4 Controllers: Programming Application Logic

Controller, the name suggests its job—it controls, supervises, and manages. In CakePHP, controllers are the classes that handle browser requests and facilitate communication between models and views. It is the central hub where application logics are defined to control program flows of browser requests.

In CakePHP, every public method of a controller is called 'action'. Each action represents a URL. When a URL is requested from browser, the respective controller action is invoked. A controller generally uses a model class to manipulate and process the user data. Once the data is processed, controller takes it from the model and forwards it to the appropriate view file. The view file is then sent back as the response and displayed in the user's browser. In such a way, controller coordinates between the user, the model, and the views.

In this chapter, we will learn the nuts and bolts of CakePHP controller. We will particularly find out:

1. How to interact with model classes from controllers
2. How to pass controller data to the view
3. How to create a controller action and use action parameters
4. How to get form data from view
5. How to redirect to another action
6. How to add common functionalities to all controllers
7. How to create reusable components that can be used to add functionalities to controllers

Interacting with Model

Most commonly, one single controller manages the logic for one single model. In chapter 3, we already saw how CakePHP automatically finds out that relevant model's class name from a controller's name. The related model class is automatically associated and can be accessed from the controller—we don't need to configure it in the controller class explicitly. In the previous chapter, we also saw an example of this automatic binding. We created a `TasksController` class and CakePHP automatically found out and attached the related model `Task` (through its naming convention) with the controller. We were able to access the `Task` model from the `TasksController` as if that model class is a controller attribute (`$this->Task`).

Attaching Models and Controllers

In CakePHP, generally, every controller has one dependent model class. That's the way Cake is designed to be used. CakePHP will always look for a related model class for a controller through its naming convention unless a controller-model attachment is explicitly defined in that controller. Now, in some unusual situations, we may need a controller that does not have any dependency on any model class. In that case, we have to configure our controller to handle this scenario. Let's see how such a model-less controller can be created.

Time for Action: Controller without a Model

1. Put a fresh copy of CakePHP inside your web root folder. Rename the folder to `applogic`.
2. Inside the `/app/controllers/` directory, create a new PHP file `books_controller.php` and write the following code inside it.

```
<?php
class BooksController extends AppController {

   var $name = 'Books';
   var $uses = array();

   function index() {
      //nothing's here
   }
}
?>
```

3. Inside the `/app/views/` directory, create a new folder `books`. Create a new view file named `index.ctp` there (`/app/views/books/index.ctp`), with the following code:

```
<h2>Packt Book Store</h2>
<p>Coming Soon!</p>
```

4. Now, visit the following URL and see what shows up in the browser:
   `http://localhost/applogic/books/`

What Just Happened?

At first, we have created a new CakePHP project. We already know how to create and configure a new Cake project from Chapter 2. In this case, as we don't need any database, we did not set up the database configuration file (`/app/config/database.php`). Cake will not find any database configuration file but it will work.

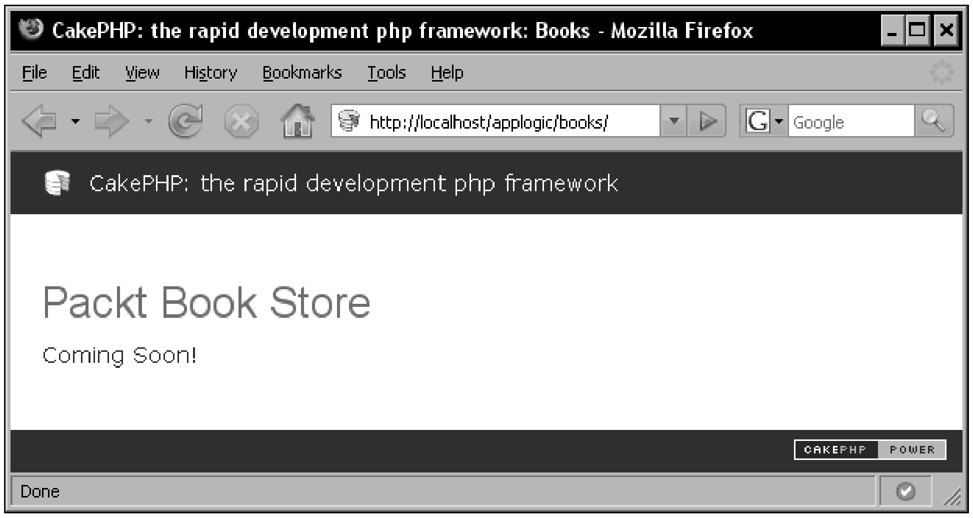
We then created a controller class named `BooksController`. Inside the controller, we defined an attribute named `$uses`. The `$uses` attribute is a special controller attribute that is used to explicitly define the relevant model class name of a controller. If `$uses` is not defined, Cake tries to find out the relevant model name through its naming convention. We assigned an empty array to this `$uses` attribute in `BooksController`. It means that `BooksController` does not use any model class. We could also assign `$uses` to `null` like the following, which would also do the same:

```
var $uses = array();
```

We then wrote an action named `index()` inside the `BooksController`. And, we also created the corresponding view file (`app/books/index.ctp`) for this particular action.

The `index()` action contains no code. And hence, when this action will be requested, Cake will just render its related view file.

