The CakePHP Framework: Your First Bite

By Fabio Cevasco

July 12th 2006

Reader Rating: 8.8

Fabio Cevasco

Fabio just started working as technical writer for Siemens Italia. He's also very fond of PHP programming and enjoys writing and blogging about it on his personal web site, H3RALD.com.

Illustration by: Matthew Magain

According to a recent study [1], PHP [2] is one of the most popular programming languages in the world. In spite of this, PHP is often criticized for its inconsistent naming conventions, its lack of important features as compared to other languages (like namespaces) and its inherent disorganization. Furthermore, PHP is very easy to learn, and this has often led to the common misconception that most PHP developers are inexperienced and that their code is therefore prone to security vulnerabilities and exploits [3].

This is all true, to a certain extent. PHP itself offers virtually no real structure or organization, and thereby leaves coders free to express themselves in the most unpredictable and dangerous ways: programming logic mixed with presentation elements, disorganized inclusion of other source files anywhere in a script, unnecessary and often forgotten database connections, and so on. These are obvious and common mistakes that can make PHP code completely unmaintainable.

PHP Needs a Framework

In recent years, PHP has re-invented itself, allowing Object Oriented Programming (OOP [4]) to enter the scene with a plethora of new rules and functionality, all of which are ingrained in more mainstream programming languages like C++ and Java [5]. Gradually, more and more PHP developers have embraced this new philosophy and started developing frameworks, drawing their inspiration from other more-established languages in the pursuit of creating a structure for an inherently unstructured language.

Many frameworks are available on the Internet, each with its own specific set of rules and conventions, achievements and failures. Some degenerate into unusable and intricate collections of pre-built libraries and tools that enslave developers into complex and truly unusable programming methodologies; others do not.

Ruby on Rails has definitely played a key role in inspiring the quest for the perfect web framework in programming languages other than Ruby. Thanks to the Rails phenomenon, more frameworks have appeared on the scene, offering functionality that's very similar to Ruby on Rails. These frameworks are often labeled Rails Clones [6].

Some of the frameworks' developers have openly admitted that they tried to port Rails to other languages, but often they overlook the fact that Ruby on Rails was built in Ruby for a reason: Ruby has features that no other programming language offers. At the same time, at least one person gave up on the idea of totally cloning Rails in PHP, but instead, decided to borrow its structure and basic concepts to make PHP more organized:

> “While it's difficult to copy Rails in PHP, it's quite possible to write an equivalent system. I like the terseness of Ruby code, but I need the structure that Rails provides, how it makes me organize my code into something sustainable. That's why I'm ripping off Rails in Cake.”

- CakePHP's founder, commenting on a famous blog post [7].

This is what makes CakePHP [8] not only different, but one of the most popular frameworks for PHP: its modest, yet important goal is to provide an appropriate structure for PHP applications.

CakePHP's Approach to the MVC Architecture

Readers who already know Ruby on Rails may find CakePHP very similar to it. For one thing, Cake is based on an MVC-like architecture that is both powerful and easy to grasp: controllers, models and views guarantee a strict but natural separation of business logic from data and presentation layers.

Controllers contain the logic of your application. Each controller can offer different functionality; controllers retrieve and modify data by accessing database tables through models; and they register variables and objects, which can be used in views.

Models are active representations of database tables: they can connect to your database, query it (if instructed to do so by a controller) and save data to the database. It is important to note that in order to correctly apply the MVC architecture, there must be no interaction between models and views: all the logic is handled by controllers.

Views can be described as template files that present their content to the user: variables, arrays [9] and objects that are used in views are registered through a

controller. Views should not contain complex business logic; only the elementary control structures necessary to perform particular operations, such as the iteration of collected data through a foreach construct, should be contained within a view.

This architecture can greatly improve the maintainability and the organization of your site's code:

- It separates business logic from presentation and data retrieval.
- A site is divided into logical sections, each governed by a particular controller.
- When testing and debugging an application, any developer accustomed to CakePHP's structure will be able to locate and correct errors without knowing all of the details of the code.

Controllers, models and views are stored in pre-defined directories within CakePHP's directory structure. Here's the directory structure that's used:

- app/
    - config/
    - controllers/
    - models/
    - plugins/
    - tmp/
    - vendors/
    - views/
    - webroot/
- cake/
    - config/
    - docs/
    - libs/
- vendors/

This directory scheme must be preserved, as it is essential if the framework itself is to work. Cake, like Rails, believes in the importance of convention over configuration: in order to deploy an application, rather than modify dozens of different configuration files, it's important only to place everything in its proper place; then, you can let the framework do the rest.

