CakePHP Cookbook Documentation

Release 2.x

Cake Software Foundation

February 08, 2013

Contents

CHAPTER 1

Getting Started

The CakePHP framework provides a robust base for your application. It can handle every aspect, from the user's initial request all the way to the final rendering of a web page. And since the framework follows the principles of MVC, it allows you to easily customize and extend most aspects of your application.

The framework also provides a basic organizational structure, from filenames to database table names, keeping your entire application consistent and logical. This concept is simple but powerful. Follow the conventions and you'll always know exactly where things are and how they're organized.

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

Blog Tutorial

Welcome to CakePHP. You're probably checking out this tutorial because you want to learn more about how CakePHP works. It's our aim to increase productivity and make coding more enjoyable: we hope you'll see this as you dive into the code.

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

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 Cake up and running without any configuration at all. Make sure you have PHP 5.2.8 or greater.
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: Cake 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. The more object-oriented programming you've done, the better: but fear not if you're a procedural fan.

4. Finally, you'll need a basic knowledge of the MVC programming pattern. A quick overview can be found in *Understanding Model-View-Controller*. Don't worry, it's only a half a page or so.

Let's get started!

Getting Cake

First, let's get a copy of fresh Cake code.

To get a fresh download, visit the CakePHP project on GitHub: http://github.com/cakephp/cakephp/downloads and download the latest release of 2.0

You can also clone the repository using git (http://git-scm.com/). `git clone git://github.com/cakephp/cakephp.git`

Regardless of how you downloaded it, place the code inside of your DocumentRoot. Once finished, your directory setup should look something like the following:

```
/path_to_document_root
    /app
    /lib
    /plugins
    /vendors
    .htaccess
    index.php
    README
```

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

Creating the Blog Database

Next, let's set up the underlying 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. Right now, we'll just create a single table to store our posts. We'll also throw in a few posts right now to use for testing purposes. Execute the following SQL statements into your database:

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

/* Then insert some posts for testing: */
INSERT INTO posts (title,body,created)
    VALUES ('The title', 'This is the post body.', NOW());
INSERT INTO posts (title,body,created)
    VALUES ('A title once again', 'And the post body follows.', NOW());
```

```
INSERT INTO posts (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 Cake's database naming conventions, and Cake'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. Cake is flexible enough to accommodate even the worst legacy database schema, but adhering to convention will save you time.

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

Cake Database Configuration

Onward and upward: let's tell Cake where our database is and how to connect to it. For many, this is the first and last time you configure anything.

A copy of CakePHP's database configuration file is found in `/app/Config/database.php.default`. Make a copy of this file in the same directory, but name it `database.php`.

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

```
public $default = array(
    'datasource' => 'Database/Mysql',
    'persistent' => false,
    'host' => 'localhost',
    'port' => '',
    'login' => 'cakeBlog',
    'password' => 'c4k3-rUl3Z',
    'database' => 'cake_blog_tutorial',
    'schema' => '',
    'prefix' => '',
    'encoding' => ''
);
```

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

Note: Remember that you'll need to have PDO, and pdo_mysql enabled in your php.ini.

Optional Configuration

There are three 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 second is defining a custom number (or "seed") for use in encryption. The third item is allowing CakePHP write access to its `tmp` folder.

The security salt is used for generating hashes. Change the default salt value by editing `/app/Config/core.php` line 187. It doesn't much matter what the new value is, as long as it's not easily guessed:

```
/**
 * A random string used in security hashing methods.
 */
Configure::write('Security.salt', 'pl345e-P45s_7h3*S@l7!');
```

The cipher seed is used for encrypt/decrypt strings. Change the default seed value by editing `/app/Config/core.php` line 192. It doesn't much matter what the new value is, as long as it's not easily guessed:

```
/**
 * A random numeric string (digits only) used to encrypt/decrypt strings.
 */
Configure::write('Security.cipherSeed', '7485712659625147843639846751');
```