When someone visits the URL `http://localhost/applogic/books/`, the default action (that is `index()`) of the `BooksController` is invoked, and the related view file is rendered. It displays something like the following in the browser:


CakePHP: the rapid development php framework: Books - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/applogic/books/
Google
CakePHP: the rapid development php framework

Packt Book Store

Coming Soon!

In CakePHP, we can associate models with controllers in 2 ways:

1. **Automatic binding**: CakePHP automatically binds a model with a controller through its naming convention. Like a controller named `BooksController` will be tied with a model named `Book` automatically (unless something else is manually defined).
2. **Manual binding**: If we want to override the automatic binding, we can assign `$uses` controller attribute to an array of models. Those models will be available to the controller.

'Convention over configuration' is one of the principal philosophies of CakePHP framework. It is recommended to follow the naming conventions of controllers and models and let Cake attach related controllers and models automatically. It would simplify things.

We have already seen how the second method (Manual binding) works. We assigned an empty array to `$uses` attribute of `BooksController` to tell Cake that this controller has no dependency on any model class. We could also manually attach more than one model(s) to a controller using the `$uses` attribute. In that case, we just have to put all the model names in the `$uses` attribute, like this:

```
$uses = array ( 'ModelName1', 'ModelName2' ) ;
```

We just learnt how controllers can be tied up with models. Now, we will see how they can interact with the presentation files, a.k.a views.

Action, Parameters, and Views

In CakePHP, actions are public methods of controllers that represent URLs. A general Cake URL contains suffixes like `/controller_name/action_name` and from this pattern Cake automatically maps the URL with a controller's action. Again, every such controller action can automatically call a view file that contains the display logic for that particular action. The appropriate view file is determined from the controller and action names. As an example, if the `index()` action of the `BooksController` is requested, the view file in `/app/views/books/index.ctp` will be rendered. We very often need to supply processed data to those view files from controllers, so that we can present the data in a suitable format to the user.

Interacting with View

CakePHP determines the appropriate view file for a controller's action by its naming convention. Controller can also supply processed data to those view files. To do that we can use the controller method `set()`. In chapter 3, we saw some uses of this `set()` method. In this section, we will learn some more on how we can interact with view files from controllers.

Time for Action: Passing Variables to a View

1. Change the `index()` action of the `BooksController` (`/app/controllers/books_controller.php`).

```
<?php
class BooksController extends AppController {

   var $name = 'Books';
   var $uses = array();

   function index() {
      $this->set('page_heading', 'Packt Book Store');

      $book  = array (
               'book_title'   => 'Object Oriented Programming
                                                 with PHP5',
               'author'      => 'Hasin Hayder',
               'isbn'               => '1847192564',
               'release_date' => 'December 2007'
              );
      $this->set($book);

      $this->pageTitle = 'Welcome to the Packt Book Store!';
   }
}
?>
```

2. Change view file `index.ctp` (`/app/views/books/index.ctp`) with the following code:

```
<h2><?php echo $page_heading; ?></h2>
<dl>
<lh><?php echo $bookTitle; ?></lh>
<dt>Author:</dt><dd><?php echo $author; ?></dd>
<dt>ISBN:</dt><dd><?php echo $isbn; ?></dd>
<dt>Release Date:</dt><dd><?php echo $releaseDate; ?></dd>
</dl>
```

3. Now enter the following URL in your browser.

   `http://localhost/applogic/books/`.

What Just Happened?

In the `index()` action, we first used the `set()` method to set a view variable named `page_heading`.

```
$this->set('page_heading', 'Packt Book Store');
```

The first parameter of `set()` specifies the view variable's name and the second parameter defines its value. In the view file, in the first line, we simply printed out the `$page_heading` variable that displays the text `Packt Book Store` (that was set in the controller).

In the `index()` action, we then created an associative array named `$book`.

```
$book   = array (
               'book_title'   => 'Object Oriented Programming with
PHP5',
               'author'    => 'Hasin Hayder',
               'isbn'          => '1847192564',
               'release_date' => 'December 2007'
              );
```

And then passed this array to the view files using the `set()` method like this:

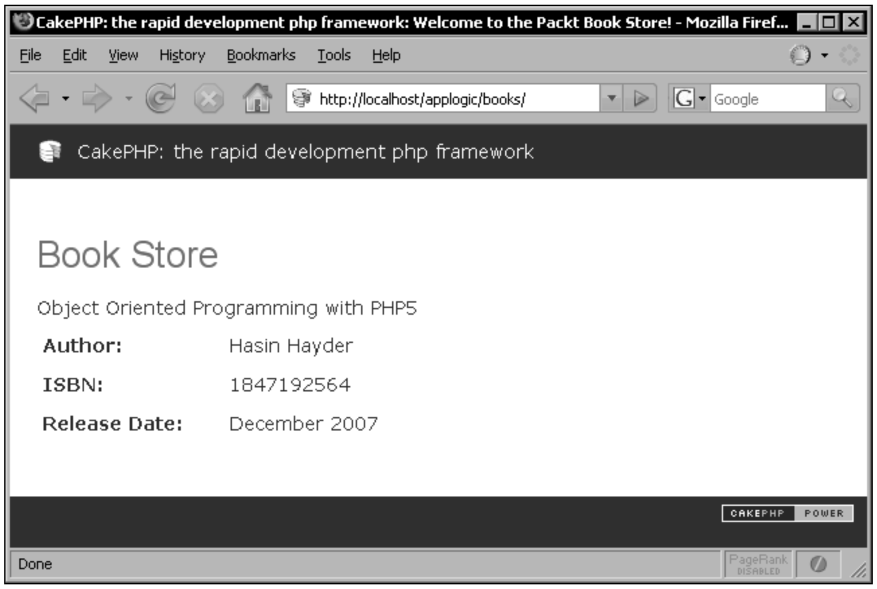
```
$this->set($book);
```

As we can see, the `set()` method can take a single parameter as well. We can create an associative array (as we created the `$book` array) and pass that array to the `set()` method. It will automatically set all these `key=>value` pairs of the associative array respectively, as view variables and their values. This method can be pretty handy if we want to assign a set of variables to the view quickly. One thing to be noted, in this case, all the underscored array keys will become CamelCased view variables. Like, in our case, the `book_title` and `release_date` keys set in the controller became `$bookTitle` and `$releaseDate` variables in the correspondent view. Inside the view file, we then printed out all those variables set through the associative array `$book`.

Lastly, in the controller action, we defined a controller attribute named `$pageTitle`.

```
$this->pageTitle = 'Welcome to the Packt Book Store!';
```

`$pageTitle` is a special attribute that sets the title of the rendered page. Now, if we visit this page it would look something like the following:



CakePHP: the rapid development php framework: Welcome to the Packt Book Store! - Mozilla Firef...
File Edit View History Bookmarks Tools Help
http://localhost/applogic/books/
Google
CakePHP: the rapid development php framework
CAKEPHP POWER
Done

Book Store

Object Oriented Programming with PHP5

Author:	Hasin Hayder
ISBN:	1847192564
Release Date:	December 2007

Actions and Parameters

In chapter 3, we already learned how parameters can be passed to a controller action by adding suffixes to the URL. A typical Cake URL looks like this: `http://yourhost/controller/[/action][/parameters]`. The elements in the URL that are appended to the hostname and separated by / are known as request parameters. We will now see more closely how these request parameters are handled by controller actions.

Time for Action: Understanding Actions and Parameters

1. Change the `index()` action of the `BooksController` like the following:

```
<?php
class BooksController extends AppController {

   var $name = 'Books';
   var $uses = array();

   function index( $id = 0 ) {
      $books = array (
                        '0' => array (
                              'book_title' => 'Object Oriented
                                               Programming with PHP5',
                                'author'       => 'Hasin Hayder',
                                'isbn'                => '1847192564',
                                'release_date' => 'December 2007'
                             ),
                         '1' => array (
                                'book_title'   => 'Building Websites
                                                   with Joomla! v1.0',
                                'author'       => 'Hagen Graf',
                                'isbn'                => '1904811949',
                                'release_date' => 'March 2006'
                         )
                      );
          $id = intval($id);
        if( $id < 0 || $id >= count($books) ) {
           $id = 0;
        }
        $this->set($books[$id] );
        $this->set('page_heading', 'Book Store');
        $this->pageTitle = 'Welcome to the Packt Book Store!';
    }
}
?>
```

2. Now visit the following links and see what shows up in the browser:
   `http://localhost/applogic/books/index/0`
   `http://localhost/applogic/books/index/1`
   `http://localhost/applogic/books/index/xyz`

What Just Happened?

We first recreated the `BooksController`'s action `index()`.The `index()` action can take a parameter named `$id`:

```
function index( $id = 0 ) {
```

That means, if someone requests the URL `http://localhost/applogic/books/index/1`, the `$id` parameter of the `index()` action will be set to `1`. The default value of the parameter `$id` is `0`. So, if no request parameter is provided through URL (like `http://localhost/applogic/books/index/`), the parameter `$id` will have the value `0`.

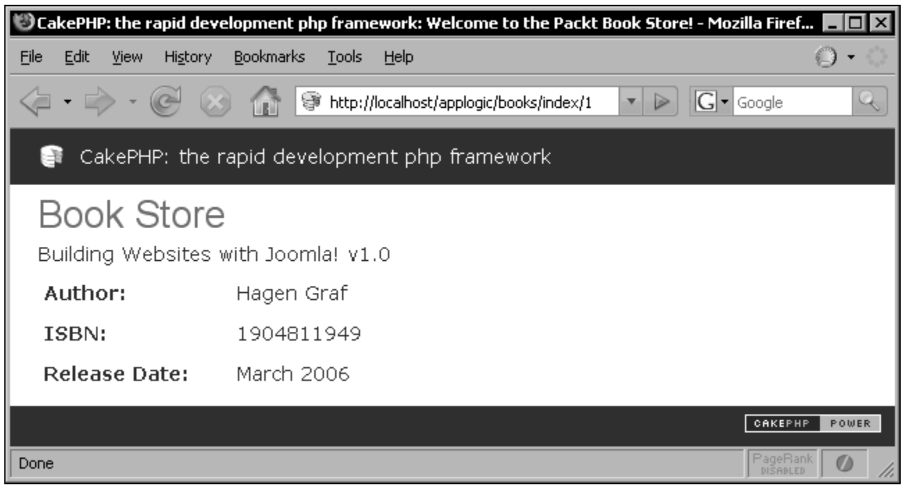
We also defined an array named `$books` inside the `index()` action. The `$books` array has two elements containing information about two different books.

```
$books    = array (
                    '0' => array (
                          'book_title' => 'Object Oriented
Programming with PHP5',
                           'author'    => 'Hasin Hayder',
                           'isbn'             => '1847192564',
                           'release_date' => 'December 2007'
                    ),
                   '1' => array (
                           'book_title'   => 'Building Websites with
Joomla! v1.0',
                          'author'     => 'Hagen Graf',
                          'isbn'              => '1904811949',
                          'release_date' => 'March 2006'
                    )
                );
```

