

# **Super Awesome Advanced CakePHP Tips**

**Matt Curry**

# Super Awesome Advanced CakePHP Tips

This book is available for free at <http://www.pseudocoder.com/free-cakephp-book>

Version 1.1

By Matt Curry  
[pseudocoder.com](http://pseudocoder.com)  
[twitter.com/mcurry](https://twitter.com/mcurry)  
[matt@pseudocoder.com](mailto:matt@pseudocoder.com)

With Contributions From  
Mark Story  
[mark-story.com](http://mark-story.com)  
[twitter.com/mark\\_story](https://twitter.com/mark_story)



Super Awesome Advanced CakePHP Tips by [Matt Curry](#) is licensed under a [Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States License](#).

CakePHP is a registered trademark of the  
Cake Software Foundation (<http://cakefoundation.org>)

# Contents

<a href="#"><u>Who Should Read This Book</u></a>	6
<a href="#"><u>How to Read This Book</u></a>	6
<a href="#"><u>How to Learn CakePHP</u></a>	7
<a href="#"><u>The Paid Way</u></a>	7
<a href="#"><u>The Free Way</u></a>	8
<a href="#"><u>After you've done Either/Both Of The Above</u></a>	8
<a href="#"><u>Models</u></a>	9
<a href="#"><u>Recursion</u></a>	9
<a href="#"><u>Containable Behavior</u></a>	9
<a href="#"><u>Why You Should Use It</u></a>	9
<a href="#"><u>This Will Catch Everyone At Least Once</u></a>	10
<a href="#"><u>Custom Find Types</u></a>	11
<a href="#"><u>The Unofficial Cake Way</u></a>	11
<a href="#"><u>My Way</u></a>	12
<a href="#"><u>Comparison</u></a>	14
<a href="#"><u>App Model</u></a>	15
<a href="#"><u>Getting the Logged In User from Anywhere</u></a>	16
<a href="#"><u>The User Model</u></a>	16
<a href="#"><u>In The AppController</u></a>	17
<a href="#"><u>Back to the User Model</u></a>	17
<a href="#"><u>Usage</u></a>	18
<a href="#"><u>What About The Configure Class?</u></a>	18
<a href="#"><u>Full Source</u></a>	19
<a href="#"><u>Automatically Tracking Created/Modified By</u></a>	20
<a href="#"><u>Database</u></a>	20
<a href="#"><u>Model Relations</u></a>	20
<a href="#"><u>Model beforeValidate Callback</u></a>	20
<a href="#"><u>Behavior beforeValidate Callback</u></a>	20
<a href="#"><u>The Full Behavior</u></a>	21
<a href="#"><u>Routing</u></a>	22
<a href="#"><u>Case Insensitive</u></a>	22
<a href="#"><u>Unit Testing</u></a>	23
<a href="#"><u>Views</u></a>	23

<a href="#">Setting Up The Files</a>	23
<a href="#">Setting Up The Test Class</a>	23
<a href="#">Standard Index View</a>	24
<a href="#">Creating The View Test</a>	25
<a href="#">Testing the Rendered View</a>	26
<a href="#">Controllers</a>	28
<a href="#">Doing Things the Hard Way</a>	28
<a href="#">Testing A Controller Method</a>	30
<a href="#">Making assertions</a>	31
<b><a href="#">Mock Objects</a></b>	<b>33</b>
<a href="#">What is a Mock Object</a>	33
<a href="#">Where can I get one of these fabulous devices?</a>	33
<a href="#">Makings expectations with Mock Objects</a>	34
<b><a href="#">Models</a></b>	<b>36</b>
<a href="#">Test Case</a>	36
<a href="#">Fixtures</a>	36
<b><a href="#">Merging Add and Edit Actions</a></b>	<b>37</b>
<a href="#">The Controller</a>	37
<a href="#">Telling Add And Edit Apart</a>	38
<a href="#">The View</a>	39
<b><a href="#">Cake Tricks from The Core</a></b>	<b>40</b>
<a href="#">Cake Style \$options Parameter</a>	40
<a href="#">Handling Data Arrays with a Single Record or an Array of Records</a>	41
<b><a href="#">Stupid Easy URL Slugs</a></b>	<b>43</b>
<a href="#">Making The Slugs A Bit Smarter</a>	44
<a href="#">Changing The Underscore To A Hyphen</a>	44
<a href="#">Consistency And SEO Improvements</a>	44
<b><a href="#">jQuery</a></b>	<b>45</b>
<a href="#">Replacing \$javascript-&gt;event()</a>	45
<a href="#">Replacing \$ajax-&gt;link()</a>	45
<b><a href="#">Expanding Trees With jQuery</a></b>	<b>47</b>
<a href="#">Basic Tree</a>	47
<a href="#">TreeHelper</a>	47
<a href="#">TreeHelper With jQuery</a>	50
<a href="#">Cleaning Up the Images</a>	52
<b><a href="#">JavaScript In Views</a></b>	<b>53</b>
<b><a href="#">Make Your Cake App Fast</a></b>	<b>54</b>
<a href="#">Don't Use \$uses Unless You Really, Absolutely Have To</a>	54
<a href="#">Model Chains</a>	54
<a href="#">Controller::loadModel and ClassRegistry::init</a>	54
<a href="#">Use Containable</a>	55
<a href="#">Set Debug to 0</a>	55

<a href="#"><u>Cache your slow queries/web service requests/whatever</u></a>	55
<a href="#"><u>View Caching</u></a>	55
<a href="#"><u>HTML Caching</u></a>	56
<a href="#"><u>APC (or some other opcode cache)</u></a>	56
<a href="#"><u>Persistent Models</u></a>	56
<a href="#"><u>Store The Persistent Cache in APC</u></a>	57
<a href="#"><u>Speed Up Reverse Routing</u></a>	57
<a href="#"><u>Unchain Your Models</u></a>	57
<a href="#"><u>The Giant Configuration, Version Control and Deployment Section</u></a>	59
<a href="#"><u>Version Control</u></a>	59
<a href="#"><u>core.php</u></a>	59
<a href="#"><u>bootstrap.php</u></a>	59
<a href="#"><u>database.php</u></a>	60
<a href="#"><u>Multiple Environments</u></a>	61
<a href="#"><u>Deployment</u></a>	61
<a href="#"><u>Debug</u></a>	61
<a href="#"><u>Cache</u></a>	61
<a href="#"><u>Alternate Methods</u></a>	61
<a href="#"><u>CakePHP Reserved Classes</u></a>	62
<a href="#"><u>From The Bakery (And Other Places)</u></a>	63
<a href="#"><u>Behaviors</u></a>	63
<a href="#"><u>Sluggable</u></a>	63
<a href="#"><u>Soft Deletable</u></a>	63
<a href="#"><u>Linkable</u></a>	63
<a href="#"><u>Plugins</u></a>	63
<a href="#"><u>DebugKit</u></a>	63
<a href="#"><u>NamedScope</u></a>	63
<a href="#"><u>Helpers</u></a>	64
<a href="#"><u>Asset</u></a>	64
<a href="#"><u>Jquery Validation</u></a>	64
<a href="#"><u>HtmlCache</u></a>	64
<a href="#"><u>Copyright</u></a>	65
<a href="#"><u>Revisions</u></a>	66
<a href="#"><u>V1.0 – May 13, 2009</u></a>	66
<a href="#"><u>V1.1 – May 15, 2009</u></a>	66
<a href="#"><u>V1.2 – June 12, 2009</u></a>	66

# Who Should Read This Book

This book isn't meant for people wanting to learn CakePHP. There are already plenty of resources for that. If you're completely new to "Cake" (that's what we in-the-know call it) check out the section below, which outlines two paths for getting started.

I will make several assumptions about you, the reader, in this e-book.

- You have built a CakePHP app before or at least are familiar with the [blog tutorial](#).
- You want to improve your CakePHP skills. CakePHP alone does not magically make you write better code. I've seen plenty of shitty code in CakePHP apps.
- You understand that this book is free and I will probably mess up parts and other parts will become out of date. You will want to help me fix this by sending me an email at [matt@pseudocoder.com](mailto:matt@pseudocoder.com).

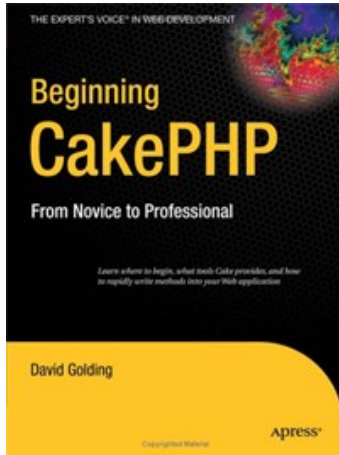
## How to Read This Book

This book doesn't have to be read cover-to-cover. The information, generally, doesn't rely on what you've learned in previous sections. Feel free to jump around or skip directly to sections that interest you. Think of it as a series of blog posts in book form.

# How to Learn CakePHP

## The Paid Way

There are a couple CakePHP books out right now. The biggest caveat with all of them is that they were written before the stable release of CakePHP 1.2, so there may be discrepancies with the most recent version of the framework.



### [\*Beginning CakePHP: From Novice to Professional\*](#)

by David Golding

[Available at Amazon.com for \\$28.37](#)

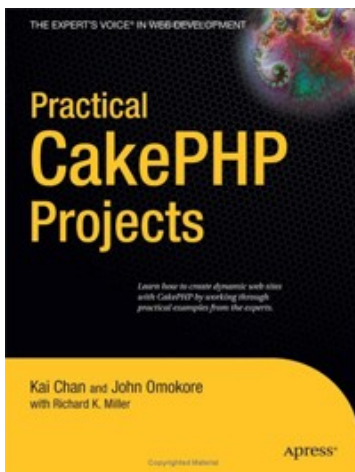


### [\*CakePHP Application Development\*](#)

Step-by-step introduction to rapid web development using the open-source MVC CakePHP

by Ahsanul Bari and Anupom Syam

[Available at Amazon.com for \\$35.99](#)



### [\*Practical CakePHP Projects\*](#)

by Kai Chan and John Omokore with Richard Miller

[Available at Amazon.com for \\$28.37](#)

## The Free Way

Go to [The CakePHP Cookbook](#) and read the following sections in this order:

1. Chapter 1
  - [The Intro](#)
  - [1.1 What is CakePHP? Why Use it?](#)
  - [1.3 Understanding Model-View-Controller](#)
  - [1.3.1 Benefits](#)
2. Chapter 2
  - [All of it, including any subsections and subsubsections](#)
3. Chapter 3
  - [3.1 Requirements](#)
  - [3.2 Installation Preparation](#)
  - [3.2.1 Getting CakePHP](#)
  - [3.2.2 Permissions](#)
  - [3.3 Installation](#)
  - [3.3.1 Development](#)
4. Chapter 10.1
  - [Sample Blog App - Do The WHOLE Thing](#)

## After you've done Either/Both Of The Above

At this point you should have a pretty good understanding of the basics of CakePHP. Feel free to investigate the rest of the Cookbook, check out some [other resources](#) or finish this book.