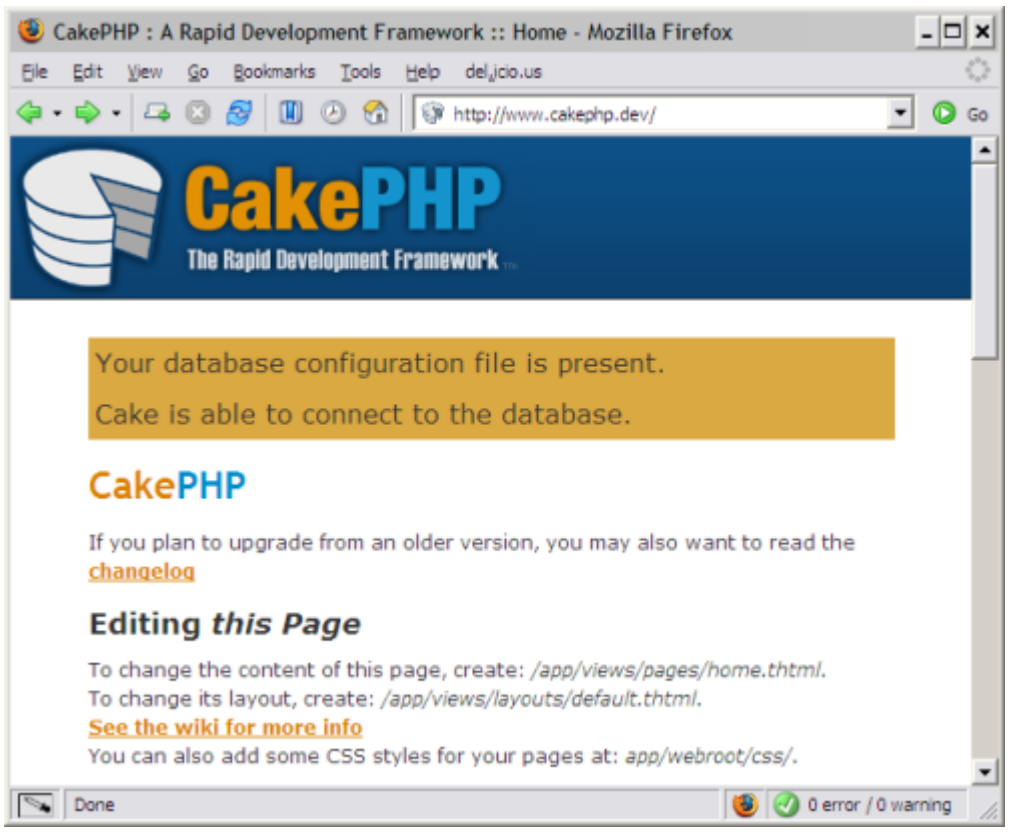
Although this may seem worrisome for some developers, it's a good compromise that can really accelerate the development process.

Tasting the Batter

As an example, let's now look at building a simple memo application that allows you to add, edit, display and delete personal notes, which are stored in a MySQL [10] database. In order to try this yourself, you'll need to download the latest stable version of CakePHP [11] and ensure that your development environment meets the following requirements:

- has access to a web server like Apache [12], although others like IIS and Lighttpd are supported
- has the ability to rewrite URLs (e.g. using the mod_rewrite module for Apache -- by default CakePHP comes with a .htaccess file to handle this)
- has MySQL [13], or a similar database server installed, and you have the necessary privileges to create a new database; other solutions like PostgreSQL or SQLite should work, although MySQL is recommended (this example will assume that you are using MySQL)
- has PHP version 4.3 [14] or above; CakePHP seamlessly supports both PHP4 and PHP5

After downloading the CakePHP package, extract its contents to the document root directory of your web server, or one of its subdirectories. For this example, I'll assume that your CakePHP application is installed in the document root directory and that it's accessible [15] at `http://localhost [16]/`.



Confirming configuration of CakePHP

If you now try to access your application, the CakePHP default page will be displayed, warning you that a database connection could not be established. We need to create a new MySQL database named memo that's accessible by a user named memouser, and a new table named notes:

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

Note how the table uses the plural notes. Now, edit your database configuration file (`/app/config/database.php.default`) as follows, and save it as `database.php` (yes: the name matters!):

```
<?php
class DATABASE_CONFIG
{
   var $default = array('driver'   => 'mysql',
                        'connect'  => 'mysql_pconnect',
                        'host'     => 'localhost',
                        'login'    => 'memouser',
                        'password' => 'userpassword',
                        'database' => 'memo' );
   var $test   =  array('driver'   => 'mysql',
                        'connect'  => 'mysql_pconnect',
                        'host'     => 'localhost',
                        'login'    => 'user',
                        'password' => 'password',
                        'database' => 'project_name-test');
}
?>
```

If you refresh the default page, CakePHP will notify you that the database is now accessible.

CakePHP is now configured properly, and we can turn to the development of our application. The framework offers a useful Rails-inspired feature called scaffolding, which basically allows the creation of an interface that's able to perform Create, Read, Update and Delete (CRUD) database operations with only a few lines of code. This is particularly useful when you want a particular area of your application to be available for testing purposes quickly, and you don't want to spend time coding it properly -- yet.

To create a scaffolded version of our memo application we need to create two very basic files: a controller and a model.