Now, we want to show the appropriate book information depending on the request parameter. That is if `0` is supplied as the request parameter, we will show information of the 0th book. We can get the request parameter's value through `$id`. We just passed the `$id`th element of the `$books` array to the view file:

```
$this->set($books[$id] );
```

The view file (`/app/views/books/index.ctp`) that we created in the previous *Time for Action* will work for this one too, without any change. Now, if someone visits the URL `http://localhost/applogic/books/index/0`, the `index()` action will show the information of the first book. If `1` is supplied as parameter `$id`, the second book's information will be displayed.


CakePHP: the rapid development php framework: Welcome to the Packt Book Store! - Mozilla Firef...
File Edit View History Bookmarks Tools Help
http://localhost/applogic/books/index/1
Google
CakePHP: the rapid development php framework
CAKEPHP POWER
Done

Book Store

Building Websites with Joomla! v1.0

Author: Hagen Graf

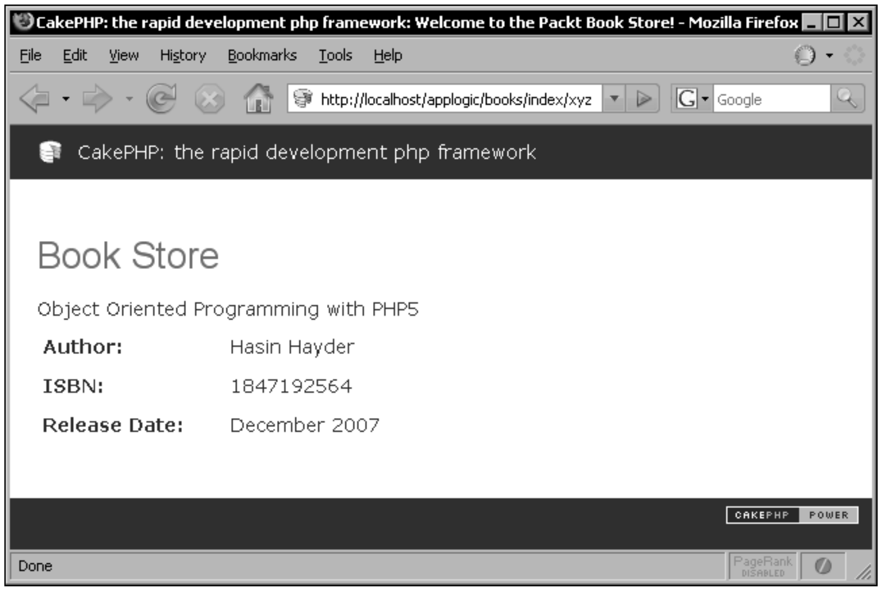
ISBN: 1904811949

Release Date: March 2006

But what if the parameter supplied has got a value that is unacceptable parameter—like a non integer, or an integer that is less than `0` or greater than `1` (the highest index of the `$books` array). In those cases, our code would just throw errors. To handle these exceptional cases, we added an `if` condition in the action. If any of those exception happens, we made `$id = 0`, so that we can bypass the errors gracefully.

```
$id = intval($id);
if( $id < 0 || $id >= count($books) ) {
$id = 0;
}
```

Now if we go to the URL `http://localhost/applogic/books/index/xyz`, it would just show the information of the first book.


CakePHP: the rapid development php framework: Welcome to the Packt Book Store! - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/applogic/books/index/xyz
Google
CakePHP: the rapid development php framework
CAKEPHP POWER
Done
PageRank DISABLED

Book Store

Object Oriented Programming with PHP5

Author:	Hasin Hayder
ISBN:	1847192564
Release Date:	December 2007

Following the same technique, we can have more than one parameter for a particular action. In the next *Time for Action*, we will see an example of such a controller action.

Time for Action: Handling more than One Request Parameter

1. Create a new controller `MathsController` with the following code:

```
<?php
class MathsController extends AppController {

   var $name = 'Maths';
   var $uses = array();
```

```
        function add_digits( $digit1 = 0, $digit2 = 0, $digit3 = 0 ) {
            $sum = intval($digit1) + intval($digit2) + intval($digit3);
            $this->set('sum', $sum);
        }
    }
    ?>
```