Also the book "[Refactoring Legacy Applications Using CakePHP](#)" by Chris Hartjes is a great resource for only \$10 ([I reviewed it here](#)). The book provides great, practical examples showing how procedural, clusterfucked PHP code can be rewritten in the Cake Framework.

My recommendation would be come up with an app you want to build. It doesn't have to be anything great and can simply be a clone of something else that's already out there. Then build it. You'll encounter different roadblocks that will force you to learn more about the framework. At each roadblock take a look on the Cookbook or Google and see if there is a best way to accomplish what you want.

If you can't figure out why something isn't working try taking a look in the framework code. It can be intimidating at first, but it's well coded and you'll learn a lot more than if you just post a question and get a quick answer.

I can't recommend this highly enough: Get remote debugging working ([Google "xdebug"](#)). The ability to put break points in your code, then step through it line by line, with the power to inspect and change variables, is a revelation. It's like doing `echo 'here'; die;` on steroids, HGH and speed.



# Models

## Recursion

The [recursive property in CakePHP models](#) is used when determining how much information is returned with your queries. The problem is that it gives you only limited control. You can select the level of information you want, but if you have many related tables you will often retrieve more information than you need, which will result in wasted queries.

Here are some signs you have recursion set too high in your app:

- When you're looking at the outputted queries with debug on you've reached the cap (200) that will be displayed.
- Your page takes so long to load that you have enough time to benchmark all the major PHP frameworks and post about it on your blog.
- You're using a recursive level that isn't -1.

Now you're confused, right? The first thing I do with any new CakePHP app is create an AppModel and set `var $recursive = -1;`

How do you retrieve related tables? Read on...

## Containable Behavior

The containable behavior gives you precise control over what data is returned. This means you won't waste queries loading related tables that you may not care about.

The CakePHP Book [explains it pretty well](#), so I won't waste space going over the same material. Instead I'll emphasize just how important it is to use.

## Why You Should Use It

You may be thinking "My app is pretty small. Using Containable would be overkill." And maybe you're right. Maybe you just have a Post model that has a belongsTo association to a User model. It's easy enough for you to set \$recursive to -1, 0 or 1 as needed. So you do this. All over your app.

Then a few months later you add a Tag model and Stat model and who knows what else, but they're all related to Post. Now all of a sudden you're getting back all this extra information you don't need, when you really just want User.

If you use Containable you never have to worry about new model associations blowing up your existing queries.

## This Will Catch Everyone At Least Once

So you've set `$recursive` to `-1` in the `AppModel` and are happily containing your related models. You fire up a standard index with the built in paginator:

```
function index() {
    $this->Post->recursive = 0;
    $this->set('posts', $this->paginate());
}
```

Ugh. It has that ugly call to set recursive to 0. So you delete it, but you still have to load the `User` model to show who created each post. So you make the code look like this:

```
function index() {
    $this->Post->contain('User');
    $this->set('posts', $this->paginate());
}
```

When you refresh the page it's broken. The `Post` data loaded, but the associated `User` model wasn't. Looking closer you see that the count query was fine:

```
SELECT COUNT(*) AS `count`
FROM `posts` AS `Post`
LEFT JOIN `users` AS `User` ON (`Post`.`modified_user_id` = `User`.`id`)
WHERE 1 = 1
```

But the second query, the one that loads the actual data, is messed up - there is no join to the `users` table.

```
SELECT `Post`.`id`, `Post`.`title`, `Post`.`body`
FROM `posts` AS `Post`
WHERE 1 = 1
LIMIT 20
```

The issue is that even though you are containing the `User` model, it is being reset after the first query, the count query. So when the second query runs, the one to get the actual data, it doesn't know anything about the binding. There is a simple fix for this, and [it's mentioned in the manual](#), but it's going to come up if you force recursive to always be `-1`, so it's worth stating again.

You can set the controller's `paginate` options to contain any related models you want before you call `$this->paginate()`.

```
function index() {
    $this->paginate = array('contain' => 'User');
    $this->set('posts', $this->paginate());
}
```

# Custom Find Types

You've been writing some pretty boring apps if you haven't had to use the [built in CakePHP find types](#) to retrieve some data. Either that or you've just been using `$Model->query` and writing custom SQL all the time (I've seen this). Or you've been making your own database connection with the standard PHP MySQL functions and used `mysql_query` (yes, I've seen this in a CakePHP app - serious facepalm).

You're probably familiar with the types "first" and "all" at a bare minimum. Wouldn't it be cool to create your own types? Like if you had a model Post and wanted to do `$Post->find('latest')` or `$Post->find('tag', array('php', 'beer', 'nsfw'))`.

You'll find a bunch of different ways to do this around the web. I'll explain two different ways to accomplish this. Personally I like my method described below (the second one). You could probably make an argument that some of the other ways are better, but if I don't get my way I'll take my e-book and go home.

## The Unofficial Cake Way

Cake's base Model class has an attribute defined that keeps track of all the default find types. It looks like this:

```
var $_findMethods = array(
    'all' => true, 'first' => true, 'count' => true,
    'neighbors' => true, 'list' => true, 'threaded' => true
);
```

To create your custom find type you need to add it to this list. If your custom find type was "latest" you would do something like this:

```
class PostModel extends AppModel {
    $this->_findMethods = array('latest' => true);
}
```

Now you have to create the method that will be called. The method name must match the pattern `_findCustomType`. In the example about it would be `_findLatest`. The tricky part is that this method will be called twice. Once before the database query is made and once after.

```
function _findLatest($state, $query, $results = array()) {
    if ($state == 'before') {
        $query['limit'] = 10;
        $query['order'] = 'created DESC';
        return $query;
    } elseif ($state == 'after') {
        return $results;
    }
}
```

In the before state you will have access to the `$query` array, where you can set conditions, limit, order...all the usual find options. In the after state the `$results` array is passed. You can then alter `$results` to fit whatever you're trying to do or just return it directly.

## My Way

My way requires a bit more initial setup, but really it's just a matter of copying and pasting some code to your AppModel and you don't have to add anything for each new find method.

I'll step through what it does, then include the entire block at the end.

First in your AppModel you'll need to override the find function.

```
function find($type, $options = array()) {  
  
}
```

In the actual model class (not AppModel) we will create methods for each find type that uses the `$type` parameter as a piece of the method name.

```
$method = sprintf('__find%s', Inflector::camelize($type));
```

For example calling `$Post->find('latest')` will look for the method `"__findLatest"` in the Post model class. If you can't come up with snappy names for your find types and end up with `$Post->find('latest_with_comments')` the private method will be `"__findLatestWithComments"`.

Then it is simply a matter of checking if the method exists and calling it. Or if it doesn't exist, calling the parent.

```
if(method_exists($this, $method)) {  
    return $this->{$method}($options);  
} else {  
    return parent::find($type, $options);  
}
```

There is one catch. Cake will sometimes call find internally and pass an array as the `$type` parameter. To catch that simply check if `$type` is a string before setting `$method`.

## The whole thing:

([also available as a plugin on GitHub](#))

```
function find($type, $options = array()) {
    $method = null;
    if(is_string($type)) {
        $method = sprintf('__find%s', Inflector::camelize($type));
    }

    if($method && method_exists($this, $method)) {
        return $this->{$method}($options);
    } else {
        $args = func_get_args();
        return call_user_func_array(array('parent', 'find'), $args);
    }
}
```

Now you can create a method in your Post model like:

```
function __findLatest($options) {
    $options = am(array('conditions' => array('published' => true),
        'order' => array('created' => 'desc'),
        'limit' => 10
    ), $options
    );
    return parent::find('all', $options);
}
```

Then you can simply call:

```
$Post->find('latest');
```

## Comparison

There are advantages and disadvantages to each method.

The Unofficial Cake Way:

- + Less initial code
- + Access to the before and after states
- Have to handle before and after states

My Way:

- + Don't have to override the Model `__constructor` and setup each custom find type. Just write the method.
- Some initial setup.
- + Don't have to handle before and after states.

I'll explain that last one. With My Way you have access to both states, but don't have to explicitly handle them. For example what if your custom find type was really just a wrapper to one of the default find types. Let's say you wanted to get a count of comments on your blog and double it so you looked popular. With My Way you'd simply do:

```
function __findCommentCount($options) {  
    return $this->find('count')) * 2;  
}
```

Doesn't get much simpler than that. Here's the same thing with the Unofficial Cake Way:

```
function _findCommentCount($state, $query, $results = array()) {  
    if ($state == 'before') {  
        return $this->_findCount($state, $query, $results);  
    } else {  
        return $results * 2;  
    }  
}
```

## App Model

You can pull together all of the above to create a base AppModel suitable for any new app you start.

```
<?php
class AppModel extends Model {
    var $actsAs = array('Containable');
    var $recursive = -1;

    function find($type, $options = array()) {
        $method = null;
        if(is_string($type)) {
            $method = sprintf('__find%s', Inflector::camelize($type));
        }

        if($method && method_exists($this, $method)) {
            return $this->{$method}($options);
        } else {
            $args = func_get_args();
            return call_user_func_array(array('parent', 'find'), $args);
        }
    }
}
?>
```

# Getting the Logged In User from Anywhere

This is a problem that comes up often. Generally you need access to the session to retrieve information about the logged in user. In the view this is done with the session helper.

```
$session->read('Auth.User.id');
```

In the controller you can use the SessionComponent.

```
$this->Session->read('Auth.User.id');
```

But what if you have a model where you need to set a modified user field every time you save? It would be great if you could handle this automatically in the beforeFilter. You can always cheat and use the `$_SESSION` superglobal, but that's not very Cakey. Plus, now we are using three different methods depending on the part of the app we're in.