Create a file named `note.php` (again, the name matters -- notice how the file and the class defined here are the singular note of the database table notes) and save it in your `/app/models/` directory. You need only include the following lines in it:

```
<?php
class Note extends AppModel
{
```

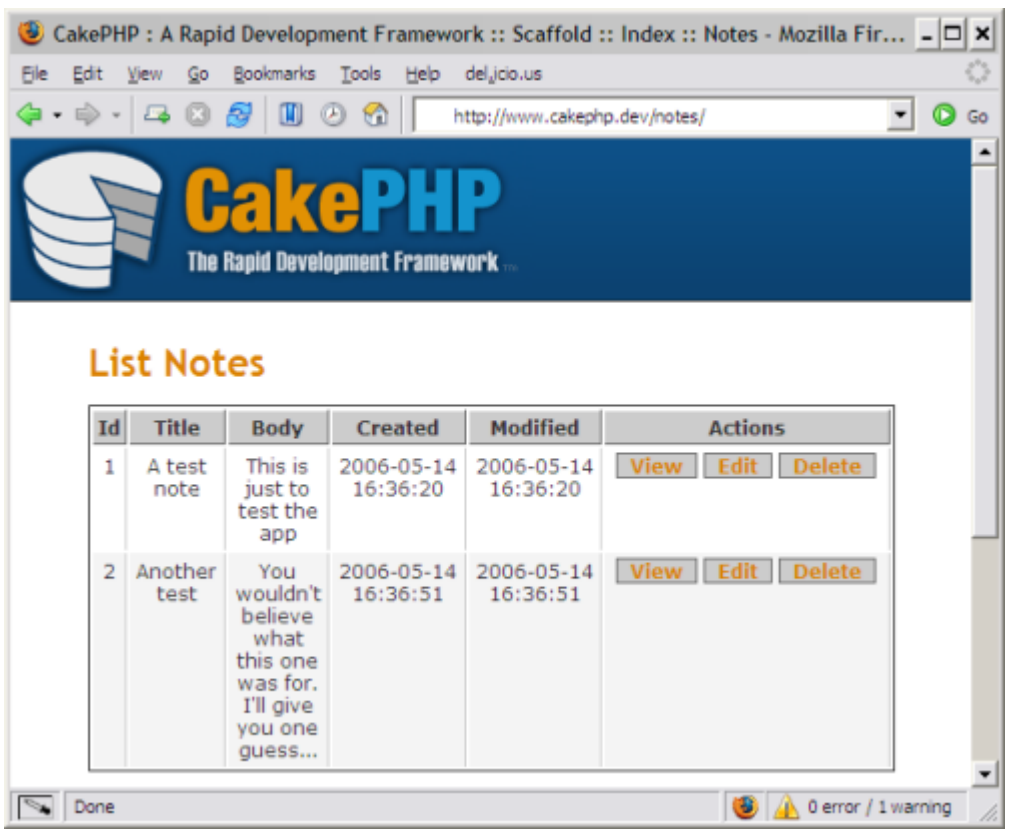
```
 var $name = 'Note';
}
?>
```

Similarly, create a `notes_controller.php` file containing the code below, and place it in `/app/controllers/`.

```
<?php
class NotesController extends AppController
{
var $name = 'Notes';
var $scaffold;
}
?>
```

The `$scaffold` variable will trigger CakePHP's default scaffolding behavior: a fully-functional Notes section will be created, and will be accessible at `http://localhost/notes/`.

That's all there is to it. You are now able to create, update, delete and display your notes with (literally) five lines of PHP code!



The default edit view for a CakePHP application

This is an old trick, and if you've ever read a beginners' tutorial to Ruby on Rails [17] you probably won't be too amazed; however, it's nice to know that a formerly Rails-only feature has been ported to PHP.

Creating your First Application

After playing with your new application for a while -- feel free to create and delete a few notes -- you'll start to notice its obvious limitations:

- the layout is very plain, and apparently is not customizable
- notes are deleted without confirmation
- there's no validation for any data input by users

We'll now remove our scaffolding and start to develop something that's slightly more advanced. If you paid attention to the previous example, you will notice that no view files were created. That's because Cake uses predefined templates for scaffolding; in reality, you'll need a view for almost every action listed in your controller.

Furthermore, our controller had no actions, and that is also part of the scaffold magic. A hint for the action names could be seen in the scaffolded application's URLs as we added and removed notes, namely:

```
http://localhost/notes/
http://localhost/notes/add/
http://localhost/notes/edit/1/
http://localhost/notes/view/2/
http://localhost/notes/delete/3/
```

In other words, all our URLs match a common pattern: they're all written in the form `/<controller>/<action>/<first_parameter>/`. So we need to create at least three views for the CRUD operations -- we'll name them `add.thtml`, `edit.thtml` and `view.thtml` -- as well as a default view (`index.thtml`) to list and manage all of the notes. The "t" in these `thtml` files indicates that these files are Cake templates. And what about `delete.thtml`? This file does not need to be created; we'll see why shortly.