2. Create its corresponding view file `add_digits.ctp` (`/app/views/maths/add_digits.ctp`) using the following code:

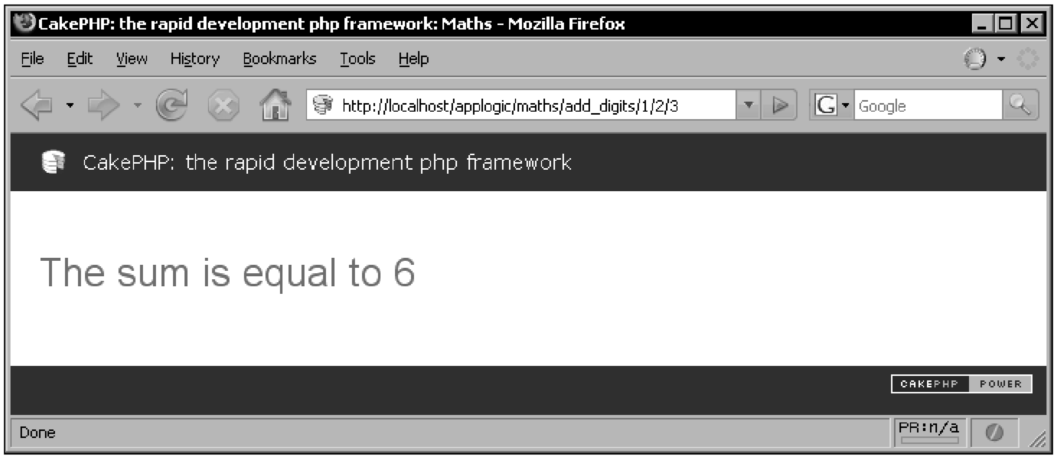
```
<h2>The sum is equal to <?php echo $sum; ?></h2>
```

3. Now visit the following links and see what shows up in the browser:

```
http://localhost/applogic/maths/add_digit/1/2/3
http://localhost/applogic/books/index/1/2
```

What Just Happened?

We first created a new controller named `MathsController`. Inside the controller, we wrote an action called `add_digits()`. The `add_digits()` action takes 3 parameters. Each having a default value `0`. When this action is requested through URL, it would sum up the digits and pass the result to its view file. We created a very simple view file to display the sum supplied from the controller action. Now, if we visit the URL `http://localhost/applogic/maths/add_digits/1/2/3` it should display the sum of the three numbers 1, 2 and 3, like the following:


CakePHP: the rapid development php framework: Maths - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/applogic/maths/add_digits/1/2/3
Google
CakePHP: the rapid development php framework
CAKEPHP POWER
Done
PR:n/a

The sum is equal to 6

How Cake Handles an Incoming Request?

Cake automatically routes to the appropriate controller and action. It also loads the appropriate model class and displays the correct view file. If all the conventions are followed correctly, it always does the right thing for us without any extra configuration. We will now skim through the whole process to have a clearer understanding of how Cake handles incoming requests. Let's take the `BooksController` example and assume a request has arrived in the URL `http://localhost/applogic/books/index/123`. Now, let's see what Cake will basically do with this request:

1. Cake accepts an incoming request from a browser. It determines the target controller's name that should handle this particular request from the first request parameter. In this case, the target controller is the `BooksController`.
2. It instantiates an object of class `BooksController` found in the file `books_controller.php` in the directory `/app/controllers/`.
3. It loads the related model class `Books` from the file `books.php` found in the directory `/app/models/`. In our case, the target controller is 'configured' to use no model and hence actually it does not load or look for any related model class.
4. The next request parameter suggests the action to be invoked in the target controller. If no second request parameter is provided, it routes to the default action that is `index()`. In this case, there is a second parameter and it suggests the target action is `index()`.
5. Cake invokes that target action of the target controller, that is the `index()` action of our `BooksController`. If some additional parameters are supplied with the request, Cake passes them as parameters while calling the target action. In our case, one additional request parameter is provided with value `123`. So, `123` is passed to the `index()` method of `BooksController` as parameter.
6. It locates the appropriate view file from the target controller and target action's names that is the file `index.ctp` inside the directory `/app/views/books/`. It passes view variables that were set in the controller action to that file. It then renders that view file and the rendered view is sent to browser as a response.

So, this is how an incoming request is generally handled by Cake. We can, though, override the default behavior by configuring our controllers, models or routes. We will learn about them all through this book. So keep reading...

Getting Post Data from the View

HTML forms are the most common way of taking user inputs in web applications. In chapter 3, we already saw how to create simple HTML forms using CakePHP's `FormHelper`. In this section, we will see how to access the submitted form data from a controller action. Let's dive into it...

Time for Action: Getting Post Data from the View

1. Create a controller class `UsersController` using the following code and save it as `users_controller.php` inside the `/app/controllers/` directory.

```
<?php
class UsersController extends AppController {

   var $name = 'Users';
   var $uses = array();

   function index() {
      if (!empty($this->data)) {
         //data posted
         echo $this->data['name'];
         $this->autoRender = false;
      }
   }
}
?>
```

2. Create a view file named `index.ctp` inside the `/app/views/users/` with the following code:

```
<?php echo $form->create(null, array('action' => 'index'));?>
<fieldset>
<legend>Enter Your Name</legend>
<?php echo $form->input('name'); ?>
</fieldset>
<?php echo $form->end('Go');?> .
```

3. Now visit the following links and see what shows up in the browser:

   `http://localhost/applogic/users/`.

What Just Happened?

Before going through the code, let's first learn about a special controller attribute `$data`. This attribute is used to store the POST data sent from HTML forms to the controller. When some data is submitted from an HTML form, Cake automatically fills up the attribute `$data` with the posted data. That's all we need to know. Now, we are ready to step forward and check out what the above piece of code is actually doing.

In the first line of the `index()` action, we checked if `$this->data` attribute is empty. If `$this->data` is not empty, this means some data is submitted from an HTML form. In that case, we will execute the codes inside the `if{}` block. Let's ignore what's going on inside this block for now and check out what's going to happen if no data is submitted. In that case, we will skip the `if{}` block and directly render the corresponding view file (`/apps/views/users/index.ctp`).