How awesome would it be if you could just call `User::get('id')` from anywhere? On a scale of 0%-100%, that's like 82% awesome, right?

## The User Model

Here's how to do it. First you need a User model, which you probably already have. This code is a little misplaced among the rest of the User model code, but for the `User::get` syntax to work it has to go in the model. Deal with it.

The first thing you'll need is a way to get a singleton instance of your user instance.

```
function &getInstance($user=null) {
    static $instance = array();
    if ($user) {
        $instance[0] =& $user;
    }
    if (!$instance) {
        trigger_error(__("User not set.", true), E_USER_WARNING);
        return false;
    }
    return $instance[0];
}
```

This method will be used internally to store and retrieve the user's information. Don't worry if you don't understand what's going on here. Just trust that it works.



Before you can access the user info you need to store the user in the static instance. The method "set" is already taken by Cake's base Model class so we'll use "store". The store method is very simple:

```
function store($user) {
    User::getInstance($user);
}
```

## In The AppController

The User::store method is used in the AppController BeforeFilter.

```
App::import('Model', 'User');
User::store($this->Auth->user());
```

The logged in user is pulled from the AuthComponent and passed as a parameter to the store method. If you're not using the AuthComponent you can substitute whatever half-assed, cobbled-together authentication system you're using. You just need to pass in the user information as an array.

## Back to the User Model

The User::get method can then be used to get the instance and pull out a specific piece of information from the user array.

```
function get($path) {
    $_user =& User::getInstance();

    $path = str_replace('.', '/', $path);
    if (strpos($path, 'User') !== 0) {
        $path = sprintf('User/%s', $path);
    }

    if (strpos($path, '/') !== 0) {
        $path = sprintf('/%s', $path);
    }

    $value = Set::extract($path, $_user);
    if (!$value) {
        return false;
    }

    return $value[0];
}
```

## Usage

To get the currently logged in user's id:

```
User::get('id');
```

To get the currently logged in user's username (assuming you have a field "username" in your users table):

```
User::get('username');
```

Any other fields in your user table can be retrieved in the same manner.

Also if you store related model data in your user session it can be retrieved:

```
User::get('Model.fieldname');
```

## What About The Configure Class?

Some of this code will look familiar to those of you who have dug around in the Cake core. That's because it's ~~blatantly stolen~~ inspired by the [Configure class](#). Obviously, you could skip all the user setup and just use Configure like this:

```
//in the AppController
Configure::write('User', $this->Auth->user());
//from anywhere else
Configure::read('User.id');
```

It's not awful, but the syntax isn't as appealing. But, wait. Why not just use the same setup for User, but have it wrap calls to Configure. Then User doesn't have to bother getting the static instance of itself and you get the cooler `User::get()` syntax. Again, certainly an option. I wouldn't hold it against you if you did it that way. However sticking the logged in user in the Configure class is a bit misplaced. It's not "end of the world" bad to do it this way. More like "shit, I just dropped my iPod in the toilet" bad.

Really it's a personal decision that each developer will need to search deep within their programmer souls to find the solution that fits them best.

## Full Source

Here is the full source, which is also [available on GitHub](#).

```
function &getInstance($user=null) {
    static $instance = array();

    if ($user) {
        $instance[0] =& $user;
    }

    if (!$instance) {
        trigger_error(__("User not set.", true), E_USER_WARNING);
        return false;
    }

    return $instance[0];
}

function store($user) {
    if (empty($user)) {
        return false;
    }

    User::getInstance($user);
}

function get($path) {
    $_user =& User::getInstance();
    if (strpos($path, 'User') !== 0) {
        $path = sprintf('User/%s', $path);
    }

    if (strpos($path, '/') !== 0) {
        $path = sprintf('/%s', $path);
    }

    $value = Set::extract($path, $_user);
    if (!$value) {
        return false;
    }

    return $value[0];
}
```

# Automatically Tracking Created/Modified By

In the previous section we establish a method for retrieving the logged in user from anywhere in your application. Now we can apply this to setting the created by and modified by fields for various database records.

## Database

First off, add the fields "created\_user\_id" and "modified\_user\_id", both ints, to any database table you want to track changes on.

## Model Relations

Next you need to set up a belongsTo Association for each of the fields.

```
var $belongsTo = array('CreatedUser' => array('className' => 'User'),  
                        'ModifiedUser' => array('className' => 'User'));
```

## Model beforeValidate Callback

This is the point that most books would show you how to do a normal model beforeValidate callback, while you're thinking "What a friggin' idiot. He calls this e-book/e-pamphlet Super Awesome Advanced CakePHP Tips, but then doesn't know enough to make this into a behavior." Then in the next section the author would make it into a behavior and you would feel mildly foolish, but justified in believing the author should have just skipped right to the behavior method. So I won't insult your intelligence and skip to the right way to handle this.

## Behavior beforeValidate Callback

First let's deal with the created by scenario.

```
function beforeValidate(&$model) {  
  
    if(empty($model->data[$model->alias]['id'])) {  
        $model->data[$model->alias]['created_user_id'] = User::get('id');  
    }  
  
    return true;  
}
```

The logic here is pretty simple. We check if the "id" field is set to determine if this is a new record or an update. We're making a bit of a leap here, in believing that if the id is set it will be an update. It is certainly possible that an invalid id is passed. Cake handles this by checking if the id exists first. You'll usually see a query like this before an insert or update:

```
SELECT COUNT(*) AS `count` FROM `posts` AS `Post` WHERE `Post`.`id` = 1
```

So there is the outside possibility that the `created_user_id` field may not get set this way. If you wanted to be diligent you could move the logic to an `afterSave`, which is passed a `$created` boolean.

To set the `modified_user_id` you use basically the same code, less the check to see if this is a new record.

```
$model->data[$model->alias]['modified_user_id'] = User::get('id');
```

## The Full Behavior

```
<?php
class TrackableBehavior extends ModelBehavior {
    function beforeValidate(&$model) {
        if (empty($model->data[$model->alias]['id'])) {
            $model->data[$model->alias]['created_user_id'] = User::get('id');
        }

        $model->data[$model->alias]['modified_user_id'] = User::get('id');

        return true;
    }
}
?>
```

# Routing

## Case Insensitive

Generally Internet URLs are case insensitive. You can put the address <http://en.wikipedia.org/wiki/CakePHP> or <http://en.wikipedia.org/wiki/cakephp> in your browser and you'll end up at the same place. Routes in CakePHP are not case insensitive by default. Fortunately the routing system uses regular expressions, so it is easy to make a specific route work no matter how it is typed in.

If you want to have an “about” page that is linked to by “/about”, “/About”, or “/aBoUt” you can use:

```
Router::connect('/:?(i)about', array('controller' => 'pages',  
                                     'action' => 'display',  
                                     'about'));
```

The “(i)” part tells the regular expression engine to ignore the case for all the text that follows. There is no way to apply this rule universally to all your routes, so you'll have to handle instances individually.

# Unit Testing

## Views

The CakePHP Cookbook describes a [way to test views using web testing](#). This is different than unit testing in that you actually make a request to the web page and check the html response. By doing this there is no way to specify that the test database should be used, therefore any data that is saved will go in the “live” database. This may not be a big deal if the “live” database is just your development environment.

In addition, since you are testing the view by making a request, you aren’t isolating the view. The Cake framework, the controller and any models that are normally used in generating the output for the view will be used. This makes it hard to tell if an error is due to an issue with the view or from somewhere else.

The method described isolated testing to just the view. No models or controllers are needed as the view class is called directly. Because this method bypasses the controller, any helpers will need to be included manually and any data needed by the view will need to be faked.

### Setting Up The Files

To setup a view test you’ll first need to create a directory `/app/tests/cases/views`. In there put a file for each controller. The naming convention `<controller>_view.test.php` works well. In this example I’ll be testing the views for the Post controller/model, so I created a file `/app/tests/cases/views/post_view.test.php`.

The file itself is similar to the model or controller tests, in that it will extend `CakeTestCase`.

```
class PostViewTestCase extends CakeTestCase {  
}
```

### Setting Up The Test Class

The first thing needed is an instance of the View class. A good place to do this is in the `startTest` method.

```
function startTest() {  
    $Controller = new Controller();  
    $this->View = new View($Controller);  
    $this->View->layout = null;  
    $this->View->viewPath = 'posts';  
}
```

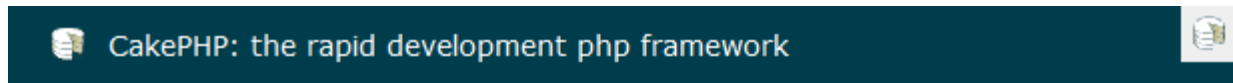
The View class expects a controller to be passed as a parameter to the constructor, so a base controller is created for this. The layout is set to null, which allows the view to be rendered without the default layout. Also the `viewPath` needs to be set to the folder in views that will be tested.

To be safe we'll be sure that the View was created successfully:

```
function testPostInstance() {  
    $this->assertTrue(is_a($this->View, 'View'));  
}
```

## Standard Index View

If you use the CakePHP console to bake an index page it will look something like this:



## Posts

Page 1 of 1, showing 4 records out of 4 total, starting on record 1, ending on 4

Id	Title	Body	Created	Updated	Actions
1	Post 1	Lorem ipsum dolor sit amet...	2009-01-23 16:53:42	2009-02-03 12:07:48	<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>
2	Post 2	Lorem ipsum dolor sit amet...	2009-01-24 15:53:53	2009-02-03 16:13:20	<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>
3	Post 3	Lorem ipsum dolor sit amet...	2009-02-03 16:46:10	2009-02-03 16:46:10	<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>
4	Post 4	Lorem ipsum dolor sit amet...	2009-02-03 16:46:59	2009-02-03 16:47:16	<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>

<< previous | next >>

## New Post



The page consists of a title (Posts), information about the number of records (pagination), a table with a header row and the records themselves, some more pagination and finally a new post link. To render this view Cake used two helpers, Pagination and HTML, plus the data for the records.



