

Cook up Web sites fast with CakePHP, Part 5: Adding cache

A good way to improve the performance of your apps

Level: Intermediate

Duane O'Brien (d@duaneobrien.com), PHP developer, Freelance

16 Jan 2007

CakePHP is a stable production-ready, rapid-development aid for building Web sites in PHP. This "Cook up Web sites fast with CakePHP" series shows how to build an online product catalog using CakePHP. [Part 1](#) focuses on getting CakePHP up and running, and [Part 2](#) demonstrates how to use Scaffolding and Bake. [Part 3](#) shows how to use CakePHP's Sanitize and Security components to help secure your user-submitted data, and [Part 4](#) focuses on the Session component of CakePHP. Here in Part 5, you will learn how to use CakePHP's Sanitize and Security components to help secure your user-submitted data, and you will learn how to handle invalid requests.

Introduction

This series is designed for PHP application developers who want to start using CakePHP to make their lives easier. In the end, you will have learned how to install and configure CakePHP, the basics of Model-View-Controller (MVC) design, how to validate user data in CakePHP, how to use CakePHP Helpers, and how to get an application up and running quickly using CakePHP. It might sound like a lot to learn, but don't worry -- CakePHP does most of it for you.

This article assumes you have already completed [Part 1](#), [Part 2](#), [Part 3](#), [Part 4](#), and that you still have the working environment you set up for those pieces. If you do not have CakePHP installed, you should run through Parts 1 and 2 before continuing.

It is assumed that you are familiar with the PHP programming language, have a fundamental grasp of database design, and are comfortable getting your hands dirty.

System requirements

Before you begin, you need to have an environment in which you can work. CakePHP has minimal server requirements:

1. An HTTP server that supports sessions (and preferably mod_rewrite). This article was written using Apache V1.3 with mod_rewrite enabled.
2. PHP V4.3.2 or greater (including PHP V5). This article was written using PHP V5.0.4.
3. A supported database engine. Today, this means MySQL, PostgreSQL, or use of a wrapper around ADODB. This article was written using MySQL V4.1.15.

You'll also need a database and database user ready for your application to use. This article provides the syntax for creating any necessary tables in MySQL.

The simplest way to download CakePHP is to visit [CakeForge.org](#) and download the latest stable version. This tutorial was written using V1.1.8. (Nightly builds and copies straight from Subversion are also available. Details are in the CakePHP Manual (see [Resources](#)).)

Tor so far

In [Part 4](#), you were given an opportunity to streamline *Tor*. How did you do?

Adding the Favorites Action and View

To view the Favorites list, you need to add a favorites action to the users controller. It might look something like Listing 1.

Listing 1. Adding a favorites action to the users controller

```
function favorites () {
    $username = $this->Session->read(' user' );
    $favorites = array();
    if ( $username )
    {
        $this->User->recursive = 2;
        $results = $this->User->findByUsername($username);
        foreach($results[' Product' ] as $product)
        {
            $favorites[] = array(' Product' => $product, ' Dealer' => $product[' Dealer' ] );
        }
        $this->set(' products', $favorites);
    } else {
        $this->redirect(' /users/login' );
    }
}
```

Note the redirection if the user is not logged in. This keeps the user from seeing an error when viewing the page without logging in.

You would also need a favorites.html in the app/views/users/ directory. It might look something like this:

Listing 2. Favorites.html

```
<h2>Your Favorite Products</h2>
<?php
echo $this->renderElement(' products_table', array(' products' => $products) );
?>
```

All the view really needs to do is show the products table. However, right now, the products all show up with an **Add To Favorites** link, which is silly, considering you are looking at your list of favorites. It should say **Remove From Favorites**.

Putting in the Remove From Favorites Link

Your other task was to put in a **Remove From Favorites** link in the products table, setting it up so that users saw the **Remove** link if a product was in their favorites list, and an **Add** link if the product was not in their favorites list. Taking a look at the products table again, the following section is the most important.