Inside the view, we created a simple form using CakePHP's built-in `FormHelper`. We will learn more about the `FormHelper` in Chapter 7. For now, just understand that we can define the model name through the first parameter of the `FormHelper`'s `create()` method. In our case, we defined it as `null`, which means the form fields do not belong to any model. And in the second parameter, we specified that the form is to be submitted to the `index()` action of the current controller.

```
<?php echo $form->create(null, array('action' => 'index'));?>
```

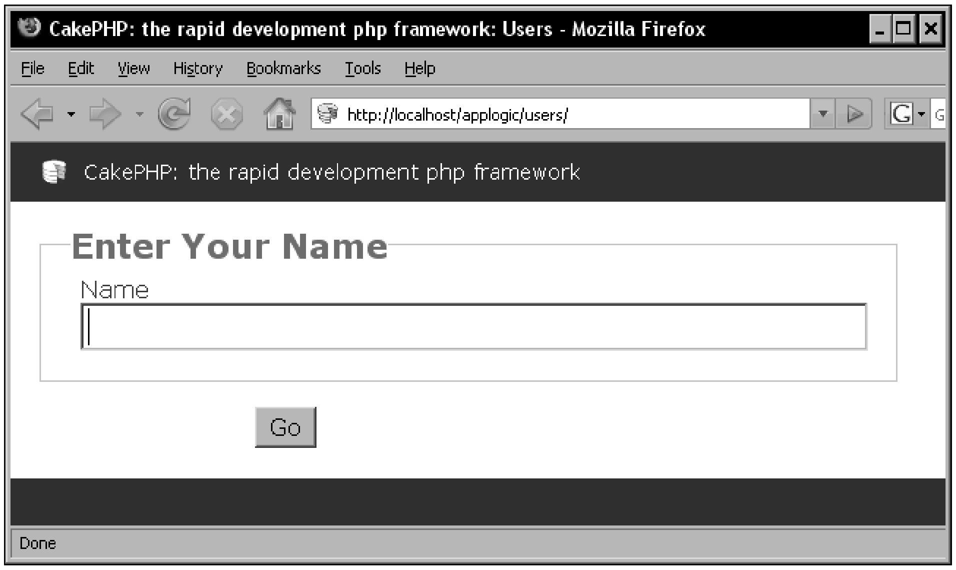
We then created a text input field using the `input()` method of the `FormHelper`. As we want to take user's name using this field, we supplied `name` as a parameter to this `input()` method.

```
<?php echo $form->input('name'); ?>
```

This `input()` method will generate an HTML input element like this:

```
<input type="text" id="Name" value="" name="data[name]"/>
```

At the end of this view file, we ended the form with a submit button called **Go** created using the `FormHelper`'s `end()` method. Now, if we visit the page `http://localhost/applogic/users/` it will display a form like the following:


CakePHP: the rapid development php framework: Users - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/applogic/users/
CakePHP: the rapid development php framework
Enter Your Name
Name
Go
Done

Now, if a name is entered in the input field and the form is submitted, the controller attribute `$data` will be filled up with the submitted POST data. The condition `if (!empty($this->data))` will return a `true`. And the codes inside the `if{}` block will get executed. Inside the block, we directly printed out the value entered in the input field:

```
echo $this->data['name'];
```

Notice how we accessed the value entered in the input field through the `$data` attribute. The key `name` of the `$data` array simply returns the value entered in the field named `name`.

The next line sets the `$autoRender` attribute of the controller to `false`:

```
$this->autoRender = false;
```

It tells the controller that it does not have to render any view. One important thing to be noted: generally, it is not possible to print outputs from a controller action. To do that we have to set the `$autoRender` attribute to `false`. It is not recommended to generate output from controller either, but it sometimes helps for quick debugging.

Redirecting

In dynamic web applications, redirects are often used to control the flow of the application and move the user from one page to another. Without redirects, web applications would be scattered pages without any real flow. In usual PHP-based web applications, the PHP function `header()` is very commonly used to redirect the user from one page to another. Whereas, CakePHP offers a built-in controller method `redirect()` that is used to direct the user from one action to another. Now, without further ado, let's see how to redirect in CakePHP!

Time for Action: Redirecting from One Action to Another

1. Modify the `UsersController` (`/app/controllers/users_controller.php`).

```
class UsersController extends AppController {

   var $name = 'Users';
   var $uses = array();

   function index() {
      if (!empty($this->data)) {
         $this->redirect(array('controller'=>'users',
          'action'=>'welcome', urlencode($this->data['name'])));
      }
   }

   function welcome( $name = null ) {
      if(empty($name)) {
```

```
                $this->Session->setFlash('Please provide your name!',
                                                          true);
                    $this->redirect(array('controller'=>'users',
                                         'action'=>'index'));
            }
            $this->set('name', urldecode($name));
        }
    }
```

2. Create a view file named `welcome.ctp` inside the directory `/app/views/users/` with the following code:

```
<h2>Welcome, <?php echo $name; ?></h2>
```

3. Now enter the following URL in your browser:

```
http://localhost/applogic/users/
```

What Just Happened?