## Creating The View Test

Now we can create the test for the index page.

```
function testPostIndex() {  
}
```

Because a generic controller class was used, the view has no knowledge of which helpers are needed. They will need to be set manually.

```
$this->View->helpers = array('Html', 'Paginator');
```

The view expects certain pagination information to be set by the controller. For this test that information will need to be set explicitly.

```
$this->View->params['paging'] = array(  
    'Posts' => array(  
        (  
            'count' => 1,  
            'pageCount' => 1,  
            'page' => 1,  
            'current' => 1,  
            'prevPage' => false,  
            'nextPage' => false,  
            'options' => array('limit' => 20),  
            'defaults' => array()  
        )  
    )  
);
```

The records for Post also need be set so that the view has something to display.

```
$this->View->viewVars['posts'] = array (  
    array (  
        'Post' => array(  
            (  
                'id' => 1,  
                'title' => 'Lorem ipsum dolor sit amet',  
                'body' => 'Lorem ipsum dolor sit amet...',  
                'created' => '2009-01-23 16:51:40',  
                'updated' => '2009-01-23 16:51:40'  
            )  
        )  
    )  
);
```

Here we are just setting one record, but you could certainly set as many as you'd like.

All that is left is to render the view and get the output.

```
$html = $this->View->render('index');
```

The string 'index' is passed, specifying which view file to use. The rendered view, not wrapped in the layout, is returned and stored in \$html.

## Testing the Rendered View

At this point the view has been setup and `View::render()` has been called, returning some HTML. The HTML is a string, so to run tests against it you can either use string parsing or convert it to another format.

If you wanted to verify some text appears in the view you could simply use PHP's `strpos` function.

```
$this->assertNotEqual(false, strpos($html,
                                     'Page 1 of 1, showing 1 records out
                                     of 1 total, starting on record 1,
                                     ending on 1'));
```

Another option is to use Cake's XML lib to parse the data. I like to further convert it to an array, which allows me to use the Set lib to test various parts of the view.

```
App::import('Core', 'Xml');
$xml = new Xml($html);
$page = $xml->first()->toArray();
```

It is now possible to check the number of rows in the table by doing:

```
$this->assertEqual(count(Set::extract('/Table/Tr', $page)), 2);
```

In this case there should be 2; the header row and the one record.

To test that the table headers are correct:

```
$this->assertEqual(Set::extract('/Table/Tr/Th/a/value', $page),
                    array('Id', 'Title', 'Body', 'Created', 'Updated'));
```

To check a specific field in a record:

```
$this->assertEqual(
    trim(array_shift(Set::extract('/Table/Tr[2]/Td/3', $page))),
    '2009-01-23 16:51:40'
);
```

There's a lot going on in that one line, so let's work from the inside out.

```
Set::extract('/Table/Tr[2]/Td/3', $page)
```

This returns the 3<sup>rd</sup> field from the 2<sup>nd</sup> row. Since the 1<sup>st</sup> row is the header, this is actually the 1<sup>st</sup> data record.

```
trim(array_shift(Set::extract('/Table/Tr[2]/Td/3', $page)))
```

Set::extract actually returns an array, but we just want the first element, which should be the only one. PHP's array\_shift pulls the one record for us. Finally, the data needs to be trimmed because the view probably looks like this:

```
<td>
    <?php echo $post['Post']['created']; ?>
</td>
```

Notice how the HTML is nicely formatted. We don't care about all that spacing, so it is trimmed off to leave just the value we want to test.

After all that the final output should match the expected value, '2009-01-23 16:51:40' in this case.

The full view test is available at

[http://github.com/mcurry/cakephp/blob/master/test\\_sample/tests/cases/views/post\\_view.test.php](http://github.com/mcurry/cakephp/blob/master/test_sample/tests/cases/views/post_view.test.php)

## Controllers

This section contributed by [Mark Story](http://mark-story.com/posts/view/testing-cakephp-controllers-the-hard-way). Originally posted at <http://mark-story.com/posts/view/testing-cakephp-controllers-the-hard-way>

By now you already know or should know about [CakeTestCase::testAction\(\)](#) and the wondrous things it can do. However, testAction has a few shortcomings. It can't handle redirects, it doesn't let you use the power of Mocks, and it's impossible to make assertions on object state changes. Sometimes you need to do things the hard way, stick your fingers in the mud and work it out. However, knowing how to test a controller the old fashioned way takes a good knowledge of CakePHP.

### Doing Things the Hard Way

Controllers require a number of callbacks to be called before they are workable objects. So let's go, we'll start with the venerable PostsController, and make a nice test for it.

```
App::import('Controller', 'Posts');

class TestPostsController extends PostsController {
    var $name = 'Posts';

    var $autoRender = false;

    function redirect($url, $status = null, $exit = true) {
        $this->redirectUrl = $url;
    }

    function render($action = null, $layout = null, $file = null) {
        $this->renderedAction = $action;
    }

    function _stop($status = 0) {
        $this->stopped = $status;
    }
}

class PostsControllerTestCase extends CakeTestCase {
    var $fixtures = array('app.post', 'app.comment',
                          'app.posts_tag', 'app.tag');

    function startTest() {
    }

    function endTest() {
    }
}
```

So we start off with a basic test class. Important things to notice are the fixtures array and the test class. I've included all the fixtures that are related to the models my controller is going to use. This is important, as you will get tons of table errors until they are all setup.

You may have noticed that I created a subclass of the test subject; this lets me do a few things. First I can test functions that call `redirect()`, as they no longer redirect. I can also call methods that use `$this->_stop()` as they no longer halt script execution. Furthermore, I override `Controller::render()` so I can test actions that use `render()` without having to deal with piles of HTML. I personally don't do many tests that assert the HTML of my views because I find it takes too much time and is tedious. Lastly, I set `$autoRender` to false just in case.

```
function startTest() {
    $this->Posts = new TestPostsController();
    $this->Posts->constructClasses();
    $this->Posts->Component->initialize($this->Posts);
}

//tests are going to go here.

function endTest() {
    unset($this->Posts);
    ClassRegistry::flush();
}
```

We then build the instance and call some basic callbacks, much like [Daniel blogged about](#). At this point we have a controller instance and all the components and models built. We are now ready to start doing some testing.

## Testing A Controller Method

Testing a controller method is just like testing any other method. Often there is a bit more setup involved as controllers require more inputs by nature. However, it is all achievable in the test suite. So we are going to do a test of our `admin_edit` method. This `admin_edit` is straight out of `bake`, so you should know what it looks like. Furthermore, I can show you how you can test methods.

```
function testAdminEdit() {
    $this->Posts->Session->write('Auth.User', array(
        'id' => 1,
        'username' => 'markstory',
    ));
    $this->Posts->data = array(
        'Post' => array(
            'id' => 2,
            'title' => 'Best article Evar!',
            'body' => 'some text',
        ),
        'Tag' => array(
            'Tag' => array(1,2,3),
        )
    );
}
```

At this point I've created the inputs I need for my controller action. I've got a session and some test data. I've provided enough information in the session that `AuthComponent` will let me by and edit my records. However, many would say that you should bypass `Auth` entirely in your unit testing and just focus on the subject method. But being thorough never hurt.

```
function testAdminEdit() {
    $this->Posts->Session->write('Auth.User', array(
        'id' => 1,
        'username' => 'markstory',
    ));
    $this->Posts->data = array(
        'Post' => array(
            'id' => 2,
            'title' => 'Best article Evar!',
            'body' => 'some text',
        ),
        'Tag' => array(
            'Tag' => array(1,2,3),
        )
    );
    $this->Posts->beforeFilter();
    $this->Posts->Component->startup($this->Posts);
    $this->Posts->admin_edit();
}
```

I've now simulated most of a request in CakePHP. It is important to fire the callbacks in the correct order. Just remember that `beforeFilter` happens before `Component::startup()`, and `Component::beforeRender()` happens after you call your controller action.

## Making assertions

When I test controllers I usually make assertions on the viewVars that are set and any records that are modified / deleted. I don't like making assertions on the contents of `$this->Session->setFlash()` as I find these messages change often which can lead to broken tests, which leads to frowns.

Continuing from before:

```
function testAdminEdit() {
    $this->Posts->Session->write('Auth.User', array(
        'id' => 1,
        'username' => 'markstory',
    ));
    $this->Posts->data = array(
        'Post' => array(
            'id' => 2,
            'title' => 'Best article Evar!',
            'body' => 'some text',
        ),
        'Tag' => array(
            'Tag' => array(1,2,3),
        )
    );
    $this->Posts->beforeFilter();
    $this->Posts->Component->startup($this->Posts);
    $this->Posts->admin_edit();

    //assert the record was changed
    $result = $this->Posts->Post->read(null, 2);
    $this->assertEqual($result['Post']['title'], 'Best article Evar!');
    $this->assertEqual($result['Post']['body'], 'some text');
    $this->assertEqual(Set::extract('/Tag/id', $result), array(1,2,3));

    //assert that some sort of session flash was set.
    $this->assertTrue($this->Posts->
        Session->check('Message.flash.message'));
    $this->assertEqual($this->Posts->redirectUrl,
        array('action' => 'index'));
}
```

So there you go, a nice simple test for a controller, with redirects and session flashes. Since we are testing with the real session, we should do the following to ensure there is no bleed-through between tests:

```
function endTest() {  
    $this->Posts->Session->destroy();  
    unset($this->Posts);  
    ClassRegistry::flush();  
}
```

By destroying the session we ensure that we have a clean slate on each test method. So that's it, really. Testing controllers really isn't as hard as it may seem. There are some additional tricks that can be done with Mocks but that is another article all together.



## Mock Objects

This section contributed by [Mark Story](http://mark-story.com/posts/view/testing-cakephp-controllers-mock-objects-edition). Originally posted at <http://mark-story.com/posts/view/testing-cakephp-controllers-mock-objects-edition>

I recently wrote an article about testing CakePHP controllers the hard way where I covered testing controllers by running their methods manually. I hinted at some additional tricks that could be performed by using Mock Objects. Today I'm going to spill the beans on Mocks, and how I use them when testing my Controllers.

### What is a Mock Object

First, figuring out what Mock objects are and are not is important. [The Wikipedia says:](#)

In object-oriented programming, mock objects are simulated objects that mimic the behavior of real objects in controlled ways. A computer programmer typically creates a mock object to test the behavior of some other object, in much the same way that a car designer uses a crash test dummy to simulate the dynamic behavior of a human in vehicle impacts.