Listing 3. Products table

```
<?php
if ( isset($ajax) ) {
    echo $ajax->link('Add to Favorites', ' /products/add_to_favorites/' .
        $product[' Product' ][ 'id' ], array(' update' => ' updated', ' loading' =>
            "Element. show(' loading' )", ' complete' => "Element. hide(' loading' )" ));
}
?>
```

The **Remove From Favorites** link will be much the same, as shown below.

Listing 4. Remove from Favorites link

```
echo $ajax->link('Remove from Favorites', ' /products/remove_from_favorites/' .
    $product[' Product' ][ 'id' ], array(' update' => ' updated', ' loading' =>
        "Element. show(' loading' )", ' complete' => "Element. hide(' loading' )" ));
```

You will also need the `removeFromFavorites` action in the products controller.

Listing 5. The removeFromFavorites action

```
function removeFromFavorites($id) {
    $username = $this->Session->read(' user' );
    $success = false;
    $user = $this->Product->User->findByUsername($username, ' User.id' );
    $this->Product->User->id = $user[' User' ][ 'id' ];
    $success = $this->Product->User->removeFavorite($id);
    if ( $this->RequestHandler->isAjax() ) {
        $this->set(' products', $this->Product->findAll());
    } else {
        if ( $success ) {
            $this->Session->setFlash(' Removed product from favorites' );
            $this->redirect(' /users/favorites' );
        } else {
            $this->Session->setFlash(' Access denied' );
            $this->redirect(' /products/index' );
        }
    }
}
```

And likewise, the `removeFavorite` method in the users controller must be created.

Listing 6. Creating the removeFavorite method

```
function removeFavorite($product_id) {
    if($this->getId()) {
        $user = $this->read();
        $fav = array();
        foreach($user[' Product' ] as $product) {
            if ($product_id != $product[' id' ])
            {
                $fav[] = $product[' id' ];
            }
        }
        $user[' Product' ] = array(' Product' => $fav);
        if($this->save($user)) {
            return true;
        } else {
            return false;
        }
    }
    return false;
}
```

As you noticed, when you view the favorites list, **Add to Favorites** is displayed. Instead, the user should see the **Remove from Favorites** button for products they already have added to their favorites list. How would you do that?

Caching

Conceptually, caching can sometimes be confusing. There are many types of caching, and each presents its own set of challenges and benefits. It is important to understand in this context what is meant by caching.

What does caching mean?

Generally, *caching* happens anytime a request is made, and the responding application says, "I don't have to go get that. I've already got one." Most of the time, when a computer user hears the word "cache," he thinks of a browser's cache. Typically, in order to speed up the user experience, your browser will keep copies of what it believes are static files -- generally images, stylesheets, static HTML, and script files. While this type of caching can sometimes cause problems for developers of Web applications, this type of caching is not our focus here.

Another example of caching would be when the browser makes a request of your Web application for some content. If your Web application uses caching, it could respond to the request with a previously generated copy of the content, eliminating the resource overhead involved in generating the content a second time. This is the type of caching this article will focus on.

Cache why?

Generally, you would use caching in your application for two reasons. One, it helps reduce resource consumption on your server. While the savings may be small in most cases, for high-traffic sites serving requests in significant volume, these small savings quickly add up to significant performance benefits. The second reason is typically speed. Because your application doesn't have to go through the process of regenerating the content for the request, the content can be served much more quickly. Again, while the savings may be small in most cases, high-traffic sites can quickly realize speed benefits by using caching.

Cache how?

OK -- you're sold. You are ready to cache anything and everything. How do you do it? What does CakePHP give you to make it easy?

For starters, you need to turn caching on. By default, it's disabled. You can enable it in app/config/core.php -- look for the following entry: `define('CACHE_CHECK' , false);` and change it to `define('CACHE_CHECK' , true);`.

By setting this value to true you are telling CakePHP that caching is now in play. Go ahead and do that now, so you can set up caching later. You're not done yet: you have to tell CakePHP exactly what you want cached, and for how long.

Cache what?