In the `index()` action of the `UsersController`, we changed the code inside the `if{}` block. Instead of printing out the name submitted from the form, we redirected the user to another action using the following line of code:

```
$this->redirect(array('controller'=>'users', 'action'=>'welcome',
urlencode($this->data['name'])));
```

The `redirect()` method is pretty easy to use. The first parameter is an associative array. We specified the controller and action name (where we want to redirect) there through `controller` and `action` keys. Also, additional request parameters can be supplied by appending new elements in this array. When the form is submitted, we redirected the user to the `UsersController`'s `welcome()` action. We also appended the user's name in the URL as a request parameter. We got the name from the `$data` controller attribute and used the PHP function `urlencode()` to make that input URL-friendly.

`redirect()` by default exits just after the execution. So, codes after the `redirect()` call do not execute, we can though set the third parameter to true while calling the `redirect()` method to avoid the force exit.

Now, let's have a look at the `welcome()` action. This action takes an optional parameter `$name`. Inside this action, we first checked if any request parameter is supplied. If no parameter is specified, we set a flash message and then sent back the user to the `index()` action of the `UsersController` using the `redirect()` method.

```
if(empty($name)) {
   $this->Session->setFlash('Please provide your name!');
   $this->redirect(array('action'=>'index'));
}
```

See, this time while calling the `redirect()` method, we only set the `action` key in the parameter. When the `controller` key is not set, the `redirect()` method redirects to the current controller's action.

If the request parameter `$name` is provided, it will ignore the `if{}` block and pass the URL-decoded name to the view as `$name` variable.

```
$this->set('name', urldecode($name));
```

In the view file, we simply welcomed the user by displaying a 'Welcome,' text before the user's name.

Now, if we visit the URL `http://localhost/applogic/users/` and enter a name there, it should redirect us to the `welcome` action and show us a greeting message.

AppController: The Parent Controller

From Object Oriented perspective, every CakePHP controller is a subclass of the `AppController` class. Recall how we start the class definition of a controller—look at the following line of code as an example:

```
class BooksController extends AppController {
```

Our controllers extend the `AppController` class—this means we set the `AppController` class as the parent class of all of our controllers. As of now, it seems alright. But the next question that quickly pops up in our mind is: where is that `AppController` class located? Well, it can be found inside the `app_controller.php` file under the `/cake/libs/controller/` directory. This `app_controller.php` is actually a placeholder file and can be overridden by our own one.

The main benefit of having our own `AppController` class is we can put in common application-wide methods inside this class and all of our controllers will just inherit them. We can use attributes and methods defined inside the `AppController` from any of our controller class. We will now see how to create our own `AppController` class and how we can make use of the methods written inside the `AppController` from a controller class.

Time for Action: Adding Common Functionalities to all Controllers

1. Create a new file named `app_controller.php` just inside the `/app/` folder with the following code,

```
<?php
class AppController extends Controller {
   function strip_and_clean ( $id, $array) {
        $id = intval($id);
        if( $id < 0 || $id >= count($array) ) {
            $id = 0;
        }
        return $id;
    }
}
?>
```

2. Modify the `BooksController`'s (`/app/controllers/books_controller.php`) code, replace the `if{}` block with the controller method `strip_and_clean()`,

```
<?php
class BooksController extends AppController {
   var $name = 'Books';
   var $uses = array();
   function index( $id = 0 ) {
      $books = array (
                     '0' => array (
                           'book_title' => 'Object Oriented
                                          Programming with PHP5',
                           'author'      => 'Hasin Hayder',
                           'isbn'          => '1847192564',
                           'release_date' => 'December 2007'
                      ),
                     '1' => array (
                           'book_title'   => 'Building Websites
                                              with Joomla! v1.0',
                           'author'       => 'Hagen Graf',
                           'isbn'                 => '1904811949',
                           'release_date' => 'March 2006'
                     )
                 );
      $id = $this->strip_and_clean( $id, $books);
      $this->set($books[$id] );
      $this->set('page_header', 'Book Store');
      $this->pageTitle = 'Welcome to the Packt Book Store!';
   }
}
?>
```

3. Now visit the following links and see what shows up in the browser:

   `http://localhost/applogic/books/index/0`

   `http://localhost/applogic/books/index/1`

   `http://localhost/applogic/books/index/xyz`

What Just Happened?

We first wrote a class named `AppController` and saved it as `app_controller.php` inside the `/app/` folder.

```
<?php
class AppController extends Controller {
```

Inside this class, we wrote a method named `strip_and_clean()`. This method takes two parameters - `$id` and `$array`.

```
function strip_and_clean ( $id, $array) {
      if( !is_int($id) || $id < 0 || $id >= count($array) ) {
         $id = 0;
      }
      return $id;
}
```

The `strip_and_clean()` method simply returns `$id`, if `$id` is a non-negative integer and `$id` is less than the length of `$array`. Otherwise it returns a `0`.

Previously, we did this checking inside the `BooksController`. And now we have moved this particular code to the `AppController` method `strip_and_clean()`. As all of our controllers inherit the `AppController` class, we can now use this particular routine from any of them.

In `BooksController`, we replaced the `if{}` block with the `strip_and_clean()` method.

```
$id = $this->strip_and_clean( $id, $books);
```

`$id` and `$books` variables are passed to this method as parameters. And the returned result of the method is then stored back to `$id`. Now, if the request parameter `$id` is invalid, it will make it to `0`. And the `/books/index/` action will show us the information about the first book defined in `$array`.