In unit testing we use mocks both as a way to isolate our unit (the object being tested) from the world and to allow us to create unexpected situations. Ever wonder how your application would react if a random value was injected, or wonder how you can easily trigger that exception that relies on FooBarComponent returning false which it only does if the file system is corrupted? Well, these are both situations that can be simulated with Mock Objects. Mock objects can also take an active role in Unit testing and contribute additional assertions to your tests in the form of expectations. Mocks are not the solution to all your testing woes, nor will they make a good cup of coffee, but moving on.

### Where can I get one of these fabulous devices?

Well, since we are working in the context of a CakePHP unit test and therefore using Simple Test, we get Mock objects from the Mock class. We generate Mock objects from existing classes, and due to the magic of Reflection and `eval()`, a Mock object class is generated. The simplest example would be:

```
App::import('Component', 'Email');
Mock::generate('EmailComponent');

//Then when we need it

$this->Posts->Email = new MockEmailComponent();
```

This will generate a class called MockEmailComponent. This MockEmailComponent class will have all the same methods as our real EmailComponent. One big difference between the real object and the mock is that all of the mock methods do not work. They all return null and take any number of arguments. But hold on a second, if its methods don't have returns what good are they? Well, lots, because you can set the return value or return reference value of all its methods.

```
$this->Posts->Email->setReturnValue('send', true);
```

This will set the return value of `send()` to `true`. However, it will not send an email, which is the nice part. Because getting emails from your test suite is never fun. Also it allows the test to run on machines that don't even have e-mail servers, like a development box. Using a Mock to test email being sent also allows you to test what happens when your email server is down or other difficult to simulate situations. Generating a Mock object also allows you to append extra methods onto your objects, such as those you are going to build but haven't. This would look a little something like:

```
Mock::generate('EmailComponent',
               'MockEmailComponent',
               array('slice', 'dice'));
```

We've now added the methods `slice()` and `dice()` to our `MockEmailComponent`. In addition to complete Mocks, we can build partial Mocks. A partial Mock is just what it sounds like. It has only a few of its methods mocked. The rest stay as is in the declared class(es). This is really handy for objects that use only a few methods to write to a resource. An example of this would be your controllers. In my previous article I used a subclass to dummy out the `render` and `redirect` methods. However, we could also do this with partial Mock objects.

```
Mock::generatePartial('PostsController',
                     'TestPostController',
                     array('render', 'redirect'));
```

In our tests we can now use Mock expectations to assert that our redirects and render calls are occurring properly.

## Making expectations with Mock Objects

Mocks can be used to feed your application values, as seen above. Furthermore, Mock Objects can be used to introspect on your unit and ensure that it is properly delegating / calling methods on its inner objects. Say for example we wanted to test the use of `SessionComponent::setFlash()`. Now, we could not mock it, and make an assertion before and after the method has run to test that the value in the session was not set and then is set. This will work just fine until we start adding lots of test methods that use the session. We could easily run into bleed-through between our tests, causing our tests to become dependent on the order in which they are run, or worse yet, create broken tests. This is no good. Furthermore, using the real session will nuke any sessions we have with the box we are testing on. Using a Mock object for our session solves all of these problems.

```
//Import and generate the mock we want.
App::import('Component', 'Session');
Mock::generate('SessionComponent');

//In one of our test methods
$this->Posts->Session = new MockSessionComponent();
$this->Posts->Session->setReturnValue('read', 1, array('Auth.User.id'));
$this->Posts->Session->expectOnce('setFlash');
$this->Posts->admin_edit();
```

In the above we've not only set a return value for our SessionComponent's read() method when passed the argument of Auth.User.id, but we've also made an expectation that setFlash() will be called exactly once. If it is called twice or never this assertion will fail, and we will get the red bar of doom. Notice that I didn't make an assertion on the value being passed to setFlash(). You totally can expect certain parameters to be passed to mock methods. However, I find setting assertions for the values being fed into methods like setFlash() can be subject to a lot of change. If we were to make assertions on these inputs, we would need to update our tests each time the message changes. I personally find I'm more interested that the method is called, giving the user feedback, than what the exact contents of that feedback are.

There is so much that can be said about mock objects. However, this post is long enough for the time being. Be sure to check out the [SimpleTest Documentation on Mock Objects](#) for a complete reference on the API and more additional information about SimpleTest mock objects.

## Models

### Test Case

As of CakePHP 1.2 Final, the bake console will produce model test cases using the ClassRegistry to get an instance of the test model. This saves you from having to extend the model you want to test and explicitly set it to use the test database. Make sure your test model is created like this:

```
function startTest() {  
    $this->Post =& ClassRegistry::init('Post');  
}
```

If you are still using App::import and then creating an instance of your model, it's time to upgrade! The downside to using the ClassRegistry method is that it will automatically load all the related models. So you have to include fixtures for all the related models as well, even if you aren't going to be testing them.

### Fixtures

If you open up the default fixture created by the console you'll probably notice a large block of code similar to this:

```
var $fields = array(  
    'id' => array('type'=>'integer', 'null' => false,  
                 'default' => NULL, 'key' => 'primary'),  
    'title' => array('type'=>'string', 'null' => false, 'default' => NULL),  
    'content' => array('type'=>'string', 'null' => false,  
                     'default' => NULL),  
    'created' => array('type'=>'datetime', 'null' => true,  
                     'default' => NULL),  
    'modified' => array('type'=>'datetime', 'null' => true,  
                      'default' => NULL),  
);
```

For most test fixtures, it really isn't necessary to re-define the table's scheme in your fixture file. If you make changes in the schema you'll need to update the fixture as well. You can simply remove this block and replace it with a line telling the fixture to use the same schema as the actual table.

```
var $import = array('table' => 'posts', 'import' => false);
```

Make sure the test database is completely empty - no tables at all - when you run your tests. If your test case fails because of a PHP error, it may leave a table hanging around. This will mess up your test the next time you try to run it, which will lead to a cycle of tests that are failing for no reason. You'll waste hours and go borderline insane. Make sure your DB is empty!

# Merging Add and Edit Actions

## The Controller

Often times when creating an application you'll have the need for both add and edit pages. The default controller and views created when using the bake console treat these as separate areas, which leads to a fair amount of duplicate code.

These are the default add and edit actions created by bake console:

```
function add() {
    if (!empty($this->data)) {
        $this->Post->create();
        if ($this->Post->save($this->data)) {
            $this->flash(__('User saved.', true),
                array('action'=>'index'));
        } else {
        }
    }
}

function edit($id = null) {
    if (!$id && empty($this->data)) {
        $this->flash(__('Invalid Post', true), array('action'=>'index'));
    }
    if (!empty($this->data)) {
        if ($this->Post->save($this->data)) {
            $this->flash(__('The Post has been saved.', true),
                array('action'=>'index'));
        } else {
        }
    }
    if (empty($this->data)) {
        $this->data = $this->Post->read(null, $id);
    }
}
```

You can see the entire add method is duplicated in the edit method. We can basically get rid of the add method altogether and let the edit method handle the creation of new items. There are a couple of different ways to handle this.

You can remove add() entirely by adding a route that points /add to the edit action. The second route is for apps with [prefix routing](#) enabled.

```
Router::connect('/:controller/add', array('action' => 'edit'));
Router::connect('/:prefix/:controller/add', array('action' => 'edit'));
```

Now some changes to the edit action need to be made so that it can handle adds as well.

Remove the first if block:

```
if (!$id && empty($this->data)) {  
    $this->flash(__('Invalid Post', true), array('action'=>'index'));  
}
```

Change the last if statement from:

```
if (empty($this->data)) {
```

to:

```
if ($id && empty($this->data)) {
```

Your edit action should now look like this:

```
function edit($id = null) {  
    if (!empty($this->data)) {  
        if ($this->Post->save($this->data)) {  
            $this->flash(__('The Post has been saved.', true),  
                array('action'=>'index'));  
        } else {  
        }  
    }  
    if ($id && empty($this->data)) {  
        $this->data = $this->Post->read(null, $id);  
    }  
}
```

## Telling Add And Edit Apart

There are times when it may be useful to distinguish which action is really being called. Maybe it's for ACL reasons, where you want to allow a user to edit a record, but not create a new one. By setting the route as described above your controller's `$this->action` will always be "edit". You could add an additional parameter to the route, which would be used to detect the intended action:

```
Router::connect('/:controller/add', array('action' => 'edit',  
                                           'origAction' => 'add'));
```

Now it is simply a matter of checking if the `origAction` is set and using that instead:

```
$action = !empty($this->params['origAction']) ? $this->  
params['origAction'] : $this->action;
```

## The View

You can delete the add.ctp view file, since it is no longer called anymore. The add.ctp and edit.ctp files are extremely similar by default, so doing this will save you a lot of code redundancy.

The edit view will work for adds without any modifications.

You may want to make a change the legend to read differently depending on the action.

```
$action = !empty($this->params['origAction']) ? $this->params['origAction'] : $this->action;
echo sprintf(__('%s %s', true),
             __(ucwords($action), true),
             __('Post', true));
```

This will change the legend to read either "Add Post" or "Edit Post", depending on the action.

# Cake Tricks from The Core

## Cake Style \$options Parameter

We've all seen functions that start out taking one parameter and then, as they grow, they end up looking like this:

```
function geoLocationXML($prodId=null, $userId=null, $startDate=null,
                        $endDate=null, $allProgs=false, $detail=true) {
    if(!$prodId) {
        $prodId = $this->id;
    }

    if(!$userId) {
        $userId = User::get('id');
    }

    if(!$startDate) {
        $startDate = strtotime('-1 month');
    }

    if(!$endDate) {
        $endDate = time();
    }
}
```

Many Cake functions solve this problem by taking an \$options parameter, which is a keyed array. This allows the option list to grow without throwing the function declaration out of whack. Within the function, defaults can be set for the various parameters. Here's the function above rewritten with \$options support:

```
function geoLocationXML($options = array()) {
    $options = array_merge(array('prodId' => $this->id,
                                'userId' => User::get('id'),
                                'startDate' => strtotime('-1 month'),
                                'endDate' => time(),
                                'allProgs' => false,
                                'detail' => true),
                            $options);
}
```

This approach merges the passed \$options with a default options array. The end \$options contains the values passed, plus the defaults for any key that wasn't included. The Cake \$options approach is easier to read, shorter to code and much more extensible.