The final task is to make the `app/tmp` directory web-writable. The best way to do this is to find out what user your webserver runs as (`<?php echo 'whoami'; ?>`) and change the ownership of the `app/tmp` directory to that user. The final command you run (in *nix) might look something like this:

```
$ chown -R www-data app/tmp
```

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

A Note on mod_rewrite

Occasionally a new user will run into mod_rewrite issues, so I'll mention them marginally here. If the CakePHP welcome page looks a little funny (no images or css styles), it probably means mod_rewrite isn't functioning on your system. Here are some tips to help get you up and running:

1. Make sure that an .htaccess override is allowed: in your httpd.conf, you should have a section that defines a section for each Directory on your server. Make sure the `AllowOverride` is set to `All` for the correct Directory. For security and performance reasons, do *not* set `AllowOverride` to `All` in `<Directory />`. Instead, look for the `<Directory>` block that refers to your actual website directory.
2. Make sure you are editing the correct httpd.conf rather than a user- or site-specific httpd.conf.
3. For some reason or another, you might have obtained a copy of CakePHP without the needed .htaccess files. This sometimes happens because some operating systems treat files that start with '.' as hidden, and don't copy them. Make sure your copy of CakePHP is from the downloads section of the site or our git repository.
4. Make sure Apache is loading up mod_rewrite correctly! You should see something like:

   ```
   LoadModule rewrite_module          libexec/httpd/mod_rewrite.so
   ```

   or (for Apache 1.3):

```
    AddModule              mod_rewrite.c
```

in your httpd.conf.

If you don't want or can't get mod_rewrite (or some other compatible module) up and running on your server, you'll need to use Cake's built in pretty URLs. In `/app/Config/core.php`, uncomment the line that looks like:

```
Configure::write('App.baseUrl', env('SCRIPT_NAME'));
```

Also remove these .htaccess files:

```
/.htaccess
/app/.htaccess
/app/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.

If you are installing CakePHP on a webserver besides Apache, you can find instructions for getting URL rewriting working for other servers under the *Advanced Installation* section.

Continue to *Blog Tutorial - Adding a layer* to start building your first CakePHP application.

Blog Tutorial - Adding a layer

Create a Post Model

The Model class is 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 go in `/app/Model`, and the file we'll be creating will be saved to `/app/Model/Post.php`. The completed file should look like this:

```
class Post extends AppModel {
}
```

Naming convention is very important in CakePHP. By naming our model Post, CakePHP can automatically infer that this model will be used in the PostsController, and will be tied to a database table called `posts`.

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

For more on models, such as table prefixes, callbacks, and validation, check out the *Models* chapter of the Manual.

Create a Posts Controller

Next, we'll create a controller for our posts. The controller is where all the business logic for post interaction will happen. In a nutshell, it's the place where you play with the models and get post-related work done. We'll place this new controller in a file called `PostsController.php` inside the `/app/Controller` directory. Here's what the basic controller should look like:

```
class PostsController extends AppController {
    public $helpers = array('Html', 'Form');
}
```

Now, lets 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/posts/index (which is also the same as www.example.com/posts/), they might expect to see a listing of posts. The code for that action would look something like this:

```
class PostsController extends AppController {
    public $helpers = array('Html', 'Form');

    public function index() {
        $this->set('posts', $this->Post->find('all'));
    }
}
```

Let me explain the action a bit. By defining function `index()` in our PostsController, users can now access the logic there by requesting www.example.com/posts/index. Similarly, if we were to define a function called `foobar()`, users would be able to access that at www.example.com/posts/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 (plural controller names, etc.) and create readable, understandable action names. You can map URLs to your code using "routes" 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 'posts' equal to the return value of the `find('all')` method of the Post model. Our Post model is automatically available at `$this->Post` because we've followed Cake's naming conventions.

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

Creating Post Views

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

Cake 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.

Layouts are presentation code that is wrapped around a view, and can be defined and switched between, but for now, let's just use the default.

Remember in the last section how we assigned the 'posts' variable to the view using the `set()` method? That would hand down data to the view that would look something like this:

```
// print_r($posts) output:

Array
(
    [0] => Array
        (
            [Post] => Array
                (
                    [id] => 1
                    [title] => The title
                    [body] => This is the post body.
                    [created] => 2008-02-13 18:34:55
                    [modified] =>
                )
        )
    [1] => Array
        (
            [Post] => Array
                (
                    [id] => 2
                    [title] => A title once again
                    [body] => And the post body follows.
                    [created] => 2008-02-13 18:34:56
                    [modified] =>
                )
        )
    [2] => Array
        (
            [Post] => Array
                (
                    [id] => 3
                    [title] => Title strikes back
                    [body] => This is really exciting! Not.
                    [created] => 2008-02-13 18:34:57
                    [modified] =>
                )
        )
)
```

Cake's view files are stored in `/app/View` inside a folder named after the controller they correspond to (we'll have to create a folder named 'Posts' in this case). To format this post data in a nice table, our view code might look something like this

```
<!-- File: /app/View/Posts/index.ctp -->

<h1>Blog posts</h1>
<table>
    <tr>
        <th>Id</th>
        <th>Title</th>
        <th>Created</th>
```

```
    </tr>

    <!-- Here is where we loop through our $posts array, printing out post info -->

    <?php foreach ($posts as $post): ?>
    <tr>
        <td><?php echo $post['Post']['id']; ?></td>
        <td>
            <?php echo $this->Html->link($post['Post']['title'],
array('controller' => 'posts', 'action' => 'view', $post['Post']['id'])); ?>
        </td>
        <td><?php echo $post['Post']['created']; ?></td>
    </tr>
    <?php endforeach; ?>
    <?php unset($post); ?>
</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 `HtmlHelper` class. CakePHP comes with a set of view helpers that make things like linking, form output, JavaScript and Ajax 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 Cake, 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.

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

If you happened to have clicked on one of the links we created in this view (that link a post's title to a URL /posts/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 PostsController now:

```
class PostsController extends AppController {
    public $helpers = array('Html', 'Form');

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

    public function view($id = null) {
        if (!$id) {
            throw new NotFoundException(__('Invalid post'));
        }

        $post = $this->Post->findById($id);
        if (!$post) {
            throw new NotFoundException(__('Invalid post'));
```

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

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

Notice that our view action takes a parameter: the ID of the post we'd like to see. This parameter is handed to the action through the requested URL. If a user requests `/posts/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. If a user requests `/posts/view`, we will throw a `NotFoundException` and let the CakePHP ErrorHandler take over. We also perform a similar check to make sure the user has accessed a record that exists.

Now let's create the view for our new 'view' action and place it in `/app/View/Posts/view.ctp`

```
<!-- File: /app/View/Posts/view.ctp -->

<h1><?php echo h($post['Post']['title']); ?></h1>

<p><small>Created: <?php echo $post['Post']['created']; ?></small></p>

<p><?php echo h($post['Post']['body']); ?></p>
```

Verify that this is working by trying the links at `/posts/index` or manually requesting a post by accessing `/posts/view/1`.

Adding Posts

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

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

```
class PostsController extends AppController {
    public $helpers = array('Html', 'Form', 'Session');
    public $components = array('Session');

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

    public function view($id) {
        if (!$id) {
            throw new NotFoundException(__('Invalid post'));
        }

        $post = $this->Post->findById($id);
        if (!$post) {
            throw new NotFoundException(__('Invalid post'));
        }
```

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

    public function add() {
        if ($this->request->is('post')) {
            $this->Post->create();
            if ($this->Post->save($this->request->data)) {
                $this->Session->setFlash('Your post has been saved.');
                $this->redirect(array('action' => 'index'));
            } else {
                $this->Session->setFlash('Unable to add your post.');
            }
        }
    }
}
```

Note: You need to include the SessionComponent - and SessionHelper - 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 Post 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 `CakeRequest` 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 `CakeRequest::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->data`. You can use the `pr()` or `debug()` functions to print it out if you want to see what it looks like.

We use the SessionComponent's `SessionComponent::setFlash()` method to set a message to a session variable to be displayed on the page after redirection. In the layout we have `SessionHelper::flash` which displays the message and clears the corresponding session variable. The controller's `Controller::redirect` function redirects to another URL. The param `array('action' => 'index')` translates to URL /posts i.e the index action of posts controller. You can refer to `Router::url()` function on the API (http://api20.cakephp.org) to see the formats in which you can specify a URL for various Cake 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

Cake goes a long way in 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 Cake's FormHelper in your views. The `FormHelper` is available by default to all views at `$this->Form`.

Here's our add view:

```
<!-- File: /app/View/Posts/add.ctp -->

<h1>Add Post</h1>
<?php
echo $this->Form->create('Post');
echo $this->Form->input('title');
echo $this->Form->input('body', array('rows' => '3'));
echo $this->Form->end('Save Post');
?>
```

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

```
<form id="PostAddForm" method="post" action="/posts/add">
```

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

The `$this->Form->input()` 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: `input()` will output different form elements based on the model field specified.

The `$this->Form->end()` call generates a submit button and ends the form. If a string is supplied as the first parameter to `end()`, the FormHelper outputs a submit button named accordingly along with the closing form tag. Again, refer to *Helpers* for more on helpers.

Now let's go back and update our `/app/View/Posts/index.ctp` view to include a new "Add Post" link. Before the `<table>`, add the following line:

```
<?php echo $this->Html->link(
    'Add Post',
    array('controller' => 'posts', '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 Post model and make a few adjustments:

```
class Post extends AppModel {
    public $validate = array(
        'title' => array(
            'rule' => 'notEmpty'
        ),
        'body' => array(
            'rule' => 'notEmpty'
        )
    );
}
```

The `$validate` array tells CakePHP how to validate your data when the `save()` method is called. Here, I've specified that both the body and title fields must not be empty. 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 *Data Validation*.

Now that you have your validation rules in place, use the app to try to add a post with an empty title or body to see how it works. Since we've used the `FormHelper::input()` method of the FormHelper to create our form elements, our validation error messages will be shown automatically.

Editing Posts

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 PostsController would look like:

```
public function edit($id = null) {
    if (!$id) {
        throw new NotFoundException(__('Invalid post'));
    }

    $post = $this->Post->findById($id);
    if (!$post) {
        throw new NotFoundException(__('Invalid post'));
    }

    if ($this->request->is('post') || $this->request->is('put')) {
        $this->Post->id = $id;
        if ($this->Post->save($this->request->data)) {
            $this->Session->setFlash('Your post has been updated.');
            $this->redirect(array('action' => 'index'));
        } else {
            $this->Session->setFlash('Unable to update your post.');
        }
    }

    if (!$this->request->data) {
        $this->request->data = $post;
    }
}
```

This action first ensures that the user has tried to access an existing record. If they haven't passed in a passed in an `$id` parameter, or the post does not exist, we throw a `NotFoundException` for the CakePHP ErrorHandler to take care of.

Next the action checks that the request is a POST request. If it is, then we use the POST data to update our Post record, or kick back and show the user validation errors.

If there is no data set to `$this->request->data`, we simply set it to the previously retrieved post.

The edit view might look something like this:

```
<!-- File: /app/View/Posts/edit.ctp -->

<h1>Edit Post</h1>
<?php
    echo $this->Form->create('Post');
    echo $this->Form->input('title');
```

```
    echo $this->Form->input('body', array('rows' => '3'));
    echo $this->Form->input('id', array('type' => 'hidden'));
    echo $this->Form->end('Save Post');
```

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

One thing to note here: CakePHP will assume that you are editing a model if the 'id' field is present in the data array. If no 'id' is present (look back at our add view), Cake will assume that you are inserting a new model when `save()` is called.

