2.x. Any of the query keys that were used in previous versions will be converted automatically for you in 3.x to the correct functions:

```
// This works in both CakePHP 2.x and 3.0
$articles = $this->Articles->find('all', [
    'fields' => ['id', 'title'],
    'conditions' => [
        'OR' => ['title' => 'Cake', 'author_id' => 1],
        'published' => true
    ],
    'contain' => ['Authors'], // The only change! (notice plural)
    'order' => ['title' => 'DESC'],
    'limit' => 10,
]);
```

If your application uses 'magic' or *Dynamic Finders*, you will have to adapt those calls. In 3.x the `findAllBy*` methods have been removed, instead `findBy*` always returns a query object. To get the first result, you need to use the `first()` method:

```
$article = $this->Articles->findByTitle('A great post!')->first();
```

Hopefully, migrating from older versions is not as daunting as it first seems. Many of the features we have added will help you remove code as you can better express your requirements using the new ORM and at the same time the compatibility wrappers will help you rewrite those tiny differences in a fast and painless way.

One of the other nice improvements in 3.x around finder methods is that behaviors can implement finder methods with no fuss. By simply defining a method with a matching name and signature on a Behavior the finder will automatically be available on any tables the behavior is attached to.

Recursive and ContainableBehavior Removed

In previous versions of CakePHP you needed to use `recursive`, `bindModel()`, `unbindModel()` and `ContainableBehavior` to reduce the loaded data to the set of associations you were interested in. A common tactic to manage associations was to set `recursive` to `-1` and use Containable to manage all associations. In CakePHP 3.0 ContainableBehavior, recursive, bindModel, and unbindModel have all been removed. Instead the `contain()` method has been promoted to be a core feature of the query builder. Associations are only loaded if they are explicitly turned on. For example:

```
$query = $this->Articles->find('all');
```

Will **only** load data from the `articles` table as no associations have been included. To load articles and their related authors you would do:

```
$query = $this->Articles->find('all')->contain(['Authors']);
```

By only loading associated data that has been specifically requested you spend less time fighting the ORM trying to get only the data you want.

No afterFind Event or Virtual Fields

In previous versions of CakePHP you needed to make extensive use of the `afterFind` callback and virtual fields in order to create generated data properties. These features have been removed in 3.0. Because of how ResultSets iteratively generate entities, the `afterFind` callback was not possible. Both afterFind and virtual fields can largely be replaced with virtual properties on entities. For example if your User entity has both first and last name columns you can add an accessor for *full_name* and generate the property on the fly:

```
namespace App\Model\Entity;

use Cake\ORM\Entity;

class User extends Entity
{
    protected function _getFullName()
    {
        return $this->first_name . '  ' . $this->last_name;
    }
}
```

Once defined you can access your new property using `$user->full_name`. Using the *Modifying Results with Map/Reduce* features of the ORM allow you to build aggregated data from your results, which is another use case that the `afterFind` callback was often used for.

While virtual fields are no longer an explicit feature of the ORM, adding calculated fields is easy to do in your finder methods. By using the query builder and expression objects you can achieve the same results that virtual fields gave:

```
namespace App\Model\Table;

use Cake\ORM\Table;
use Cake\ORM\Query;

class ReviewsTable extends Table
{
    public function findAverage(Query $query, array $options = [])
    {
        $avg = $query->func()->avg('rating');
        $query->select(['average' => $avg]);
        return $query;
    }
}
```

Associations No Longer Defined as Properties

In previous versions of CakePHP the various associations your models had were defined in properties like `$belongsTo` and `$hasMany`. In CakePHP 3.0, associations are created with methods. Using methods allows us to sidestep the many limitations class definitions have, and provide only one way to define associations. Your `initialize()` method and all other parts of your application code, interact with the same API when manipulating associations:

```
namespace App\Model\Table;

use Cake\ORM\Table;
use Cake\ORM\Query;

class ReviewsTable extends Table
```

```
{

    public function initialize(array $config)
    {
        $this->belongsTo('Movies');
        $this->hasOne('Ratings');
        $this->hasMany('Comments')
        $this->belongsToMany('Tags')
    }

}
```

As you can see from the example above each of the association types uses a method to create the association. One other difference is that `hasAndBelongsToMany` has been renamed to `belongsToMany`. To find out more about creating associations in 3.0 see the section on *Associations - Linking Tables Together*.

Another welcome improvement to CakePHP is the ability to create your own association classes. If you have association types that are not covered by the built-in relation types you can create a custom `Association` sub-class and define the association logic you need.

Validation No Longer Defined as a Property

Like associations, validation rules were defined as a class property in previous versions of CakePHP. This array would then be lazily transformed into a `ModelValidator` object. This transformation step added a layer of indirection, complicating rule changes at runtime. Furthermore, validation rules being defined as a property made it difficult for a model to have multiple sets of validation rules. In CakePHP 3.0, both these problems have been remedied. Validation rules are always built with a `Validator` object, and it is trivial to have multiple sets of rules:

```
namespace App\Model\Table;

use Cake\ORM\Table;
use Cake\ORM\Query;
use Cake\Validation\Validator;

class ReviewsTable extends Table
{

    public function validationDefault(Validator $validator)
    {
        $validator->requirePresence('body')
            ->add('body', 'length', [
                'rule' => ['minLength', 20],
                'message' => 'Reviews must be 20 characters or more',
            ])
            ->add('user_id', 'numeric', [
                'rule' => 'numeric'
            ]);
        return $validator;
    }

}
```

You can define as many validation methods as you need. Each method should be prefixed with `validation` and accept a `$validator` argument.

In previous versions of CakePHP 'validation' and the related callbacks covered a few related but different uses. In CakePHP 3.0, what was formerly called validation is now split into two concepts:

1. Data type and format validation.
2. Enforcing application, or business rules.

Validation is now applied before ORM entities are created from request data. This step lets you ensure data matches the data type, format, and basic shape your application expects. You can use your validators when converting request data into entities by using the `validate` option. See the documentation on *Converting Request Data into Entities* for more information.