## Handling Data Arrays with a Single Record or an Array of Records

There are times when it is useful to have a function that can handle a single data record or an array of records. Unfortunately, since `$data` is an array in both cases, it can be difficult to tell which type it is. The trick is with the type of array. If it is just a single record, it is a keyed array - something like:

```
Array
(
    [Post] => Array
        (
            [id] => 1
            [title] => Post 1
            [body] => Lorem ipsum dolor sit amet...
            [created] => 2009-01-23 16:53:42
            [updated] => 2009-02-03 12:07:48
        )
)
```

In the case where there are multiple records, you will have an indexed array:

```
Array
(
    [0] => Array
        (
            [Post] => Array
                (
                    [id] => 1
                    [title] => Post 1
                    [body] => Lorem ipsum dolor sit amet...
                    [created] => 2009-01-23 16:53:42
                    [updated] => 2009-02-03 12:07:48
                )
            )
    [1] => Array
        (
            [Post] => Array
                (
                    [id] => 2
                    [title] => Post Two
                    [body] => Lorem ipsum dolor sit amet...
                    [created] => 2009-01-24 15:53:53
                    [updated] => 2009-02-03 16:13:20
                )
            )
)
```

The trick to telling them apart is to look at the keys. If they are all numeric, it is safe to assume you are dealing with an array of data records. Cake provides a very simply way to do this - the `Set::numeric` method.

```
Set::numeric(array_keys($data));
```

Then it is simply a matter of converting that single record into an array of records (although there is still only one entry).

```
if(!Set::numeric(array_keys($data)) {  
    $data = array($data);  
}
```

Your function can now handle a single record or an array of records, regardless of how that data is passed. You just loop through `$data` and process each of the records.

```
foreach($data as $i => $record) {  
    //do stuff here  
}
```

# Stupid Easy URL Slugs

I've seen this idea floated on a bunch of different blogs. Many offer more complicated methods or behaviors/helpers to simplify things. This is the most basic, stupidly easy way to accomplish it.

If you are viewing a single item within a Cake site, your URL will be something like `/posts/view/5`, where “posts” is the model and “5” is the id of the item.

To generate a link to `/posts/view/5` you'd use code like this:

```
$html->link('CakePHP Tips', array('controller' => 'Post',  
                                   'action' => 'view',  
                                   5));
```

But you probably aren't passing in the id directly, passing the element of a data array instead. Something like:

```
$html->link($post['Post']['title'], array('controller' => 'Post',  
                                          'action' => 'view',  
                                          $post['Post']['id']));
```

Assuming you're using a standard action for your view, like:

```
function view($id=null) { ... }
```

It is safe to create URLs with extra parameters. For example, the URL `/posts/view/5/cakephp-tips` will render the exact same as `/posts/view/5`. That means you can sneak the slug onto your link and not have to make any changes in your controller. CakePHP even provides a method for creating slugs: `Inflector::slug`.

You can now create your links like this:

```
$html->link($post['Post']['title'], array  
    (  
        'controller' => 'Post',  
        'action' => 'view',  
        $post['Post']['id'],  
        Inflector::slug($post['Post']['title'])  
    )  
);
```

This will generate the URL `/posts/view/5/cakephp_tips`. With this method you don't have to worry about keeping track of slugs in your database or re-routing slugs if the title changes.

## Making The Slugs A Bit Smarter

### Changing The Underscore To A Hyphen

If you prefer '-' instead of '\_' as the replacement for spaces, just pass it as the second parameter to `Inflector::slug` like this:

```
Inflector::slug($post['Post']['title'], '-')
```

### Consistency And SEO Improvements

Using this method for your slugs means that anything is allowed in the URL and your site will still display it. For instance, someone could make the URL be `/posts/view/5/cakephp_sucks_and_so_does_your_site` and your site would happily display the content indexed at id 5. Not only does this allow for malicious links, but it can hurt your SEO by having so many different links point to the same content.

It's easy enough to check if the slug being requested matches the intended slug and redirect the user if it doesn't. In your controller's action, after you load the post, put:

```
If(Inflector::slug($post['Post']['title']) != $this->params['pass'][1]
  || count($this->params['pass']) != 2) {
  $post = $this->Post->read(null, $id);
  $this->redirect(array($id,
                      Inflector::slug($post['Post']['title']), 301));
}
```

# jQuery

The JavaScript and AJAX helpers for CakePHP both make use of the Prototype and script.aculo.us libraries. Although these are both solid libraries, many developers are shifting to jQuery for their JavaScript needs. Some may view not having a helper for jQuery as a negative, but I probably wouldn't use it anyway. I prefer to write my jQuery code directly.

## Replacing \$javascript->event()

The JavascriptHelper used the Prototype library for the event method, which is triggered based on user interaction with the page. If you were using the CakePHP helper, you'd write something like:

```
<?php echo $javascript->event('domId',  
                                'click',  
                                'function() { alert("clicked"); }'); ?>
```

This would fire an alert with the message "clicked" anytime the DOM element with the id "domId" was clicked. Notice that this is PHP code that will generate the proper JavaScript.

To accomplish the same thing with jQuery you would use:

```
$("#domId").click(function() { alert("clicked") });
```

This is pure JavaScript code and can be placed directly in the view or in an external JS file.

For the full list of jQuery events see <http://docs.jquery.com/Event>. Also, jQuery has very powerful selector support, so you're not just stuck using a DOM ID - check out <http://docs.jquery.com/Selectors>.

## Replacing \$ajax->link()

The AjaxHelper's link method creates a special link that, when clicked, sends off an XMLHttpRequest, rather than redirecting the browser. The html that is returned can then be placed in an element in the page. The code for this, using the AjaxHelper, would be:

```
<?php echo $ajax->link('Update!',  
                       '/scores/update',  
                       array('update' => 'divId')); ?>
```

This will create a link with the word "Update!". When it is clicked the Ajax request will be made to the ScoresController, update action. The resulting html will be displayed in the DOM element with the id "divId".

To accomplish the same thing with jQuery, you would first create the html link as normal, giving it an ID:

```
<?php echo $html->link('Update!',  
                        '/scores/update',  
                        array('id' => 'scoresUpdate')) ?>
```

Then you would catch the click event for this link and make the Ajax request.

```
$("#scoresUpdate").click(function() {  
    $.ajax({  
        url: "/scores/update",  
        success: function(html) {  
            $("#scoresUpdate").html(html);  
        }  
    });  
});
```

This is just the tip of the iceberg with jQuery. Check out [http://docs.jquery.com/Tutorials:How\\_jQuery\\_Works](http://docs.jquery.com/Tutorials:How_jQuery_Works) for more.

# Expanding Trees With jQuery

Here's a dead simple way to make an expanding tree system using jQuery. To pull this off we're going to use the [TreeHelper by Andy Dawson \(AD7six\)](#) and the [Treeview jQuery plugin by Jörn Zaefferer](#).

## Basic Tree

First take a look at the [example tree behavior code](#) in the Cookbook. Go ahead and get that running. You should end up with something like this:

app\controllers\categories\_controller.php (line 9)

```
Array
(
    [1] => My Categories
    [2] =>     Fun
    [3] =>     Sport
    [4] =>         Surfing
    [5] =>         Extreme knitting
    [6] =>     Friends
    [7] =>         Gerald
    [8] =>         Gwendolyn
    [9] =>     Work
    [10] =>         Reports
    [11] =>             Annual
    [12] =>             Status
    [13] =>         Trips
    [14] =>             National
    [15] =>             International
)
```

(default) 2 queries took 12 ms

Nr	Query	Error Affected	Num. rows	Took (ms)
1	DESCRIBE `categories` SELECT `Category`.`id`, `Category`.`name`, `Category`.`lft`,	5	5	11
2	`Category`.`right` FROM `categories` AS `Category` WHERE 1 = 1 ORDER BY `Category`.`lft` asc	15	15	1

## TreeHelper

Next, put a copy of the [TreeHelper from bakery.cakephp.org](#) into your /app/helpers directory and include it in your controller. You will also need to change the query to retrieve your data. You can use the built in 'threaded' find type to get the tree data in a format suitable for the Tree

Helper. Here's the full controller code:

```
<?php
class CategoriesController extends ApplicationController {

    var $name = 'Categories';
    var $helpers = array('Tree');

    function index() {
        $categories = $this->Category->find('threaded');
        $this->set('categories', $categories);
    }
}
?>
```

Create a /views/categories/index.ctp file with:

```
<?php
echo $tree->generate($categories);
?>
```



Refresh /categories in your browser and you should end up with:



## CakePHP: the rapid development php framework

- My Categories
  - Fun
    - Sport
      - Surfing
      - Extreme knitting
    - Friends
      - Gerald
      - Gwendolyn
  - Work
    - Reports
      - Annual
      - Status
    - Trips
      - National
      - International

CAKEPHP POWER

(default) 2 queries took 7 ms

Nr	Query	Error	Affected	Num. rows	Took (ms)
1	DESCRIBE `categories`		5	5	6
2	SELECT `Category`.`name`, `Category`.`lft`, `Category`.`right` FROM `categories` AS `Category` WHERE 1 = 1 ORDER BY `lft` ASC		15	15	1

The HTML that is outputted is a standard unordered list.

```
<ul>
  <li>My Categories
    <ul>
      <li>Fun
        <ul>
          <li>Sport
            <ul>
              <li>Surfing</li>
              <li>Extreme knitting</li>
            </ul>
          </li>
          <li>Friends
            <ul>
              <li>Gerald</li>
              <li>Gwendolyn</li>
            </ul>
          </li>
        </ul>
      </li>
      <li>Work
        <ul>
          <li>Reports
            <ul>
              <li>Annual</li>
              <li>Status</li>
            </ul>
          </li>
          <li>Trips
            <ul>
              <li>National</li>
              <li>International</li>
            </ul>
          </li>
        </ul>
      </li>
    </ul>
  </li>
</ul>
```

## TreeHelper With jQuery

Download the [Treeview plugin](#) and extract the zip. To get the Treeview plugin set up you need to copy three items into your webroot.

Put jquery.treeview.js (or one of the minified versions) in your /app/webroot/js. Put jquery.treeview.css in your /app/webroot/css. Put the images folder in /app/webroot/css. We'll move this to a better spot later, but for the sake of getting this working we'll leave it there for now.