Before proceeding, remove this line from your `NotesController` class:

```
var $scaffold;
```

Viewing your Notes

The first view we should create is a list of all the notes stored in the database, which will be the default page that displays when we access `http://localhost/notes/`. Create a new subdirectory named `notes` in your `/app/views/` directory, then create a new file named `index.thtml` inside that. This file should contain the following code:

```
<h1>My Notes</h1>
<table>
   <tr>
       <th>Id</th>
       <th>Title</th>
       <th>Created</th>
   </tr>
   <?php foreach ($notes as $note): ?>
   <tr>
       <td><?php echo $note['Note']['id']; ?></td>
       <td>
           <a href="/notes/view/<?php echo $note['Note']['id']?>">
               <?php echo $note['Note']['title']?>
           </a>
       </td>
       <td><?php echo $note['Note']['created']; ?></td>
 </tr>
   <?php endforeach; ?>
</table>
```

Note that our template code is not a complete HTML document -- things like the doctype and header information for all files is also provided by the framework, and the default can of course be overridden later.

This should display a list of all the stored notes, but if you try accessing `http://localhost/notes/` right now, you'll get an error saying that the action index is not defined in your controller.

The code for this action needs to be created in your controller. It simply needs to retrieve all records from your notes database table and store them in an array. Cake achieves this task in one line of code:

```
function index()
   {
         $this->set('notes', $this->Note->findAll());
   }
```

The method set is defined in Cake's `Controller` class, and is also inherited by `AppController`, `NotesController` and any other controller in your application. The purpose of set is to create a variable (`$notes`) that will be available in your default view (`index.thtml`), and its syntax is `$this->set(string $variable_name, mixed $value)`.

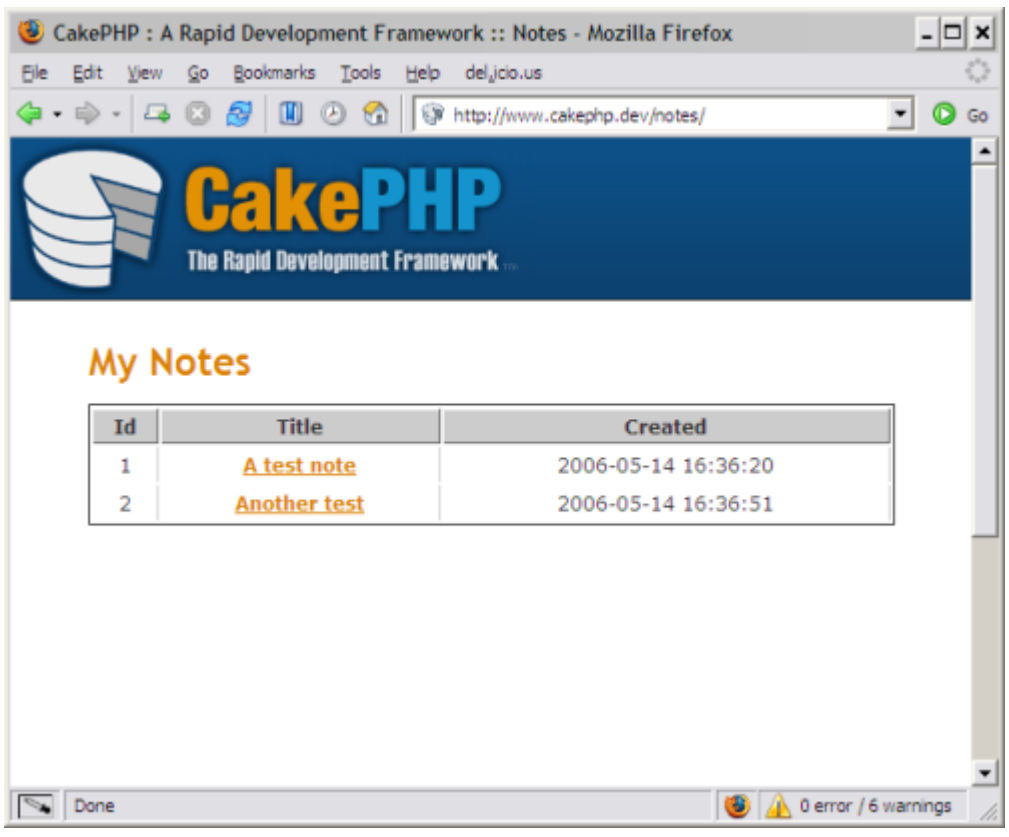
The value of the `$notes` variable is a multi-dimensional array returned by `$this->Note->findAll()`. `findAll` is a method defined in Cake's `Model` class, which fetches all records in the database table associated with the model. In this example, we'll access our Note model and call the method from our controller.

Assuming that your notes table has some records, the output of `findAll` will be something like this:

```
// print_r($notes) output:
Array
(
   [0] => Array
       (
           [Note] => Array
               (
                   [id] => 1
                   [title] => First note's title
                   [body] => Some text.
                   [created] => 2006-04-20 14:21:42
                   [modified] =>
               )
        )
```

```
    [1] => Array
        (
            [Note] => Array
                (
                    [id] => 2
                    [title] => Title...
                    [body] => body text
                    [created] => 2006-04-20 17:22:23
                    [modified] =>
                )
        )
)
```

As I mentioned before, this output is accomplished with only one line of code. CakePHP dramatically reduces the amount of repetitive and boring code required in your apps, thanks to its efficient built-in classes and intuitive conventions.