Having turned caching on, you have to specify what you want cached. This starts in the controller for the views you want to cache by adding cache to the `helpers` array. For example, if you want to cache the products views, you would have to have cache in the `helpers` array for the products controller. When you built the products controller for Tor, you specified that the HTML and form helpers were in use. Adding cache to this list, the `helpers` array would look like this: `var $helpers = array('Html' , 'Form' , 'Cache');`.

Now that the cache helper is in use, you need to specify exactly what you want cached. There are several ways you can do this, but all of them hinge on the `$cacheAction` array.

Cache a specific request

Suppose you wanted to cache a specific request. Suppose there are three or four products that get high traffic on the view action, and you want to cache just the "view" views for these products. In this case, you would specify the requests you want to cache as array keys for `$cacheAction` and specify the length of time as the value for the key. The `$cacheAction` array is a class variable, like the `$helpers` array. To cache these specific views, `$cacheAction` might look like Listing 7.

Listing 7. \$cacheAction

```
var $cacheAction = array (
    'view/1/' => 3600,
    'view/2/' => 3600,
    'view/3/' => 3600
);
```

This `$cacheAction` is telling CakePHP to cache the "view" views for products 1-3 for 3,600 seconds (one hour). The length of time you specify can be any format that `strtotime()` can interpret. You could just as easily say `1 hour`.

Cache a whole action

Maybe it's not enough to just cache a few products. Perhaps you want to cache all the views for a particular action. Suppose you want to use caching on the views for the edit action. To do so, you would specify the action as an array key and the length of time to keep the views in cache much like you did above.

```
var $cacheAction = array (
    'edit/' => '+1 hour'
);
```

You can even mix and match the two.

Listing 8. Mixing and matching

```
var $cacheAction = array (
    'view/1/' => 3600,
    'view/2/' => 3600,
    'view/3/' => 3600,
    'edit/' => '+1 hour'
);
```

That helps save some time. Now you have cached the "view" views for the most commonly viewed products and all of the edit views. But maybe you want to do more.

Cache everything the controller does

You might want to cache everything the controller does. If so, it's not necessary to specify every action in the `$cacheAction` array. You can simply set the value of `$cacheAction` to a length of time to keep it all cached: `var $cacheAction = '+1 hour';`

The value must be a string that `strtotime()` can interpret. By setting `$cacheAction` to a single value, CakePHP knows to cache any views for the controller.

Cache from within an action

Because `$cacheAction` is a class variable, you can also access the variable from within an action. Supposing you wanted to modify `$cacheAction` from within an action, you would use the same syntax you would use to modify any class variable.

```
function foo() {
    $this->cacheAction = array();
}
```

This generally wouldn't be necessary, but you may find an occasion where it is just what you need. If that's the case, now you know you can. CakePHP offers several ways you can cache your views. That's the easy part. Learning when to cache -- or more specifically, when **not** to cache -- can be a little trickier.

Cache when?

Caching might strike you as the best thing since sliced bread. Moreover, most of the time, caching is exactly what you want it to be. So when might you not want to cache your views?

Most of the time, you wouldn't want to completely cache data that is being updated constantly. For example, suppose the users of *Tor* were adding products several times a minute. In this case, caching the index view might prove to be more of a hindrance than a help. If the content is being updated so frequently that the cached page is never actually served, all you have done is add the overhead of saving the cached page and checking to see if it has been updated to each request.

That doesn't mean that caching isn't of any use, however. You just need to be more specific when you tell CakePHP what to cache.

View <cake:nocache></cake:nocache> markup

Within a view or layout, CakePHP allows you to specifically exclude some content from caching by wrapping the content in <cake:nocache></cake:nocache> tags. Proper use of this markup will allow CakePHP to cache the static parts of the view or layout, while making sure that the dynamic portions of the page are retrieved for every request.

Not everything can be wrapped in <cake:nocache></cake:nocache> tags. Specifically, you cannot just wrap variables in <cake:nocache></cake:nocache> tags to make them dynamic. Generally speaking, the things that can be wrapped in <cake:nocache></cake:nocache> tags are CakePHP constructs, like helpers and element calls.