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

```
<!-- File: /app/View/Posts/index.ctp  (edit links added) -->

<h1>Blog posts</h1>
<p><?php echo $this->Html->link("Add Post", array('action' => 'add')); ?></p>
<table>
    <tr>
        <th>Id</th>
        <th>Title</th>
                <th>Action</th>
        <th>Created</th>
    </tr>

<!-- Here's where we loop through our $posts array, printing out post info -->

<?php foreach ($posts as $post): ?>
    <tr>
        <td><?php echo $post['Post']['id']; ?></td>
        <td>
            <?php echo $this->Html->link($post['Post']['title'], array('action' => 'view',
        </td>
        <td>
            <?php echo $this->Html->link('Edit', array('action' => 'edit', $post['Post']['
        </td>
        <td>
            <?php echo $post['Post']['created']; ?>
        </td>
    </tr>
<?php endforeach; ?>

</table>
```

Deleting Posts

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

```
public function delete($id) {
    if ($this->request->is('get')) {
        throw new MethodNotAllowedException();
    }
```

```
    if ($this->Post->delete($id)) {
        $this->Session->setFlash('The post with id: ' . $id . ' has been deleted.');
        $this->redirect(array('action' => 'index'));
    }
}
```

This logic deletes the post specified by $id, and uses `$this->Session->setFlash()` to show the user a confirmation message after redirecting them on to `/posts`. If the user attempts to do a delete using a GET request, we 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 posts, however:

```
<!-- File: /app/View/Posts/index.ctp -->

<h1>Blog posts</h1>
<p><?php echo $this->Html->link('Add Post', array('action' => 'add')); ?></p>
<table>
    <tr>
        <th>Id</th>
        <th>Title</th>
        <th>Actions</th>
        <th>Created</th>
    </tr>

<!-- Here's where we loop through our $posts array, printing out post info -->

    <?php foreach ($posts as $post): ?>
    <tr>
        <td><?php echo $post['Post']['id']; ?></td>
        <td>
            <?php echo $this->Html->link($post['Post']['title'], array('action' => 'view',
        </td>
        <td>
            <?php echo $this->Form->postLink(
                'Delete',
                array('action' => 'delete', $post['Post']['id']),
                array('confirm' => 'Are you sure?'));
            ?>
            <?php echo $this->Html->link('Edit', array('action' => 'edit', $post['Post']['
        </td>
        <td>
            <?php echo $post['Post']['created']; ?>
        </td>
    </tr>
    <?php endforeach; ?>

</table>
```

Using `postLink()` will create a link that uses Javascript to do a POST request deleting our post. 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 a post.

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 *Routes Configuration*.

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

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

```
Router::connect('/', array('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:

```
Router::connect('/', array('controller' => 'posts', 'action' => 'index'));
```

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

Note: CakePHP also makes use of 'reverse routing' - if with the above route defined you pass `array('controller' => 'posts', 'action' => 'index')` to a function expecting an array, the resultant 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 too.

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 Cake application you're ready for the real thing. Start your own project, read the rest of the `Cookbook` and API (http://api20.cakephp.org).

If you need help, come see us in #cakephp. 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. *Scaffolding*: Prototyping before creating code
4. *Code Generation with Bake* Generating basic CRUD code
5. *Simple Authentication and Authorization Application*: User authentication and authorization tutorial

Additional Reading

A Typical CakePHP Request

We've covered the basic ingredients in CakePHP, so let's look at how objects work together to complete a basic request. Continuing with our original request example, let's imagine that our friend Ricardo just clicked on the "Buy A Custom Cake Now!" link on a CakePHP application's landing page.

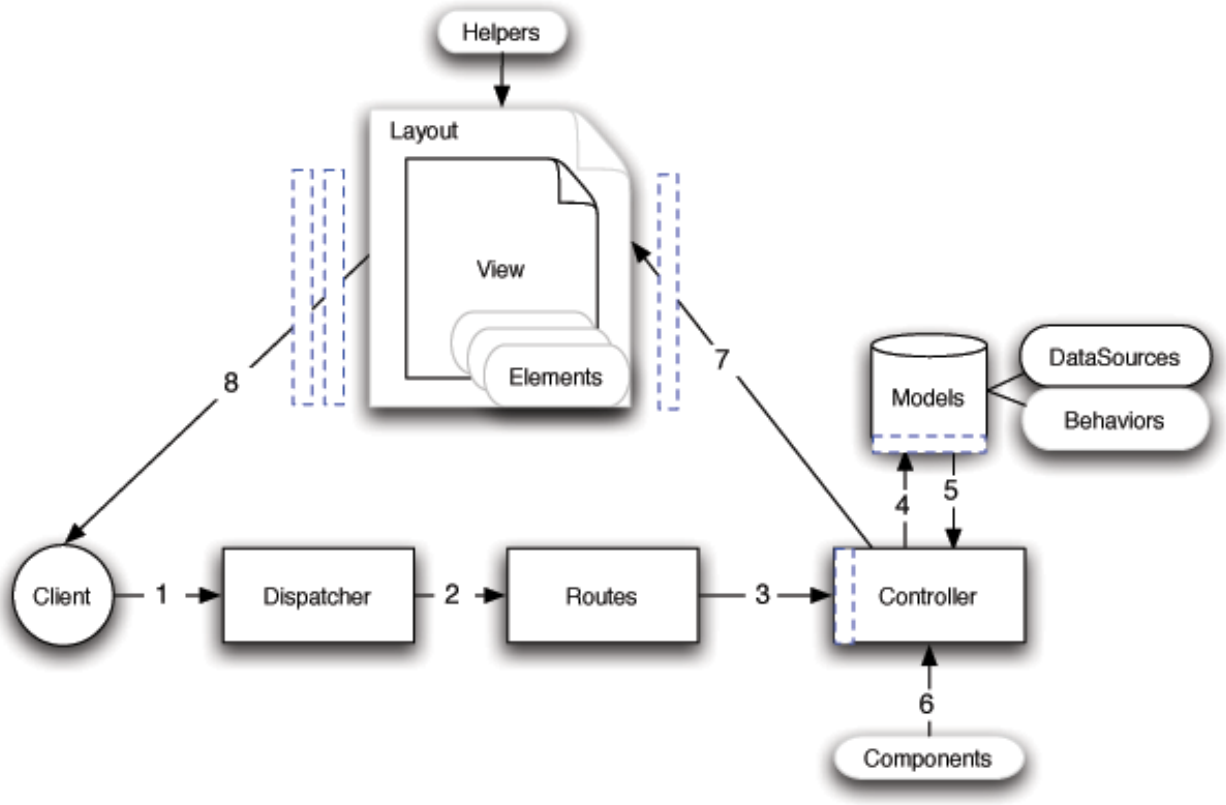

Figure 1.1: Flow diagram showing a typical CakePHP request

Figure: 2. Typical Cake Request.

Black = required element, Gray = optional element, Blue = callback

1. Ricardo clicks the link pointing to http://www.example.com/cakes/buy, and his browser makes a request to your web server.
2. The Router parses the URL in order to extract the parameters for this request: the controller, action, and any other arguments that will affect the business logic during this request.
3. Using routes, a request URL is mapped to a controller action (a method in a specific controller class). In this case, it's the buy() method of the CakesController. The controller's beforeFilter() callback is called before any controller action logic is executed.
4. The controller may use models to gain access to the application's data. In this example, the controller uses a model to fetch Ricardo's last purchases from the database. Any applicable model callbacks, behaviors, and DataSources may apply during this operation. While model usage is not required, all CakePHP controllers initially require at least one model.
5. After the model has retrieved the data, it is returned to the controller. Model callbacks may apply.
6. The controller may use components to further refine the data or perform other operations (session manipulation, authentication, or sending emails, for example).
7. Once the controller has used models and components to prepare the data sufficiently, that data is handed to the view using the controller's set() method. Controller callbacks may be applied before the data is sent. The view logic is performed, which may include the use of elements and/or helpers. By default, the view is rendered inside of a layout.
8. Additional controller callbacks (like afterFilter) may be applied. The complete, rendered view code is sent to Ricardo's browser.

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 convention, you get free functionality, and you free yourself from the maintenance nightmare of tracking config files. Convention also makes for a very uniform system development, allowing other developers to jump in and help more easily.

CakePHP's conventions have been distilled out of years of web development experience and best practices. While we suggest you use these conventions while developing with CakePHP, we should mention that many of these tenets are easily overridden – something that is especially handy when working with legacy systems.

Controller Conventions

Controller classnames are plural, CamelCased, and end in `Controller`. `PeopleController` and `LatestArticlesController` are both examples of conventional controller names.

The first method you write for a controller might be the `index()` method. When a request specifies a controller but not an action, the default CakePHP behavior is to execute the `index()` method of that controller. For example, a request for http://www.example.com/apples/ maps to a call on the `index()` method of the `ApplesController`, whereas http://www.example.com/apples/view/ maps to a call on the `view()` method of the `ApplesController`.

You can also change the visibility of controller methods in CakePHP by prefixing controller method names with underscores. If a controller method has been prefixed with an underscore, the method will not be accessible directly from the web but is available for internal use. For example:

```
class NewsController extends AppController {

    public function latest() {
        $this->_findNewArticles();
    }

    protected function _findNewArticles() {
        // Logic to find latest news articles
    }
}
```

While the page http://www.example.com/news/latest/ would be accessible to the user as usual, someone trying to get to the page http://www.example.com/news/_findNewArticles/ would get an error, because the method is preceded with an underscore. You can also use PHP's visibility keywords to indicate whether or not a method can be accessed from a url. Non-public methods cannot be accessed.

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

Multiple word controllers *can* be any 'inflected' form which equals the controller name so:

- /redApples
- /RedApples
- /Red_apples
- /red_apples

will all resolve to the index of the RedApples controller. However, the convention is that your urls are lowercase and underscored, therefore /red_apples/go_pick is the correct form to access the `RedApplesController::go_pick` action.

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

File and Classname Conventions

In general, filenames match the classnames, which are CamelCased. So if you have a class **MyNiftyClass**, then in Cake, the file should be named **MyNiftyClass.php**. Below are examples of how to name the file for each of the different types of classes you would typically use in a CakePHP application:

- The Controller class **KissesAndHugsController** would be found in a file named **KissesAndHugsController.php**
- The Component class **MyHandyComponent** would be found in a file named **MyHandyComponent.php**
- The Model 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 in your app folder.

Model and Database Conventions

Model classnames are singular and CamelCased. Person, BigPerson, and ReallyBigPerson 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 `people`, `big_people`, and `really_big_people`, respectively.

You can use the utility library `Inflector` to check the singular/plural of words. See the *Inflector* for more information.

Field names with two or more words are underscored like, first_name.

Foreign keys in hasMany, belongsTo or hasOne relationships are recognized by default as the (singular) name of the related table followed by _id. So if a Baker hasMany Cake, the cakes table will refer to the bakers table via a baker_id foreign key. For a multiple worded table like category_types, the foreign key would be category_type_id.

Join tables, used in hasAndBelongsToMany (HABTM) relationships between models should be named after the model tables they will join in alphabetical order (apples_zebras rather than zebras_apples).

All tables with which CakePHP models interact (with the exception of join tables), require a singular primary key to uniquely identify each row. If you wish to model a table which does not have a single-field primary key, CakePHP's convention is that a single-field primary key is added to the table. You have to add a single-field primary key if you want to use that table's model.

CakePHP does not support composite primary keys. If you want to directly manipulate your join table data, use direct *query* calls or add a primary key to act on it as a normal model. E.g.:

```
CREATE TABLE posts_tags (
id INT(10) NOT NULL AUTO_INCREMENT,
post_id INT(10) NOT NULL,
tag_id INT(10) NOT NULL,
PRIMARY KEY(id));
```

Rather than using an auto-increment key as the primary key, you may also use char(36). Cake will then use a unique 36 character uuid (String::uuid) whenever you save a new record using the Model::save method.

View Conventions

View template files are named after the controller functions they display, in an underscored form. The getReady() function of the PeopleController class will look for a view template in /app/View/People/get_ready.ctp.