Creating our first view

We proceed similarly to view a single note. First, we need a `view.thtml` view file in our `/app/views/notes/` directory:

```
<h1><?php echo $data['Note']['title']?></h1>
<p><small>
Created: <?php echo $data['Note']['created']?>
</small></p>
<p><?php echo $data['Note']['body']?></p>
```

Then, we add the corresponding view action to our controller:

```
function view($id)
   {
       $this->Note->id = $id;
       $this->set('data', $this->Note->read());
   }
```

This method takes one parameter: the ID of the note we want to view ($id). In order to retrieve a particular note, we have to set the `$id` variable of our `Note` model to the `$id` parameter we passed to the method. Then we create a `$data` variable, which is available in our view via the set method. It contains an array returned by `$this->Note->read()`. `read` fetches only one row from our notes table, which corresponds to a particular `$id`.

Adding, Editing and Deleting Notes

Next, we'll create a view to add a new note. All we need is a file named `add.thtml` in the `/app/views/notes/` directory:

```
<h1>Add Note</h1>
<form action="<?php echo $html->url("/notes/add"); ?>" method="post">

   <p>
       Title:
       <?php echo $html->input('Note/title', array('size' => '40'))?>
   </p>
   <p>
```

```
        Body:
        <?php echo $html->textarea('Note/body') ?>
    </p>
    <p>
        <?php echo $html->submit('Save') ?>
    </p>
</form>
```

This code creates a basic form that allows users to enter a title and text for a note, and to save it. This time, I decided to use some convenience code to create the two input tags via the so-called HTML Helper. Helpers will be discussed in detail in the next section of this article, but to be brief, they are classes that are accessible from views, and they contain useful methods for formatting text, creating tags, adding Javascript [18] or AJAX [19] code, and so on. The HTML Helper is available by default in all views, and is used to create (X)HTML tags. I used it in this view to create an input tag, a textarea and a submit button. The syntax is relatively straightforward, but it's important to note that in order to map the input fields to our table columns easily, and thus automate the insertion process, the names of the input fields (usually the first parameter of each method of the HTML Helper) must be in the form `<model_name>/<table_field>`.

The `add` method for the Notes Controller can be something like this:

```
function add()
    {
    if (!empty($this->data['Note']))
        {
            if($this->Note->save($this->data['Note']))
            {
                $this->flash [20]('Your note has been updated.','/notes/');
            }
        }
    }
```

First of all we check whether or not the `$this->data` variable -- a sort of "optimized" version of the `$_POST` array -- is empty. If it contains something, that data is automatically saved in your notes table through the `$this->Note->save()` method call.

The flash method that's called afterwards will be familiar to anyone who has dabbled in Rails: it's used to keep small amounts of data in between requests, such as error messages or warnings; in this case it displays a temporary message for a few seconds, then redirects the user to `http://localhost/notes/`.

Note: The created and modified fields of our notes table are automatically populated with relevant data whenever a note is added or modified via the save method, so there's no need to keep track of those actions manually. Pretty useful, hey?

At this point you should notice that something is wrong. The `add.thtml` view and the add action described above are potentially very, very dangerous in their simplicity: there is no data validation whatsoever, so, at the moment, any kind of data entered by our users will be stored in our database without being filtered or checked. Cake has some built-in validation and input sanitizing mechanisms (which we'll examine briefly in the next section), but we'll keep things simple for now, as this is just a very elementary example to introduce CakePHP's basic features.

Editing a note is similar to adding a new one, the difference being that the edit form's values must already contain data.

`/app/views/notes/edit.thtml:`

```
<h1>Edit Note</h1>
<form action="<?php echo $html->url('/notes/edit')?>" method="post">
    <?php echo $html->hidden('Note/id'); ?>
    <p>
        Title:
        <?php echo $html->input('Note/title', array('size' => '40'))?>
    </p>
    <p>
        Body:
        <?php echo $html->textarea('Note/body') ?>
    </p>
    <p>
        <?php echo $html->submit('Save') ?>
    </p>
</form>

/app/controllers/notes_controller.php:
function edit($id = null)
{
    if (empty($this->data['Note']))
    {
        $this->Note->id = $id;
```

```
        $this->data = $this->Note->read();
    }
    else
    {
        if($this->Note->save($this->data['Note']))
        {
             $this->flash('Your note has been updated.','/notes/');
        }
    }
}
```

In this case, if no data is submitted, the values from the record we want to edit are retrieved and displayed in the view. Otherwise, if data is submitted, the record is updated via the save method as usual. Again, there are some obvious limitations to this simple function:

- We do not validate, filter or check the `$id` parameter (in reality, we should make sure that the `$id` is numeric and that it actually exists).
- Submitted data is not validated or filtered.
- No error handling occurs -- if something goes wrong, the user will never receive a warning message.
- Finally, in order to delete a note, all we need to do is create a delete action in our `NotesController`; no view file is necessary, since users will be redirected to the index page, where a message will be displayed.

```
/app/controllers/notes_controller.php:
function delete($id)
    {
        if ($this->Note->del($id))
        {
            $this->flash('The note with id: '.$id.' has been deleted.', '/notes');
        }
    }
```

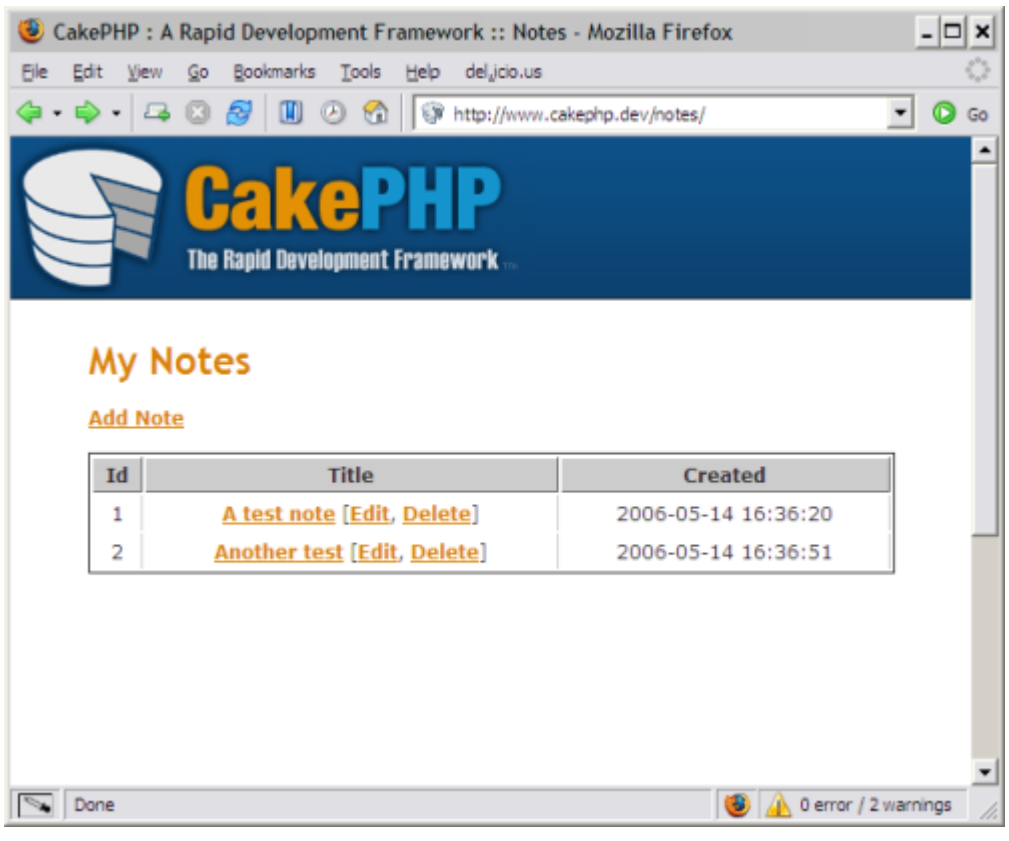
After defining all of our CRUD operations, we can make the interface easier to use by adding some convenient links for adding, editing and deleting notes. We can also rewrite our `index.thtml` view using the HTML Helper:

```
<h1>My Notes</h1>
<p>
<?php echo $html->link('Add Note', '/notes/add') ?>
</p>
<table>
    <tr>
        <th>Id</th>
        <th>Title</th>
        <th>Created</th>
    </tr>
    <?php foreach ($notes as $note): ?>
    <tr>
        <td><?php echo $note['Note']['id']; ?></td>
        <td>
      <?php echo $html->link($note['Note']['title'], "/notes/view/{$note['Note']['id']}")?>
      [<?php echo $html->link('Edit', "/notes/edit/{$note['Note']['id']}")?>,
      <?php echo $html->link('Delete', "/notes/delete/{$note['Note']['id']}", null, 'Are you sure?')?>]
        </td>

        <td><?php echo $note['Note']['created']; ?></td>
    </tr>
    <?php endforeach; ?>
</table>
```

In this example, I used the `$html->link()` method call, which is able to easily create "Cake-friendly" links. It can take up to six parameters:

- the text of the link
- the internal URL
- an array of HTML attributes (if any)
- text for a Javascript confirmation message
- whether we want to convert special characters in the title to HTML entities
- whether this method should either return or output a value `link($title`, `$url=null`, `$htmlAttributes=null`, `$confirmMessage=false`, `$escapeTitle=true`, `$return=false)`