The new approach is better only when we have some common application wide methods. In those cases, we can just write them inside the `AppController` class and call them from any of our controllers. But we also have to remember that writing methods inside the `AppController` is not always the best way. There may be other controllers that do not require those methods (like we have `UsersController`, which does not need the `strip_and_clean()` method) but still will inherit them.

The better approach is to load codes on demand, load the methods only where it is required. Moreover, writing methods inside the `AppController` does not increase reusability outside the application—it just increases application-specific and application-wide reusability. We just cannot reuse the methods written inside the `AppController` for one application in any other application directly. If we want to write reusable codes and follow the DRY (Don't Repeat Yourself) principle, it is always better to write our methods in separate classes and load them only from the controllers that require those methods.

CakePHP also provides a way for writing and using reusable classes. In CakePHP, they are called components. In the next section, we will work with CakePHP components and see how it can increase reusability.

Working with Components

Components are reusable classes that can be used from any controller in CakePHP applications. By definition, components are only limited to be used from and inside controllers and its other components. They are used to help controllers with a goal of reusability—we can just port the component classes to any of our Cake applications and use them in those applications. CakePHP ships with some useful components, like `AuthComponent`, `SessionCompoenent`, etc. We will learn about them in Chapter 10. Now, let's see how to create our own custom component and use them from a controller.

Time for action: Creating and Using Reusable Components

1. Create a component class `UtilComponent` using the following code and save it in a file named `util.php` under `/app/controllers/components/` directory.

```
<?php
class UtilComponent extends Object
{
   function strip_and_clean ( $id, $array) {
      $id = intval($id);
      if( $id < 0 || $id >= count($array) ) {
            $id = 0;
      }
      return $id;
   }
}
?>
```

2. Remove the `strip_and_clean()` method from the `AppController` (`/app/app_controller.php`) class.

```
<?php
class AppController extends Controller {
}
?>
```

3. Modify the `BooksController` class (`/app/controllers/books_controller.php`),

```
<?php
class BooksController extends AppController {
   var $name = 'Books';
   var $uses = array();
   var $components = array('Util');
   function index( $id = 0 ) {
      $books = array (
                     '0' => array (
                           'book_title'   => 'Object Oriented
                                             Programming with PHP5',
                            'author'      => 'Hasin Hayder',
                            'isbn'           => '1847192564',
                            'release_date' => 'December 2007'
                     ),
                     '1' => array (
                            'book_title'   => 'Building Websites
                                                with Joomla! v1.0',
                            'author'      => 'Hagen Graf',
                            'isbn'          => '1904811949',
                            'release_date' => 'March 2006'
                     )
                );
      $id = $this->Util->strip_and_clean($id);
      $this->set('book', $books[$id] );
      $this->pageTitle = 'Welcome to the Packt Book Store!';
   }
}
?>
```

4. Now visit the following links and see what shows up in the browser

`http://localhost/applogic/books/index/0`

`http://localhost/applogic/books/index/1`

`http://localhost/applogic/books/index/xyz`

What Just Happened?

At first, we created a component class named `UtilComponent` and saved it in a file named `util.php` under the `/app/controllers/components/` folder. We then moved (cut-pasted) the `strip_and_clean()` method from the `AppController` to the `UtilComponent`.

> The class name of a component should always have a `Component` postfix. Every component class file should be placed under the `/app/controllers/components/` directory. The filenames should be named after the lowercased classname of the component (without the Component postfix).

Now, to use this `UtilComponent` from our `BooksController`, we have to add this component to the controller. It can be done through the controller attribute `$components`. `$components` holds an array of all the component names that we want to make available inside the controller. In our case, as we want to load only the `Util` component, we defined the `$components` array like the following:

```
var $components = array('Util');
```

> Notice, the component name used in the `$components` array is `Util` (without the Component postfix) not `UtilComponent`.

Once loaded, the `UtilComponent` class can be referred and used from the controller. Now instead of using the `strip_and_clean()` method of the `AppController`, we called the `UtilComponent`'s method `strip_and_clean()`.

```
$id = $this->Util->strip_and_clean($id);
```

Everything else in the controller remains unchanged. And it will work, as it was working before. The only difference is, now we have a more reusable code that can be used from any controller or any other application when needed.

Summary

A controller is used to manage the application logic. It controls the application flow and works as a bridge between the model and the view. CakePHP applications use a common format for the URL. From the URL, Cake finds out the appropriate controller action to invoke to handle a particular browser request. Request parameters can be appended to the URL and can be accessed from the actions. Controller can load a model class to interact with the database. By default, CakePHP finds out the related model class from the controller name. We can though override the default and tell the controller to load other model classes using the `$uses` attribute. Every controller action usually renders a view file to send formatted response to the browser. CakePHP has some conventions for saving and naming view files so that they can be loaded and rendered automatically. We can though skip the view rendering by setting controller attribute `$autorender` to `false`. Controller method `set()` is very commonly used to pass data to the relevant view file. Redirection is an established method to forward a user from one action to another and in CakePHP it is done using the method `redirect()`. In Cake, all controllers are subclasses of the `AppController` class. Methods defined inside the `AppController` class that can be called from different controllers. In CakePHP applications, common controller functionalities are written inside components to increase reusability.

Well, that's the end of the controller story. Now, get prepared for the next chapter which is about CakePHP models. We will learn lots of cool stuffs there!