Obviously, you'll also need the jQuery library. Include the two JavaScript files and the CSS file in your view or layout.

```
$javascript->link(array('jquery', 'jquery.treeview'), false);  
$html->css('jquery.treeview', null, null, false);
```

Also, make sure you have included the JavascriptHelper class in your controller or ApplicationController class.

```
var $helpers = array('Javascript', 'Tree');
```


The Treeview plugin needs a way to target your tree list. The easiest way to do this to add an element id to the first <ul>.


```
echo $tree->generate($categories, array('id' => 'tree'));
```

Now all you have to do is point the Treeview plugin at your unordered list and let it do its thing.

```
<script type="text/javascript">  
  $(document).ready(function(){  
    $("#tree").treeview();  
  });  
</script>
```

Refresh your browser and you should now be looking at a collapsible tree with all the nodes expanded.

 **CakePHP: the rapid development php framework**



```
graph TD
    MC[My Categories] --> Fun[Fun]
    MC --> Work[Work]
    Fun --> Sport[Sport]
    Fun --> Friends[Friends]
    Sport --> Surfing[Surfing]
    Sport --> EK[Extreme knitting]
    Friends --> Gerald[Gerald]
    Friends --> Gwendolyn[Gwendolyn]
    Work --> Reports[Reports]
    Work --> Trips[Trips]
    Reports --> Annual[Annual]
    Reports --> Status[Status]
    Trips --> National[National]
    Trips --> International[International]
```

CAKEPHP POWER

(default) 2 queries took 7 ms

Nr	Query	Error	Affected	Num. rows	Took (ms)
1	DESCRIBE `categories`		5	5	6
2	SELECT `Category`.`name`, `Category`.`lft`, `Category`.`rght` FROM `categories` AS `Category` WHERE 1 = 1 ORDER BY `lft` ASC		15	15	1

The Treeview plugin has a bunch of different option for how to display or animate the tree. See the [Treeview documentation](#) for the full details.

## Cleaning Up the Images

Leaving the Treeview images in `/app/webroot/css/images` isn't really a great idea. So instead, let's move them to `/app/webroot/img/treeview`. The images are all referenced in the Treeview CSS file, so you'll need to open that and fix the paths. Do a find and replace on "images/". You can replace it with the absolute path ("`/img/treeview/`") or the relative path ("`../img/treeview/`"). Save the CSS file and refresh the categories page in your browser. Everything should look the same as before.

# JavaScript In Views

In some corners of the development world people are morally against putting JavaScript “inline” with the HTML. Anytime you put JavaScript in a Cake view it will be rendered in the middle of the content. One easy way to get around this is to use the `$javascript->codeBlock()` function with the “inline” option set to false and the [Heredoc syntax](#).

In the previous section code that would appear in a view, like this:

```
<script type="text/javascript">
  $(document).ready(function(){
    $("#tree").treeview();
  });
</script>
```

Could alternately be written:

```
<?php
  $javascript->codeBlock(
<<<END
    $(document).ready(function(){
      $("#tree").treeview();
    });
END
, array('inline' => false));
?>
```

This code would get written to your <HEAD> section, in place of the `$scripts_for_layout` variable. The resulting output would be:

```
<script type="text/javascript">
//<![CDATA[
  $(document).ready(function(){
    $("#tree").treeview();
  });
//]]>
</script>
```

A couple notes:

- The END identifier must appear on a line by itself (or with a single closing ;) and no indentation or trailing spaces.
- Heredoc parses PHP variables, much like double quotes. So you can use any view variables in the string. However if you wanted to the \$ to appear in the JavaScript it would need to be escaped with a backslash(\). The reason this isn't needed in the above example is that `$(...)` isn't a valid PHP variable and therefore isn't interpreted.

# Make Your Cake App Fast

## Don't Use \$uses Unless You Really, Absolutely Have To

\$uses is a controller attribute that allows you to access additional models to the default one. Say you have a blog application and one of the controllers is posts. By default you have access to the Post model. If you wanted to also have access to the Comment model you could do:

```
<?php
class PostsController extends AppController {
    var $name = 'Posts';
    var $uses = array('Post', 'Comment');
}
?>
```

### Model Chains

Since Comment is associated to Post through a HasMany relationship you can access the Comment model through Post. Like this:

```
$comments = $this->Post->Comment->findAllByPostId($id);
```

The relation chain extends infinitely, including all models down the line.

### Controller::loadModel and ClassRegistry::init.

Great, but sometimes you do legitimately need access to a model that isn't anywhere in the relation chain. If you are going to use the model throughout the controller go ahead and include it in \$uses, but if you only need it in one action there are better ways. They are Controller::loadModel() and ClassRegistry::init().

```
//the loadModel way
$this->loadModel('Comment');
$comments = $this->Comment->findAllByPostId($id);

//the ClassRegistry way
$Comment = ClassRegistry::init('Comment');
$comments = $Comment->findAllByPostId($id);
```

Controller::loadModel, not to be mistaken for the deprecated loadModel function, creates an instance of the model and assigns it to the controller. You can then access it the same way as you would as if it was loaded through \$uses. ClassRegistry returns an instance of the model.

**Approximate Increase:** Having one or two extra models in your controller's \$uses probably isn't going to kill your app. With one extra model there was about a 4% increase. Seven extra models added approximately 40% increase in page load. Roughly 4-6% for every additional model is a good rule of thumb.

## Use Containable

I described this one above. Go read that section, then come back.

## Set Debug to 0

This one should be a no-brainer, but enough people miss it, so I'm going to include it.

For the Cake engine to run it generates two cache sections. The first is `/tmp/cache/models`. In there you'll find a file for every model in your system containing the table schema. You know those `DESCRIBE table;` queries you see in the query output? That's what they're for. Those queries go away when debug is 0.

The second cache is `/tmp/cache/persistent`. There are a couple different files in there that are used by Cake when running your app. The one that generally causes the most slow down to generate is `cake_core_file_map`. This file stores the paths to various classes in your app. To build the file Cake does a logical, but still time consuming, search of your directory tree looking for the right file.

So what is the difference between debug 0 and debug >0? Oh, about 2.73517104 years. When debug is >0 the cache lifetime on these files is 10 seconds. Switching debug to 0 pushes the expiration to 999 days.

**Approximate Increase:** +80% to 100%

## Cache your slow queries/web service requests/whatever

The Cake cache lib is a great tool for caching single parts of your application. It handles all the gory work of writing to a file or tying into a memory-based caching engine. All you need to do is figure out what to cache.

Let's say you have a query that has been indexed and optimized, but is still too slow. [The Cookbook provides an example](#) of how to wrap it with the cache lib so that you don't need to run it every request.

Or if you have a part of your site that is filled with data returned from a web service, like a recent tweets block (not a great example, since most of the Twitter widgets are JavaScript, but roll with me here). There really is no reason to make the call to the web service on every request. Just wrap it with the cache lib like in the above example.

**Approximate Increase:** +0% to 1000000%, it really depends on your app and what you are caching.

## View Caching

Think of this as entire page caching. [The Cookbook covers the basics](#) and since rendering the page still runs through PHP, there is some flexibility for [maintaining dynamic parts of the page](#). For example, if you were running a store you could cache the product pages, but still have a block showing the user's shopping cart.

**Note:**

There's a section in the Cookbook mixed in here [that covers the various caching engines CakePHP supports](#). However, at the moment (version 1.2.2) view caching uses file based caching and is independent of the cache library.

**Approximate Increase:** +130% to 160%

## HTML Caching

[This one is my own creation](#). It's based on the same principal of the [Super Cache for WordPress](#). Basically, it takes the rendered page and writes it to your webroot as straight HTML. The next time the page is hit, your web server can serve it directly without even having to go to PHP.

There are obvious limitations for this, such as no dynamic content on the page and the cache won't be automatically cleared. Still, it's great for things like RSS feeds, APIs, documentation, etc. Anywhere the anonymous viewers all get the same page.

**Approximate Increase:** ~60000% - This isn't hyperbole, that's the real increase.

## APC (or some other opcode cache)

Wikipedia describes [APC](#) as "[a free, open source framework that optimizes PHP intermediate code and caches data and compiled code from the PHP bytecode compiler in shared memory.](#)" Whatever. It makes shit fast. And you don't have to change any of your code. Fuck yea. Where do I sign up, right?

**Approximate Increase:** +25% to 100%

## Persistent Models

This one isn't mentioned in the Cookbook (I'll add it in the next few days if no one beats me to it. I put it on my todo whiteboard, right below "figure out why putting computers in the clouds is more efficient than their traditional ground based counter parts"). This one is simple to turn on. In your controller (or ApplicationController) add the attribute:

```
var $persistModel = true;
```

After a page refresh, you'll notice two new files in /tmp/cache/persistent for each model included in the controller. One is a cache of the model and the other is a cache of the objects in the ClassRegistry. Like view caching mentioned above, this cache can only be saved on the file system.

**Approximate Increase:** +0% to 200%

How much this one helps depends on your application. If your controller only has one model and it isn't associated with any others, you're not going to see much of a boost. In my demo app there was around 100% increase. There was one model in the controller, which was associated with 3



other models, which had associations of their own.

### Warning:

This one seems to cause issues for a lot of people. It tends to work best in simpler apps where there is a one-to-one model-to-controller ratio. Once you start loading models in different controllers, the objects cache associated with the model may not match and you'll start getting errors.

## Store The Persistent Cache in APC

To enable this you need be using APC and set your “\_cake\_core\_” cache to use APC. In your core.php put:

```
Cache::config('_cake_core_', array('engine' => 'Apc',  
                                   'duration'=> 3600,  
                                   'probability'=> 100,  
                                   ));
```

This takes the cache files normally stored in /tmp/cache/persistent (not including the persistent models) and stores them in memory.

**Approximate Increase:** ~25%

## Speed Up Reverse Routing

There are two methods for doing this. The first is described in a post by [Tim at Debuggable.com](http://Tim at Debuggable.com). Tim's method only works for certain link types and breaks the reverse routing feature. I created a different approach that uses caching. There is a post describing it at [PseudoCoder.com](http://PseudoCoder.com) and the code is available at [GitHub](https://github.com).

**Approximate Increase:** ~50%

Like all of these tips, the actual increase depends on your app. If you don't use many custom routes and don't have many links on your page, you're not going to see much of a benefit.

## Unchain Your Models

Make sure you understand model chaining before reading this section. It is described above in the “Don't Use uses Section” and a little bit in [the Model section of the Cookbook](#).