Application rules allow you to define rules that ensure your application's rules, state and workflows are enforced. Rules are defined in your Table's `buildRules()` method. Behaviors can add rules using the `buildRules()` hook method. An example `buildRules()` method for our articles table could be:

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

use Cake\ORM\Table;
use Cake\ORM\RulesChecker;

class ArticlesTable extends Table
{
    public function buildRules(RulesChecker $rules)
    {
        $rules->add($rules->existsIn('user_id', 'Users'));
        $rules->add(
            function ($article, $options) {
                return ($article->published && empty($article->reviewer));
            },
            'isReviewed',
            [
                'errorField' => 'published',
                'message' => 'Articles must be reviewed before publishing.'
            ]
        );
        return $rules;
    }
}
```

Identifier Quoting Disabled by Default

In the past CakePHP has always quoted identifiers. Parsing SQL snippets and attempting to quote identifiers was both error prone and expensive. If you are following the conventions CakePHP sets out, the cost of identifier quoting far outweighs any benefit it provides. Because of this identifier quoting has been disabled by default in 3.0. You should only need to enable identifier quoting if you are using column names or table names that contain special characters or are reserved words. If required, you can enable identifier quoting when configuring a connection:

```
// In config/app.php
'Datasources' => [
    'default' => [
        'className' => 'Cake\Database\Driver\Mysql',
        'username' => 'root',
        'password' => 'super_secret',
        'host' => 'localhost',
        'database' => 'cakephp',
```

```
        'quoteIdentifiers' => true,
    ]
],
```

Note: Identifiers in `QueryExpression` objects will not be quoted, and you will need to quote them manually or use IdentifierExpression objects.

Updating Behaviors

Like most ORM related features, behaviors have changed in 3.0 as well. They now attach to `Table` instances which are the conceptual descendant of the `Model` class in previous versions of CakePHP. There are a few key differences from behaviors in CakePHP 2.x:

- Behaviors are no longer shared across multiple tables. This means you no longer have to 'namespace' settings stored in a behavior. Each table using a behavior will get its own instance.
- The method signatures for mixin methods have changed.
- The method signatures for callback methods have changed.
- The base class for behaviors have changed.
- Behaviors can add finder methods.

New Base Class

The base class for behaviors has changed. Behaviors should now extend `Cake\ORM\Behavior`; if a behavior does not extend this class an exception will be raised. In addition to the base class changing, the constructor for behaviors has been modified, and the `startup()` method has been removed. Behaviors that need access to the table they are attached to should define a constructor:

```
namespace App\Model\Behavior;

use Cake\ORM\Behavior;

class SluggableBehavior extends Behavior
{

    protected $_table;

    public function __construct(Table $table, array $config)
    {
        parent::__construct($table, $config);
        $this->_table = $table;
    }

}
```

Mixin Methods Signature Changes

Behaviors continue to offer the ability to add 'mixin' methods to Table objects, however the method signature for these methods has changed. In CakePHP 3.0, behavior mixin methods can expect the **same** arguments provided to the table

'method'. For example:

```
// Assume table has a slug() method provided by a behavior.
$table->slug($someValue);
```

The behavior providing the `slug()` method will receive only 1 argument, and its method signature should look like:

```
public function slug($value)
{
    // Code here.
}
```

Callback Method Signature Changes

Behavior callbacks have been unified with all other listener methods. Instead of their previous arguments, they need to expect an event object as their first argument:

```
public function beforeFind(Event $event, Query $query, array $options)
{
    // Code.
}
```

See *Lifecycle Callbacks* for the signatures of all the callbacks a behavior can subscribe to.

General Information

CakePHP Development Process

Here we attempt to explain the process we use when developing the CakePHP framework. We rely heavily on community interaction through tickets and IRC chat. IRC is the best place to find members of the development team[194] and discuss ideas, the latest code, and make general comments. If something more formal needs to be proposed or there is a problem with a release, the ticket system is the best place to share your thoughts.

We currently maintain 4 versions of CakePHP.

- **tagged release** : Tagged releases intended for production where stability is more important than features. Issues filed against these releases will be fixed in the related branch, and be part of the next release.
- **mainline branch** : These branches are where all bugfixes are merged into. Stable releases are tagged from these branches. `master` is the mainline branch for the current release series. `2.x` is the maintenance branch for the 2.x release series. If you are using a stable release and need fixes that haven't made their way into a tagged release check here.
- **development branches** : Development branches contain leading edge fixes and features. They are named after the major version they are for. E.g *3.next*. Once development branches have reached a stable release point they are merged into the mainline branch.
- **feature branches** : Feature branches contain unfinished or possibly unstable features and are recommended only for power users interested in the most advanced feature set and willing to contribute back to the community. Feature branches are named with the following convention *version-feature*. An example would be *3.3-router* Which would contain new features for the Router for 3.3.

Hopefully this will help you understand what version is right for you. Once you pick your version you may feel compelled to contribute a bug report or make general comments on the code.

[194] https://github.com/cakephp?tab=members

- If you are using a stable version or maintenance branch, please submit tickets or discuss with us on IRC.
- If you are using the development branch or feature branch, the first place to go is IRC. If you have a comment and cannot reach us in IRC after a day or two, please submit a ticket.

If you find an issue, the best answer is to write a test. The best advice we can offer in writing tests is to look at the ones included in the core.

As always, if you have any questions or comments, visit us at #cakephp on irc.freenode.net.

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.

DSN Data Source Name. A connection string format that is formed like a URI. CakePHP supports DSN's 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

Cake\Cache, 533
Cake\Collection, 713
Cake\Console, 543
Cake\Console\Exception, 597
Cake\Controller, 209
Cake\Controller\Component, 243
Cake\Controller\Exception, 596
Cake\Core, 157
Cake\Core\Exception, 597
Cake\Database, 375
Cake\Database\Exception, 597
Cake\Database\Schema, 516
Cake\Datasource, 374
Cake\Datasource\Exception, 597
Cake\Error, 576
Cake\Filesystem, 733
Cake\Form, 633
Cake\Http, 757
Cake\Http\Client, 763
Cake\Http\Cookie, 206
Cake\I18n, 787
Cake\Log, 630
Cake\Mailer, 583
Cake\Network\Exception, 595
Cake\ORM, 414
Cake\ORM\Behavior, 507
Cake\ORM\Exception, 597
Cake\Routing, 163
Cake\Routing\Exception, 597
Cake\Utility, 795
Cake\Validation, 699
Cake\View, 259
Cake\View\Exception, 596
Cake\View\Helper, 361

Index

Symbols

() (method), **257**
:action, 164
:controller, 164
:plugin, 164
$this->request, 191
$this->response, 198
__() (global function), **799**
__d() (global function), **799**
__dn() (global function), **799**
__dx() (global function), **799**
__dxn() (global function), **799**
__n() (global function), **800**
__x() (global function), **800**
__xn() (global function), **800**

A

acceptLanguage() (Cake\Http\ServerRequest method), **197**
accepts() (Cake\Http\ServerRequest method), **197**
accepts() (RequestHandlerComponent method), **248**
addArgument() (Cake\Console\ConsoleOptionParser method), **557**
addArguments() (Cake\Console\ConsoleOptionParser method), **557**
addBehavior() (Cake\ORM\Table method), **419**
addCrumb() (Cake\View\Helper\HtmlHelper method), **338**
addDetector() (Cake\Http\ServerRequest method), **194**
addOption() (Cake\Console\ConsoleOptionParser method), **558**
addOptions() (Cake\Console\ConsoleOptionParser method), **558**
addPathElement() (Cake\Filesystem\Folder method), **734**
addSubcommand() (Cake\Console\ConsoleOptionParser method), **559**
admin routing, 171
afterDelete() (Cake\ORM\Table method), **419**
afterDeleteCommit() (Cake\ORM\Table method), **419**
afterFilter() (Cake\Controller\Controller method), **217**
afterLayout() (Helper method), **368**
afterRender() (Helper method), **368**
afterRenderFile() (Helper method), **368**
afterRules() (Cake\ORM\Table method), **418**
afterSave() (Cake\ORM\Table method), **418**
afterSaveCommit() (Cake\ORM\Table method), **419**
alert() (Cake\Log\Log method), **631**
allControls() (Cake\View\Helper\FormHelper method), **322**
allow() (AuthComponent method), **232**
allowMethod() (Cake\Http\ServerRequest method), **196**
App (class in Cake\Core), **709**
APP (global constant), **801**
app.php, 153
app.php.default, 153
APP_DIR (global constant), **801**