The customized index page

The complete controller should look like this:

```
<?php
class NotesController extends AppController
{
 var $name = 'Notes';
 function index()
   {
         $this->set('notes', $this->Note->findAll());
   }
 function view($id)
   {
       $this->Note->id = $id;
       $this->set('data', $this->Note->read());
   }
 function add()
   {
   if (!empty($this->data['Note']))
       {
           if($this->Note->save($this->data['Note']))
           {
                $this->flash('Your note has been updated.','/notes/');
           }
       }
   }
```

```
 function edit($id = null)
{
   if (empty($this->data['Note']))
   {
       $this->Note->id = $id;
       $this->data = $this->Note->read();
   }
   else
   {
       if($this->Note->save($this->data['Note']))
       {
            $this->flash('Your note has been updated.','/notes/');
       }
   }
}


 function delete($id)
   {
       if ($this->Note->del($id))
```

```
        {
            $this->flash('The note with id: '.$id.' has been deleted.', '/notes');
        }
    }
}
?>
```

Not too difficult, is it? Granted, if you're not accustomed to the MVC pattern, this might all seem a bit strange, but our PHP code definitely looks much more organized and it's much easier to maintain than most unstructured PHP architectures.

One thing to keep in mind is that all those little conventions used in Cake actually matter: for example, the name of the controller must be plural and the model must be singular, while database tables should be plural (CakePHP's Inflector class does the rest), views must be placed in a folder named after the controller, and so on. Yes, you can get around some of these conventions, but it is precisely these details that make Cake virtually self-configuring: it's a case of convention over configuration, exactly like Rails. CakePHP may not be not the best solution for everybody, but it's certainly a simple and intuitive way to solve many of the problems associated with web development.

At this point, you probably have a lot of questions. For example, I wrote that CakePHP has a native validation mechanism and it can sanitize data. What does that mean? Why didn't we modify our model class? We'll answer these and other questions in the next section.

FAQs about CakePHP's Additional Features

CakePHP offers a lot of features that cannot properly be described in a single article. However, I've included a shortlist of frequently asked questions that may help you to understand this framework further.

1. How can I make my application more secure?

The examples in this article are inherently insecure. Luckily, CakePHP comes with a Sanitize class, which can be used in Cake applications to filter strings or arrays to make them safe for display or insertion into the database.
More information about sanitizing can be found in the CakePHP manual [21].

Regarding validation, it's possible to make sure that the entered data satisfies particular rules or patterns by adding some validation rules to our model, like this:

```
<?php
class Note extends AppModel
{
   var $name = 'Note';
   var $validate = array(
       'title'  => VALID_NOT_EMPTY,
       'body'   => VALID_NOT_EMPTY
   );
}
?>
```

`VALID_NOT_EMPTY` is a constant defined in `/cake/libs/validators.php`, and can be used to make sure that a particular field is not left blank. CakePHP comes with some predefined constants, but custom constants can be created.

After you define validation rules, all relevant actions and views should be modified accordingly. More information and examples are available in these pages of the manual [22].

2. Is there any way to turn off Cake's 'debugging mode'? Is there a main configuration file?

Yes. A main configuration file, which governs some of CakePHP's core settings, is located in `/app/config/core.php`. Some of the settings that can be modified via this file include:

- CakePHP's debugging verbosity and type
- logging level
- cookies [23] and session duration
- session storage location

3. All the business logic should go in my controllers, but what if I want to re-use something elsewhere?

Good question. You will almost always have to create some complex logic for an application, and you usually want to re-use part of that logic. The most common way to include an application-wide function or variable so that it's available in every controller is to define it in your AppController file. This file basically consists of an empty class that extends Cake's internal Controller class, and is located in the /cake/ directory. You can move it to your `/app/` directory and create methods that will be available in all of your custom controllers that extend AppController. Even if you're not planning to use an AppController at first, it's often wise to create custom controllers which extend AppController rather than the `Controller` class.

An easy way to create custom classes handling a specific task is to create a component. Components can be loaded automatically in controllers (and only inside controllers) by adding a variable named `$components`:

```
var $components = array('Session', 'MyCustomComponent');
```

CakePHP comes with some default components such as Session, which offers convenient ways to organize session data, or RequestHandler, which can be used to determine more information about HTTP requests. These are documented in the CakePHP manual:

- Session component manual pages [24]
- Request Handler component manual pages [25]

4. Does CakePHP require PHP5?

No. CakePHP is 100% compatible with PHP4. Personally, I think this is one of Cake's main strengths. For example, the `__construct()` method can be used on PHP4 on all classes extending the `Object` core class, which is to say nearly everything in CakePHP. Similar patches have been included in the core libraries to offer additional functionality in PHP4 as well. Unfortunately, variables and methods don't support access modifiers, and a private method should be prefixed with an underscore. This is not just a convention: in a controller, it really means that the method is private. If someone tries to access it (e.g. via `http://localhost/notes/_privatemethod/`), Cake will return an error.

5. What are CakePHP's default helpers?

CakePHP comes with some very handy helpers that can really make your life easier when it comes to creating views:

- HTML -- allows quick creation of HTML tags, including links and input fields
- JavaScript -- offers an easy way to manage JavaScript code
- Number -- a set of useful methods to format numeric data
- Time -- functions to format time strings and timestamps
- Text -- auto-link URLs, truncate strings, create excerpts, highlight, strip links and more
- AJAX -- a truly amazing AJAX helper, to be used in conjunction with the popular Prototype and script.aculo.us libraries; this helper can really speed up the creation of AJAX interfaces

More information about helpers is available in the CakePHP manual [26].