A lot of you probably realized that although it's great to be able to access related models through chaining, building the chain on every request is not very efficient. Using persistent models attempts to solve that problem by caching the objects, but this doesn't work in every application. Particularly, if you access models using `Controller::loadModel` or `ClassRegistry::init`, using persistent models will often give errors such as:

```
Catchable fatal error: Object of class __PHP_Incomplete_Class
```

Rather than caching the models, it makes sense to not create the chain by default and instead add the links as needed. Thanks to PHP's `__get()` and `__isset()` this is possible. These functions are part of [PHP's overloading ability](#).

To accomplish this check out my [LazyLoader plugin](#) available from GitHub. Unfortunately, `__isset()` is only available in PHP 5.1.0 and above, so this code is limited to those versions.

**Approximate Increase:** Roughly 4-6% for every model that is unchained. You'll see the biggest gain in applications with many interconnected models.

# The Giant Configuration, Version Control and Deployment Section

The topic of handling configuration, multiple environments, and deployment has been covered extensively. At the end of this section are some links to alternate methods. The way described here is much simpler, requires virtually no coding, and isn't dependent on any server detection.

## Version Control

There are three main configuration files for CakePHP. For each one you'll need to decide whether the actual file or just a template of the file is kept under version control .

### core.php

Keep in version control: **Yes**

The core.php configuration file contains a set of global configuration options for your application. In general these options apply to the application, regardless of the environment. It is true that you may need to change a specific setting depending on the app instance. The "debug" setting is the most common. This will be covered in the bootstrap.php section below.

### bootstrap.php

Keep in version control: **Yes** and **No**

I find it convenient to split the bootstrap file into two files. The normal bootstrap.php and a second bootstrap.local.php. The original bootstrap.php is maintained in source control, while the local version is not. You may choose to keep a bootstrap.local.php.template in your source control, which shows some sample settings.

The local version of the bootstrap is included at the bottom of the main bootstrap file like this:

```
<?php
// some bootstrap code here

if(file_exists('bootstrap.local.php')) {
    include('bootstrap.local.php');
}
//EOF
?>
```

This method allows you to maintain your application wide defaults in core.php and bootstrap.php, then override them for any particular environment. For example, on your dev machine you'll likely want to turn on debug. Instead of editing core.php, add the line to your bootstrap.local.php.

```
Configure::write('debug', 2);
```

## database.php

Keep in version control: **No**

To me, this one is the easiest. Don't put it in version control; instead maintain a template of sample settings. I know a lot of people don't agree with me here.

You would never put this in version control with a single `$default` database, since the settings for that database would most definitely need to be changed for each development environment. Then there would be the danger of these development settings getting checked in by mistake. This may get caught before the file is actually deployed to production, but you would still have to deal with the constant annoyance of your `local database.php` getting overwritten.

A second option is to have multiple variables for each environments database. Say `$development` and `$production`, like this:

```
class DATABASE_CONFIG {
    var $development = array(
        'driver' => 'mysql',
        'host' => 'localhost',
        'login' => 'root',
        'password' => '',
        'database' => 'dev_app'
    );
    var $production = array(
        'driver' => 'mysql',
        'host' => 'db.myapp.com',
        'login' => 'production',
        'password' => 'ju3uspaBubrewrap',
        'database' => 'prod_app'
    );
}
```

Then you'd have some logic somewhere that detects the current server and selects the appropriate database config. This would likely lead to the same problems as with the single `$default` option, as the `$development` array may be different for each developer and would lead to headaches if the file was committed.

In addition, your production database settings are now available to anyone with source control access. Many times you won't want all the developers to have this kind of access, whether they are a freelancer or full-time. Granted, this info may not be enough to access the database alone, but why risk it?

Also, any accidental changes made to the `$production` array are unlikely to be caught until the file is actually deployed to production.

Finally, is it really that hard to copy the template to `database.php` and fix the settings? How often do you even do it? Once, twice? Instead you're going to rely on the `$_SERVER` variable? And what happens when you want to roll out a new environment? You need to update and commit the code so that it knows about this new instance. Ugh. Why add all the extra trouble?

## Multiple Environments

Based on the plan outlined above, `core.php` and `bootstrap.php` are committed to source control and are the same on all environments. A template is provided for `database.php` and needs to be set up for each new environment. Optionally, `bootstrap.local.php` can be created, based on a template, which can override anything in `core.php` or `bootstrap.php` specific to the environment.

## Deployment

Whether you use a custom shell script or one of the build packages like [Phing](#) or [Capistrano](#), there are some common things you'll want to do when deploying an update to your site.

### Debug

If you're super paranoid about debug accidentally being enabled, you can force it to zero with a simple perl command.

```
perl -pi -e "s/debug', [0-9]{1}/debug', 0/gi" /path/to/app/config/core.php
```

### Cache

This is the single most common issue with new deployments and also the biggest frustration. If you've made changes to your database, the model caches need to be rebuilt. The easiest way to do this is just to delete them.

```
rm -rf /path/to/app/tmp/cache/models/*
```

Most likely, you'll want to delete your cached views as well.

```
find /path/to/app/tmp/cache/views/ | grep php | xargs rm -f
```

I'm using a slightly different syntax here because this way will handle a large number of files.

## Alternate Methods

Neil Crookes - <http://www.neilcrookes.com/2008/11/28/runtime-config-in-cakephp-apps/>

Chris Hartjes - <http://www.littlehart.net/atthekeyboard/2008/11/28/handling-multiple-environments-in-your-php-application/>

Rafael Bandeira - <http://rafaelbandeira3.wordpress.com/2008/12/05/handling-multiple-environments-on-cakephp/>

Kjell Bublitz - <http://cakealot.com/2008/12/environment-and-database-config-easy-cakephp-deployment/>

# CakePHP Reserved Classes

Unlike some other frameworks, Cake doesn't prefix its classes with the framework name. This may cause some conflicts if you try to create a generically named class or model that may already exist in the Cake core. Here are some of the class names you'll need to avoid and some alternatives.

Class Name	Alternative
App	Application, Site
Cache	Asset, Stash, Repo, Treasury
Configure	Conf, Config, Settings
Controller	Boss, Director
Debugger	Debug, Error
Dispatcher	Gofer, Runner, Emissary
ErrorHandler	Error, ErrorBroker
File	Data, FileHandler
Flay	Scalp, Excoriate
Folder	Directory, Dir, Wrapper
I10n	Lang, Language
I18n	Region, Locale
Model	Standard, Archetype
Multibyte	Mb
Object	Base, Entity
Router	Plotter, Signaler
Sanitize	Purify, Clean
Security	Shield, Guard, Defense
Set	Bunch, Gang, Pack
String	Term, Sequence
Validation	Proof, Verification
Xml	
XmlElement	
XmlNode	

# From The Bakery (And Other Places)

There's a lot of code in the [CakePHP Bakery](#) and although there is an approval process, some of it isn't so great. Below is a list of some of the better entries.

## Behaviors

### Sluggable

By: Mariano Iglesias

Link: <http://bakery.cakephp.org/articles/view/sluggable-behavior>

Description: This behavior lets your models act as slug-based models, useful for generating Search Engine friendly URLs. Easy to install and easy to configure.

### Soft Deletable

By: Mariano Iglesias

Link: <http://bakery.cakephp.org/articles/view/soft-delete-behavior>

Description: This behavior lets you implement soft delete for your records in your models by introducing a flag to an existing table which indicates that a row has been deleted, instead of deleting the record.

### Linkable

By: Rafael Bandeira

Link: <http://github.com/rafaelbandeira3/linkable/tree/master>

Description: LinkableBehavior is the implementation to solve ContainableBehavior's inextensibility, complexity, featurity and - mainly - its db usage.

## Plugins

### DebugKit

By: Mark Story

Link: [http://thechaw.com/debug\\_kit/wiki](http://thechaw.com/debug_kit/wiki)

Description: The CakePHP DebugKit provides a set of easy-to-use debugging information to your applications. It provides functionality for session and request inspection, SQL log inspection, and benchmarking. This functionality is provided as a plugin for existing CakePHP applications.

### NamedScope

By: Joel Moss

Link: <http://github.com/joelmoss/cakephp-namedscope/tree/master>

Description: This NamedScope behavior for CakePHP allows you to define named scopes for a model, and then apply them to any find call. It will automatically create a model method and a method for use with the findMethods property of the model.

## Helpers

### Asset

*Note: Packaged as a plugin*

By: Matt Curry

Link: <http://github.com/mcurry/asset>

Description: Automatically combine and compact JavaScript and CSS files. This helps speed up browsing by limiting the number of requests, as well as reducing the overall file size.

### Jquery Validation

*Note: Packaged as a plugin*

By: Matt Curry

Link: [http://github.com/mcurry/js\\_validate](http://github.com/mcurry/js_validate)

Description: This helper takes your model validation rules and converts them to JavaScript so they can be applied in the client's browser before submitting to the server.

### HtmlCache

*Note: Packaged as a plugin*

By: Matt Curry

Link: [http://github.com/mcurry/html\\_cache](http://github.com/mcurry/html_cache)

Description: Cake's core cache helper is great, but the files it outputs are PHP files, so it will never be as fast as straight HTML files. The HTML Cache Helper writes out pure HTML, meaning the web server doesn't have to touch PHP when a request is made. This helper is for sites with high traffic pages that have nothing unique about the user on the page. Works great for RSS.



# Copyright



You are free:



**to Share** – to copy, distribute, display and perform the work



**to Remix** – to make derivative works

Under the following conditions:



**Attribution.** You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).



**Noncommercial.** You may not use this work for commercial purposes.



**Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page.
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Apart from the remix rights granted under this license, nothing in this license impairs or restricts the author's moral rights.

This is a human-readable summary of the [Legal Code \(the full license\)](#).

# Revisions

## V1.0 – May 13, 2009

Initial Release.

## V1.1 – May 15, 2009

- Fixed custom find methods to be prefixed with two underscores to prevent a conflict with the native find methods.
- Added info about Chris Hartjes' book.
- Moved setting the user automatically to beforeValidate instead of beforeSave.
- A few typos and code syntax errors.

## V1.2 – June 12, 2009

- Added route for prefix when merging add and edit.
- TOC Links!
- Removed call to \_\_construct for defining custom find methods.
- More typos.
- Removed redundant foreignKey option from User model associations.
- Added method for distinguishing between add and edit when merging actions.
- Added advanced options slug handling.