append() (Cake\Collection\Collection method), **727**
append() (Cake\Filesystem\File method), **737**
application exceptions, 598
apply() (Cake\Utility\Hash method), **751**
attachments() (Cake\Mailer\Email method), **588**
AuthComponent (class), **217**
autoLink() (Cake\View\Helper\TextHelper method), **356**
autoLinkEmails() (Cake\View\Helper\TextHelper method), **356**
autoLinkUrls() (Cake\View\Helper\TextHelper method), **356**
autoParagraph() (Cake\View\Helper\TextHelper method), **356**
avg() (Cake\Collection\Collection method), **720**

B

BadRequestException, **595**
beforeDelete() (Cake\ORM\Table method), **419**
beforeFilter() (Cake\Controller\Controller method), **216**

beforeFind() (Cake\ORM\Table method), 417
beforeLayout() (Helper method), 368
beforeMarshal() (Cake\ORM\Table method), 417
beforeRender() (Cake\Controller\Controller method), 217
beforeRender() (Helper method), 368
beforeRenderFile() (Helper method), 368
beforeRules() (Cake\ORM\Table method), 418
beforeSave() (Cake\ORM\Table method), 418
blackHole() (SecurityComponent method), 240
BreadcrumbsHelper (class in Cake\View\Helper), 280
breakpoint() (global function), 575
buffered() (Cake\Collection\Collection method), 730
build() (Cake\Utility\Xml method), 795
build() (Cake\View\Helper\UrlHelper method), 361
buildFromArray() (Cake\Console\ConsoleOptionParser method), 560
buildRules() (Cake\ORM\Table method), 418
buildValidator() (Cake\ORM\Table method), 418
button() (Cake\View\Helper\FormHelper method), 316

C

Cache (class in Cake\Cache), 533
CACHE (global constant), 801
cache() (Cake\View\View method), 269
CacheEngine (class in Cake\Cache), 540
CAKE (global constant), 801
CAKE_CORE_INCLUDE_PATH (global constant), 801
Cake\Cache (namespace), 533
Cake\Collection (namespace), 713
Cake\Console (namespace), 543
Cake\Console\Exception (namespace), 597
Cake\Controller (namespace), 209
Cake\Controller\Component (namespace), 234, 238, 243
Cake\Controller\Exception (namespace), 596
Cake\Core (namespace), 157, 709
Cake\Core\Exception (namespace), 597
Cake\Database (namespace), 375
Cake\Database\Exception (namespace), 597
Cake\Database\Schema (namespace), 516
Cake\Datasource (namespace), 374
Cake\Datasource\Exception (namespace), 597
Cake\Error (namespace), 576
Cake\Filesystem (namespace), 733
Cake\Form (namespace), 633
Cake\Http (namespace), 191, 757
Cake\Http\Client (namespace), 763
Cake\Http\Cookie (namespace), 206
Cake\I18n (namespace), 771, 787
Cake\Log (namespace), 630
Cake\Mailer (namespace), 583
Cake\Network\Exception (namespace), 595
Cake\ORM (namespace), 385, 414, 422, 430, 461, 482
Cake\ORM\Behavior (namespace), 496, 497, 499, 507
Cake\ORM\Exception (namespace), 597
Cake\Routing (namespace), 163
Cake\Routing\Exception (namespace), 597
Cake\Utility (namespace), 653, 741, 767, 779, 795
Cake\Validation (namespace), 699
Cake\View (namespace), 259
Cake\View\Exception (namespace), 596
Cake\View\Helper (namespace), 280, 283, 284, 326, 339, 343, 352, 355, 356, 360, 361
camelize() (Cake\Utility\Inflector method), 768
cd() (Cake\Filesystem\Folder method), 734
CDN, 864
charset() (Cake\View\Helper\HtmlHelper method), 327
check() (Cake\Controller\Component\CookieComponent method), 236
check() (Cake\Core\Configure method), 158
check() (Cake\Utility\Hash method), 747
check() (Cake\View\Helper\SessionHelper method), 355
check() (Session method), 663
checkbox() (Cake\View\Helper\FormHelper method), 298
checkNotModified() (Cake\Http\Response method), 205
chmod() (Cake\Filesystem\Folder method), 734
chunk() (Cake\Collection\Collection method), 717
chunkWithKeys() (Cake\Collection\Collection method), 717
classify() (Cake\Utility\Inflector method), 768
classname() (Cake\Core\App method), 709
cleanInsert() (Cake\Utility\Text method), 781
clear() (Cake\Cache\Cache method), 538
clear() (Cake\Cache\CacheEngine method), 541
clear() (Cake\ORM\TableRegistry method), 421
clearGroup() (Cake\Cache\Cache method), 539
clearGroup() (Cake\Cache\CacheEngine method), 541
Client (class in Cake\Http), 757
clientIp() (Cake\Http\ServerRequest method), 197
close() (Cake\Filesystem\File method), 737
Collection (class in Cake\Collection), 713
Collection (class in Cake\Database\Schema), 520
collection() (global function), 800
columns, 864
combine() (Cake\Collection\Collection method), 715
combine() (Cake\Utility\Hash method), 743
compile() (Cake\Collection\Collection method), 731
components (Cake\Controller\Controller property), 216
CONFIG (global constant), 801
config() (Cake\Cache\Cache method), 534
config() (Cake\Core\Configure method), 158
config() (Cake\I18n\Number method), 775
config() (Cake\Log\Log method), 630
configTransport() (Cake\Mailer\Email method), 584
configuration, 153
Configure (class in Cake\Core), 157
configured() (Cake\Log\Log method), 630
ConflictException, 595

connect() (Cake\Routing\Router method), 165
Connection (class in Cake\Database), 380
ConnectionManager (class in Cake\Datasource), 375
ConsoleException, 597
ConsoleOptionParser (class in Cake\Console), 553
consume() (Cake\Core\Configure method), 158
consume() (Session method), 663
contains() (Cake\Collection\Collection method), 725
contains() (Cake\Utility\Hash method), 747
control() (Cake\View\Helper\FormHelper method), 288
Controller (class in Cake\Controller), 209
controls() (Cake\View\Helper\FormHelper method), 321
Cookie (class in Cake\Http\Cookie), 206
CookieCollection (class in Cake\Http\Cookie), 206
CookieComponent (class in Cake\Controller\Component), 234
copy() (Cake\Filesystem\File method), 738