6. Is there any way to include my custom function/class in Cake?

Sure there is. If you want to use a custom external class, you can put it in the `/vendors/` directory and load it into your controller like this:

```
vendors('MyClassName');
```

If you need to define custom application-wide constants or functions, you can place them in `/app/config/bootstrap.php`, which will make them available everywhere in your application.

You can adapt your code and create a helper or a component to be used in conjunction with views or controllers.

You can also try to integrate other software packages into Cake. An example? Check out the CakeAMFPHP project [27].

7. What if I need to work with more than one table simultaneously?

By default, a NotesController will try to locate and load a `Note` model class. If your controller needs to access more than its default model, you can define additional models by setting the `$uses` array, like this:

```
var $uses = array(Note, AnotherModel, YetAnotherModel);
```

In some cases, two or more tables might be closely related and would therefore be used with JOIN statements: your notes may have been submitted by different people listed in an authors table, for example. In these cases, CakePHP's Associations can be used to define complex table relationships directly in your `Model` class. More information is available in these manual pages [28].

8. Is it possible to further customize my application's URLs?

Yes. Check out the `/app/config/routes.php` file, and feel free to define or modify your custom routes. For example:

```
$Route->connect ('/', array('controller'=>'notes', 'action'=>'index'));
```

This creates a default route for `http://localhost/` to:

```
http://localhost/notes/index/.
```

9. Is there an authentication mechanism in Cake?

Yes and no. There's no official authentication component, simply because needs can be very different depending on the type of application being developed. There is, however, a built-in Access Control List mechanism involving flat files or databases. More information can be found in these manual pages [29].

CakePHP Resources

The CakePHP Project is continuously growing: as more and more users start using the framework and creating their own projects, the documentation continues to improve. As such, more and more web sites and blogs are developing a lot of useful information that they're making freely available to CakePHP "bakers". Here's a shortlist of various places featuring Cake-related material:

The official CakePHP site [30]

CakePHP Wiki [31] -- a community-powered wiki with various Cake tutorials and how-tos

The CakePHP Manual [32] -- CakePHP's official manual, which is still a work in progress, but already is fairly comprehensive

CakePHP Google user group [33] -- a very lively user group; if you have a question to ask, go here

Official CakePHP IRC channel: #cakephp on irc.freenode.net -- chat with other bakers, as well as CakePHP's creators, in real time

CakeForge [34] -- the perfect place to host and share your open source CakePHP-related projects

Documentation [35] for offline use

Summary

CakePHP is a mature framework for PHP developers who want the structure and time-saving benefits of Ruby on Rails, without having to leave their comfort zone or get their head around obscure Ruby syntax. Using Cake's scaffolding, it's possible to build a prototype application quickly, using a minimal amount of code. And, with a large number of helper classes available to extend and customize your application while retaining a sensible and easily maintainable architecture, Cake makes the possibilities endless. CakePHP is being actively developed, and is backed by extensive documentation and a lively support community.

This article has given you a taste of what's possible with CakePHP. Now it's time for you to go off and do a little baking of your own!

Back to SitePoint.com

[1] http://www.tiobe.com/tpci.htm
[2] /glossary.php?q=P#term_1
[3] http://www.sitepoint.com/article/php-security-blunders
[4] /glossary.php?q=O#term_10
[5] /glossary.php?q=J#term_65
[6] http://redhanded.hobix.com/cult/railsClonesBloodsuckersOrUsefulDrones.html
[7] http://redhanded.hobix.com/cult/railsClonesBloodsuckersOrUsefulDrones.html
[8] http://www.cakephp.org/
[9] /glossary.php?q=%23#term_72
[10] /glossary.php?q=M#term_12
[11] http://cakeforge.org/projects/cakephp/
[12] http://httpd.apache.org/
[13] http://www.mysql.com/
[14] http://www.php.net/
[15] /glossary.php?q=A#term_61
[16] /glossary.php?q=L#term_42
[17] http://www.sitepoint.com/article/ruby-on-rails
[18] /glossary.php?q=J#term_9
[19] /glossary.php?q=A#term_73
[20] /glossary.php?q=F#term_16
[21] http://manual.cakephp.org/pages/ch13s01
[22] http://manual.cakephp.org/pages/ch11s01
[23] /glossary.php?q=C#term_59
[24] http://manual.cakephp.org/pages/ch14
[25] http://manual.cakephp.org/pages/ch15
[26] http://manual.cakephp.org/pages/ch09s01
[27] http://cakeforge.org/projects/cakeamfphp/
[28] http://manual.cakephp.org/pages/ch06s03
[29] http://manual.cakephp.org/pages/ch12
[30] http://www.cakephp.org/
[31] http://wiki.cakephp.org/
[32] http://manual.cakephp.org/
[33] http://groups.google.com/group/cake-php
[34] http://www.cakeforge.org
[35] http://cakeforge.org/projects/cakedocs/