copy() (Cake\Filesystem\Folder method), 734
core() (Cake\Core\App method), 710
CORE_PATH (global constant), 801
correctSlashFor() (Cake\Filesystem\Folder method), 735
countBy() (Cake\Collection\Collection method), 721
counter() (Cake\View\Helper\PaginatorHelper method), 349
CounterCacheBehavior (class in Cake\ORM\Behavior), 496
create() (Cake\Filesystem\File method), 738
create() (Cake\Filesystem\Folder method), 735
create() (Cake\View\Helper\FormHelper method), 284
createFile() (Cake\Console\Shell method), 549
critical() (Cake\Log\Log method), 631
CSRF, 864
css() (Cake\View\Helper\HtmlHelper method), 327
currency() (Cake\I18n\Number method), 771
currency() (Cake\View\Helper\NumberHelper method), 339
current() (Cake\View\Helper\PaginatorHelper method), 348

D

dasherize() (Cake\Utility\Inflector method), 768
date() (Cake\View\Helper\FormHelper method), 308
dateTime() (Cake\View\Helper\FormHelper method), 307
DAY (global constant), 802
day() (Cake\View\Helper\FormHelper method), 312
dd() (global function), 800
debug() (Cake\Log\Log method), 631
debug() (global function), 800
Debugger (class in Cake\Error), 576
decrement() (Cake\Cache\Cache method), 538
decrement() (Cake\Cache\CacheEngine method), 541
decrypt() (Cake\Utility\Security method), 653
defaultCurrency() (Cake\I18n\Number method), 772
defaultCurrency() (Cake\View\Helper\NumberHelper method), 340
defaultRouteClass() (Cake\Routing\Router method), 184
delete() (Cake\Cache\Cache method), 537
delete() (Cake\Cache\CacheEngine method), 541
delete() (Cake\Controller\Component\CookieComponent method), 236
delete() (Cake\Core\Configure method), 158
delete() (Cake\Filesystem\File method), 738
delete() (Cake\Filesystem\Folder method), 735
delete() (Cake\ORM\Table method), 482
delete() (Session method), 662
deleteAll() (Cake\ORM\Table method), 483
deleteMany() (Cake\Cache\Cache method), 537
deleteOrFail() (Cake\ORM\Table method), 484
deny() (AuthComponent method), 232
destroy() (Session method), 663
diff() (Cake\Utility\Hash method), 752
dimensions() (Cake\Utility\Hash method), 750
dirsize() (Cake\Filesystem\Folder method), 735
dirty() (Cake\ORM\Entity method), 425
disable() (Cake\Cache\Cache method), 540
dispatchShell() (Cake\Console\Shell method), 548
doc (role), 123
docType() (Cake\View\Helper\HtmlHelper method), 329
domain() (Cake\Http\ServerRequest method), 196
dot notation, 864
drop() (Cake\Cache\Cache method), 535
drop() (Cake\Core\Configure method), 159
drop() (Cake\Log\Log method), 630
dropTransport() (Cake\Mailer\Email method), 585
DRY, 864
DS (global constant), 801
DSN, 864
dump() (Cake\Core\Configure method), 159
dump() (Cake\Error\Debugger method), 576

E

each() (Cake\Collection\Collection method), 714
element() (Cake\View\View method), 267
Email (class in Cake\Mailer), 583
emailPattern() (Cake\Mailer\Email method), 590
emergency() (Cake\Log\Log method), 631
enable() (Cake\Cache\Cache method), 540
enabled() (Cake\Cache\Cache method), 540
encrypt() (Cake\Utility\Security method), 653
end() (Cake\View\Helper\FormHelper method), 317
Entity (class in Cake\ORM), 422
env() (Cake\Http\ServerRequest method), 193
env() (global function), 800
error() (Cake\Log\Log method), 631
error() (Cake\View\Helper\FormHelper method), 314
errors() (Cake\Filesystem\Folder method), 735
errors() (Cake\ORM\Entity method), 426

every() (Cake\Collection\Collection method), 718
Exception, 597
ExceptionRenderer (class in Cake\Core\Exception), 598
excerpt() (Cake\Error\Debugger method), 577
excerpt() (Cake\Utility\Text method), 785
excerpt() (Cake\View\Helper\TextHelper method), 359
executable() (Cake\Filesystem\File method), 738
execute() (Cake\Database\Connection method), 380
exists() (Cake\Filesystem\File method), 738
expand() (Cake\Utility\Hash method), 748
ext() (Cake\Filesystem\File method), 738
extensions() (Cake\Routing\Router method), 175
extract() (Cake\Collection\Collection method), 714
extract() (Cake\Utility\Hash method), 742

F

fallbacks() (Cake\Routing\Router method), 184
fields, 864
File (class in Cake\Filesystem), 737
file extensions, 175
file() (Cake\View\Helper\FormHelper method), 305
filter() (Cake\Collection\Collection method), 718
filter() (Cake\Utility\Hash method), 748
find() (Cake\Filesystem\Folder method), 735
find() (Cake\ORM\Table method), 431
findRecursive() (Cake\Filesystem\Folder method), 735
first() (Cake\Collection\Collection method), 727
first() (Cake\View\Helper\PaginatorHelper method), 348
firstMatch() (Cake\Collection\Collection method), 719
FlashComponent (class in Cake\Controller\Component), 238
FlashHelper (class in Cake\View\Helper), 283
flatten() (Cake\Utility\Hash method), 748
Folder (Cake\Filesystem\File property), 737
Folder (class in Cake\Filesystem), 733
Folder() (Cake\Filesystem\File method), 738
ForbiddenException, 595
Form (class in Cake\Form), 633
format() (Cake\I18n\Number method), 773
format() (Cake\Utility\Hash method), 746
format() (Cake\View\Helper\NumberHelper method), 341
formatDelta() (Cake\I18n\Number method), 775
formatDelta() (Cake\View\Helper\NumberHelper method), 343
FormHelper (class in Cake\View\Helper), 284
FrozenDate (class in Cake\I18n), 793
FrozenTime (class in Cake\I18n), 793

G

gc() (Cake\Cache\Cache method), 538
gc() (Cake\Cache\CacheEngine method), 541
generateUrl() (Cake\View\Helper\PaginatorHelper method), 350
get() (Cake\Datasource\ConnectionManager method), 375
get() (Cake\ORM\Entity method), 423
get() (Cake\ORM\Table method), 431
get() (Cake\ORM\TableRegistry method), 421
get() (Cake\Utility\Hash method), 742
getCrumbList() (Cake\View\Helper\HtmlHelper method), 338
getCrumbs() (Cake\View\Helper\HtmlHelper method), 338
getData() (Cake\Http\ServerRequest method), 192
getMethod() (Cake\Http\ServerRequest method), 196
getQuery() (Cake\Http\ServerRequest method), 192
getType() (Cake\Error\Debugger method), 577
GoneException, 595
greedy star, 164
group() (Cake\Filesystem\File method), 738
groupBy() (Cake\Collection\Collection method), 721
groupConfigs() (Cake\Cache\Cache method), 539

H

h() (global function), 800
handle (Cake\Filesystem\File property), 737
Hash (class in Cake\Utility), 741
hash() (Cake\Utility\Security method), 654
hasNext() (Cake\View\Helper\PaginatorHelper method), 348
hasPage() (Cake\View\Helper\PaginatorHelper method), 349
hasPrev() (Cake\View\Helper\PaginatorHelper method), 349
Helper (class), 368
helpers (Cake\Controller\Controller property), 216
hidden() (Cake\View\Helper\FormHelper method), 295
highlight() (Cake\Utility\Text method), 782
highlight() (Cake\View\Helper\TextHelper method), 357
host() (Cake\Http\ServerRequest method), 196
HOUR (global constant), 802
hour() (Cake\View\Helper\FormHelper method), 312
HTML attributes, 864
HtmlHelper (class in Cake\View\Helper), 326
humanize() (Cake\Utility\Inflector method), 768

I

i18nFormat() (Cake\I18n\Time method), 789
identify() (AuthComponent method), 221
image() (Cake\View\Helper\HtmlHelper method), 330
in() (Cake\Console\Shell method), 549
inCakePath() (Cake\Filesystem\Folder method), 736
increment() (Cake\Cache\Cache method), 538
increment() (Cake\Cache\CacheEngine method), 541
indexBy() (Cake\Collection\Collection method), 721
Inflector (class in Cake\Utility), 767
info (Cake\Filesystem\File property), 737

info() (Cake\Filesystem\File method), 738
info() (Cake\Log\Log method), 631
initialize() (Cake\Console\Shell method), 553
initialize() (Cake\ORM\Table method), 417
inPath() (Cake\Filesystem\Folder method), 736
input() (Cake\Http\ServerRequest method), 193
insert() (Cake\Collection\Collection method), 728
insert() (Cake\Utility\Hash method), 742
insert() (Cake\Utility\Text method), 781
InternalErrorException, 595
InvalidCsrfTokenException, 595
is() (Cake\Http\ServerRequest method), 194
isAbsolute() (Cake\Filesystem\Folder method), 736
isAtom() (RequestHandlerComponent method), 249
isEmpty() (Cake\Collection\Collection method), 725
isFieldError() (Cake\View\Helper\FormHelper method), 315
isMobile() (RequestHandlerComponent method), 249
isRss() (RequestHandlerComponent method), 249
isSlashTerm() (Cake\Filesystem\Folder method), 736
isThisMonth() (Cake\I18n\Time method), 792
isThisWeek() (Cake\I18n\Time method), 792
isThisYear() (Cake\I18n\Time method), 792
isWap() (RequestHandlerComponent method), 250
isWindowsPath() (Cake\Filesystem\Folder method), 736
isWithinNext() (Cake\I18n\Time method), 792
isXml() (RequestHandlerComponent method), 249
isYesterday() (Cake\I18n\Time method), 792

J

JsonView (class), 278

L

label() (Cake\View\Helper\FormHelper method), 313
last() (Cake\Collection\Collection method), 727
last() (Cake\View\Helper\PaginatorHelper method), 348
lastAccess() (Cake\Filesystem\File method), 738
lastChange() (Cake\Filesystem\File method), 738
levels() (Cake\Log\Log method), 630
limitControl() (Cake\View\Helper\PaginatorHelper method), 350
link() (Cake\View\Helper\HtmlHelper method), 330
listNested() (Cake\Collection\Collection method), 724
load() (Cake\Core\Configure method), 159
loadComponent() (Cake\Controller\Controller method), 215
loadModel() (Cake\Controller\Controller method), 214
lock (Cake\Filesystem\File property), 737
Log (class in Cake\Log), 630
log() (Cake\Error\Debugger method), 577
log() (Cake\Log\LogTrait method), 631
logout() (AuthComponent method), 229
LOGS (global constant), 801
LogTrait (trait in Cake\Log), 631

M

map() (Cake\Collection\Collection method), 714
map() (Cake\Database\Type method), 376
map() (Cake\Utility\Hash method), 751
match() (Cake\Collection\Collection method), 718
max() (Cake\Collection\Collection method), 720
maxDimensions() (Cake\Utility\Hash method), 750
md5() (Cake\Filesystem\File method), 738
media() (Cake\View\Helper\HtmlHelper method), 332
median() (Cake\Collection\Collection method), 720
merge() (Cake\Console\ConsoleOptionParser method), 561
merge() (Cake\Utility\Hash method), 749
mergeDiff() (Cake\Utility\Hash method), 752
meridian() (Cake\View\Helper\FormHelper method), 313
messages() (Cake\Filesystem\Folder method), 736
meta() (Cake\View\Helper\HtmlHelper method), 328
MethodNotAllowedException, 595
mime() (Cake\Filesystem\File method), 739
min() (Cake\Collection\Collection method), 719
MINUTE (global constant), 802
minute() (Cake\View\Helper\FormHelper method), 312
MissingActionException, 596
MissingBehaviorException, 597
MissingCellException, 596
MissingCellViewException, 596
MissingComponentException, 596
MissingConnectionException, 597
MissingControllerException, 597
MissingDispatcherFilterException, 597
MissingDriverException, 597
MissingElementException, 596
MissingEntityException, 597
MissingExtensionException, 597
MissingHelperException, 596
MissingLayoutException, 596
MissingRouteException, 597
MissingShellException, 597
MissingShellMethodException, 597
MissingTableException, 597
MissingTaskException, 597
MissingTemplateException, 596
MissingViewException, 596
mode (Cake\Filesystem\Folder property), 734
MONTH (global constant), 802
month() (Cake\View\Helper\FormHelper method), 311
move() (Cake\Filesystem\Folder method), 736

N

name (Cake\Filesystem\File property), 737
name() (Cake\Filesystem\File method), 738
namespaceSplit() (global function), 801
nest() (Cake\Collection\Collection method), 724
nest() (Cake\Utility\Hash method), 754

nestedList() (Cake\View\Helper\HtmlHelper method), 334
newQuery() (Cake\Database\Connection method), 380
next() (Cake\View\Helper\PaginatorHelper method), 348
nice() (Cake\I18n\Time method), 790
normalize() (Cake\Utility\Hash method), 753
normalizePath() (Cake\Filesystem\Folder method), 736
NotAcceptableException, 595
NotFoundException, 595
notice() (Cake\Log\Log method), 631
NotImplementedException, 595
Number (class in Cake\I18n), 771
NumberHelper (class in Cake\View\Helper), 339
numbers() (Cake\View\Helper\PaginatorHelper method), 346
numeric() (Cake\Utility\Hash method), 750

O

offset() (Cake\Filesystem\File method), 738
open() (Cake\Filesystem\File method), 738
options() (Cake\View\Helper\PaginatorHelper method), 350
ordinal() (Cake\I18n\Number method), 774
ordinal() (Cake\View\Helper\NumberHelper method), 342
owner() (Cake\Filesystem\File method), 738

P

PaaS, 865
paginate() (Cake\Controller\Controller method), 215
PaginatorComponent (class in Cake\Controller\Component), 243
PaginatorHelper (class in Cake\View\Helper), 343
parseFileSize() (Cake\Utility\Text method), 781
passed arguments, 181
password() (Cake\View\Helper\FormHelper method), 295
path (Cake\Filesystem\File property), 737
path (Cake\Filesystem\Folder property), 734
path() (Cake\Core\App method), 709
path() (Cake\Core\Plugin method), 710
perms() (Cake\Filesystem\File method), 738
PersistenceFailedException, 597
php:attr (directive), 124
php:attr (role), 125
php:class (directive), 124
php:class (role), 125
php:const (directive), 124
php:const (role), 125
php:exc (role), 125
php:exception (directive), 124
php:func (role), 125
php:function (directive), 123
php:global (directive), 123
php:global (role), 125
php:meth (role), 125
php:method (directive), 124
php:staticmethod (directive), 124
pj() (global function), 800
plugin routing, 173
plugin syntax, 865
plugin() (Cake\Routing\Router method), 173
pluginSplit() (global function), 800
pluralize() (Cake\Utility\Inflector method), 768
postButton() (Cake\View\Helper\FormHelper method), 318
postLink() (Cake\View\Helper\FormHelper method), 318
pr() (global function), 800
precision() (Cake\I18n\Number method), 772
precision() (Cake\View\Helper\NumberHelper method), 340
prefers() (RequestHandlerComponent method), 251
prefix routing, 171
prefix() (Cake\Routing\Router method), 172
prepare() (Cake\Filesystem\File method), 738
prev() (Cake\View\Helper\PaginatorHelper method), 347
PrivateActionException, 596
properties, 865
pwd() (Cake\Filesystem\File method), 738
pwd() (Cake\Filesystem\Folder method), 736

Q

Query (class in Cake\ORM), 385
query() (Cake\Database\Connection method), 380

R

radio() (Cake\View\Helper\FormHelper method), 299
randomBytes() (Cake\Utility\Security method), 655
read() (Cake\Cache\Cache method), 537
read() (Cake\Cache\CacheEngine method), 541
read() (Cake\Controller\Component\CookieComponent method), 236
read() (Cake\Core\Configure method), 157
read() (Cake\Filesystem\File method), 738
read() (Cake\Filesystem\Folder method), 736
read() (Cake\View\Helper\SessionHelper method), 355
read() (Session method), 662
readable() (Cake\Filesystem\File method), 738
readMany() (Cake\Cache\Cache method), 537
readOrFail() (Cake\Core\Configure method), 158
realpath() (Cake\Filesystem\Folder method), 737
RecordNotFoundException, 597
redirect() (Cake\Controller\Controller method), 213
redirectUrl() (AuthComponent method), 221
reduce() (Cake\Collection\Collection method), 719
reduce() (Cake\Utility\Hash method), 751
ref (role), 123
referer() (Cake\Http\ServerRequest method), 197
reject() (Cake\Collection\Collection method), 718

remember() (Cake\Cache\Cache method), 536
remove() (Cake\Utility\Hash method), 743
render() (Cake\Controller\Controller method), 212
renderAs() (RequestHandlerComponent method), 251
renew() (Session method), 663
replaceText() (Cake\Filesystem\File method), 739
RequestHandlerComponent (class), 248
requireAuth() (SecurityComponent method), 241
requireSecure() (SecurityComponent method), 241
respondAs() (RequestHandlerComponent method), 251
Response (class in Cake\Http), 198
Response (class in Cake\Http\Client), 763
responseHeader() (Cake\Core\Exception\Exception method), 597
responseType() (RequestHandlerComponent method), 251
restore() (Cake\Core\Configure method), 160
RFC
    RFC 2606, 138
    RFC 2616#section-10.4, 595
    RFC 2616#section-10.5, 596
    RFC 4122, 780
ROOT (global constant), 801
Router (class in Cake\Routing), 163
routes.php, 163, 865
routing array, 865
RssHelper (class in Cake\View\Helper), 352
rules() (Cake\Utility\Inflector method), 770

S

safe() (Cake\Filesystem\File method), 739
sample() (Cake\Collection\Collection method), 726
save() (Cake\ORM\Table method), 473
saveMany() (Cake\ORM\Table method), 481
saveOrFail() (Cake\ORM\Table method), 480
script() (Cake\View\Helper\HtmlHelper method), 333
scriptBlock() (Cake\View\Helper\HtmlHelper method), 334
scriptEnd() (Cake\View\Helper\HtmlHelper method), 334
scriptStart() (Cake\View\Helper\HtmlHelper method), 334
SECOND (global constant), 801
secure() (Cake\View\Helper\FormHelper method), 326
Security (class in Cake\Utility), 653
SecurityComponent (class), 239
select() (Cake\View\Helper\FormHelper method), 301
ServerRequest (class in Cake\Http), 191
ServiceUnavailableException, 596
Session (class), 662
SessionHelper (class in Cake\View\Helper), 355
set() (Cake\Controller\Controller method), 211
set() (Cake\ORM\Entity method), 424
set() (Cake\View\View method), 261
setAction() (Cake\Controller\Controller method), 214
setDescription() (Cake\Console\ConsoleOptionParser method), 554
setEpilog() (Cake\Console\ConsoleOptionParser method), 556
setHelpAlias() (Cake\Console\ConsoleOptionParser method), 555
setJsonEncodeFormat() (Cake\I18n\Time method), 789
setTemplates() (Cake\View\Helper\HtmlHelper method), 337
setTemplates() (Cake\View\Helper\PaginatorHelper method), 344
setUser() (AuthComponent method), 229
Shell (class in Cake\Console), 544
shuffle() (Cake\Collection\Collection method), 726
singularize() (Cake\Utility\Inflector method), 768
size() (Cake\Filesystem\File method), 739
skip() (Cake\Collection\Collection method), 727
slashTerm() (Cake\Filesystem\Folder method), 737
slug() (Cake\Utility\Inflector method), 769
slug() (Cake\Utility\Text method), 780
some() (Cake\Collection\Collection method), 718
sort (Cake\Filesystem\Folder property), 734
sort() (Cake\Utility\Hash method), 751
sort() (Cake\View\Helper\PaginatorHelper method), 345
sortBy() (Cake\Collection\Collection method), 723
sortDir() (Cake\View\Helper\PaginatorHelper method), 346
sortKey() (Cake\View\Helper\PaginatorHelper method), 346
stackTrace() (global function), 575
startup() (Cake\Console\Shell method), 553
stopWhen() (Cake\Collection\Collection method), 716
store() (Cake\Core\Configure method), 160
stripLinks() (Cake\Utility\Text method), 783
stripLinks() (Cake\View\Helper\TextHelper method), 357
style() (Cake\View\Helper\HtmlHelper method), 328
subdomains() (Cake\Http\ServerRequest method), 196
submit() (Cake\View\Helper\FormHelper method), 315
sumOf() (Cake\Collection\Collection method), 720

T

Table (class in Cake\ORM), 430
tableCells() (Cake\View\Helper\HtmlHelper method), 336
tableHeaders() (Cake\View\Helper\HtmlHelper method), 335
tableize() (Cake\Utility\Inflector method), 769
TableRegistry (class in Cake\ORM), 420
TableSchema (class in Cake\Database\Schema), 517
tail() (Cake\Utility\Text method), 784
tail() (Cake\View\Helper\TextHelper method), 358
take() (Cake\Collection\Collection method), 726
TESTS (global constant), 801
Text (class in Cake\Utility), 779

text() (Cake\View\Helper\FormHelper method), 294
textarea() (Cake\View\Helper\FormHelper method), 295
TextHelper (class in Cake\View\Helper), 356
through() (Cake\Collection\Collection method), 729
Time (class in Cake\I18n), 787
time() (Cake\View\Helper\FormHelper method), 309
TIME_START (global constant), 801
timeAgoInWords() (Cake\I18n\Time method), 791
TimeHelper (class in Cake\View\Helper), 360
TimestampBehavior (class in Cake\ORM\Behavior), 497
TMP (global constant), 801
tokenize() (Cake\Utility\Text method), 781
toList() (Cake\Utility\Text method), 785
toList() (Cake\View\Helper\TextHelper method), 359
toPercentage() (Cake\I18n\Number method), 773
toPercentage() (Cake\View\Helper\NumberHelper method), 340
toQuarter() (Cake\I18n\Time method), 792
toReadableSize() (Cake\I18n\Number method), 773
toReadableSize() (Cake\View\Helper\NumberHelper method), 341
total() (Cake\View\Helper\PaginatorHelper method), 349
trace() (Cake\Error\Debugger method), 577
trailing star, 164
transactional() (Cake\Database\Connection method), 381
TranslateBehavior (class in Cake\ORM\Behavior), 499
transliterate() (Cake\Utility\Text method), 779
transpose() (Cake\Collection\Collection method), 726
tree() (Cake\Filesystem\Folder method), 737
TreeBehavior (class in Cake\ORM\Behavior), 507
truncate() (Cake\Utility\Text method), 783
truncate() (Cake\View\Helper\TextHelper method), 357
Type (class in Cake\Database), 375

U

UnauthorizedException, 595
underscore() (Cake\Utility\Inflector method), 768
unfold() (Cake\Collection\Collection method), 716
unlockField() (Cake\View\Helper\FormHelper method), 326
updateAll() (Cake\ORM\Table method), 481
url() (Cake\Routing\Router method), 182
UrlHelper (class in Cake\View\Helper), 361
user() (AuthComponent method), 229
uuid() (Cake\Utility\Text method), 780

V

Validator (class in Cake\Validation), 699
variable() (Cake\Utility\Inflector method), 769
vendor/cakephp-plugins.php, 637
View (class in Cake\View), 259

W

warning() (Cake\Log\Log method), 631
wasWithinLast() (Cake\I18n\Time method), 792
WEEK (global constant), 802
withBody() (Cake\Http\Response method), 201
withCache() (Cake\Http\Response method), 202
withCharset() (Cake\Http\Response method), 201
withDisabledCache() (Cake\Http\Response method), 202
withEtag() (Cake\Http\Response method), 203
withExpires() (Cake\Http\Response method), 203
withFile() (Cake\Http\Response method), 199
withHeader() (Cake\Http\Response method), 200
withModified() (Cake\Http\Response method), 204
withSharable() (Cake\Http\Response method), 202
withStringBody() (Cake\Http\Response method), 200
withType() (Cake\Http\Response method), 198
withVary() (Cake\Http\Response method), 204
wrap() (Cake\Utility\Text method), 781
wrapBlock() (Cake\Utility\Text method), 782
writable() (Cake\Filesystem\File method), 739
write() (Cake\Cache\Cache method), 536
write() (Cake\Cache\CacheEngine method), 540
write() (Cake\Controller\Component\CookieComponent method), 235
write() (Cake\Core\Configure method), 157
write() (Cake\Filesystem\File method), 739
write() (Cake\Log\Log method), 630
write() (Session method), 662
writeMany() (Cake\Cache\Cache method), 536
WWW_ROOT (global constant), 801

X

Xml (class in Cake\Utility), 795
XmlView (class), 278

Y

YEAR (global constant), 802
year() (Cake\View\Helper\FormHelper method), 310

Z

zip() (Cake\Collection\Collection method), 722