For example, in *Tor*, the default layout created in Part 4 should look something like this:

Listing 9. Default layout

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Tor : <?php echo $title_for_layout;?></title>
<link rel="icon" href="<?php echo $this->webroot . 'favicon.ico';?>"
type="image/x-icon" />
<link rel="shortcut icon" href="<?php echo $this->webroot .
'favicon.ico';?>" type="image/x-icon" />
<?php
if ( isset($javascript) ) {
    echo $javascript->link('prototype.js');
    echo $javascript->link('scriptaculous.js?load=effects');
    echo $javascript->link('controls.js');
}
?>
<?php echo $html->css('cake.generic');?>
</head>
<body>
<div id="container">
    <div id="header">
        <h1><?php echo $html->link('Tor', '/') ?> : Welcome <?php echo
$session->read('user') ?></h1>
        <div id="menu">
            <?php echo $html->link('Products', '/products') ?> |
            <?php echo $html->link('View Favorites', '/users/favorites') ?> |
            <?php echo $html->link('Login', '/users/login') ?> |
            <?php echo $html->link('Logout', '/users/logout') ?> |
        </div>
    </div>
    <div id="content">
        <?php if ($session->check('Message.flash'))
        {
            $session->flash();
        }
        echo $content_for_layout;
    ?>
    </div>
</div>
<?php echo $cakeDebug?>
</body>
</html>
```

If caching were enabled, the first time any page that uses the default layout was loaded, the Welcome <?php echo \$session->read('user') ?> would be replaced with something like Welcome or Welcome wrestler -- meaning that no matter what user was logged in, he would see a cached username.

To specify that the username shouldn't be cached, you would wrap the <?php echo \$session->read('user') ?> line in <cake:nocache></cake:nocache> tags. The end result would look like this:

Listing 10. Specifying that the username shouldn't be cached

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Tor : <?php echo $title_for_layout;?></title>
<link rel="icon" href="<?php echo $this->webroot . 'favicon.ico';?>"
type="image/x-icon" />
<link rel="shortcut icon" href="<?php echo $this->webroot .
'favicon.ico';?>" type="image/x-icon" />
<?php
if ( isset($javascript) ) {
    echo $javascript->link('prototype.js');
    echo $javascript->link('scriptaculous.js?load=effects');
    echo $javascript->link('controls.js');
}
?>
<?php echo $html->css('cake.generic');?>
</head>
<body>
<div id="container">
    <div id="header">
        <h1><?php echo $html->link('Tor', '/') ?> : Welcome
```

```
<cake:nocache><?php echo $session->read('user')
?></cake:nocache></h1>
<div id="menu">
    <?php echo $html->link('Products', '/products') ?> |
    <?php echo $html->link('View Favorites', '/users/favorites') ?> |
    <?php echo $html->link('Login', '/users/login') ?> |
    <?php echo $html->link('Logout', '/users/logout') ?> |
</div>
</div>
<div id="content">
    <?php if ($session->check('Message.flash'))
    {
        $session->flash();
    }
    echo $content_for_layout;
?>
</div>
</div>
<?php echo $cakeDebug?>
</body>
</html>
```

You can test this out by specifying that the products controller uses caching for the view action. Add cache to the list of helpers and specify a view in `$cacheAction`.

Listing 11. Specifying that the products controller uses caching for the view action

```
<?php
class ProductsController extends AppController
{
    var $name = 'Products';
    var $helpers = array('Html', 'Form', 'Javascript', 'Ajax', 'Cache');
    var $components = array('Acl', 'RequestHandler');
    var $cacheAction = array(
        'view/' => '+1 hour'
    );
    ...
}
```

Now, view a product. You should see a file created in `app/tmp/cache/views` that corresponds to the product you viewed. Log in as a different user and view the same product. You'll find that your username is not cached. Edit the product. You will find that CakePHP knows to delete the cached view. Is that not nifty?

Using the `<cake:nocache></cake:nocache>` tags can help you find that delicate mix between caching your views and keeping your content up to date. Sometimes that's not enough. Sometimes you need to clear out the cache yourself.

When to clear the cache

Even if you are smart about what is cached and what isn't, sometimes it's necessary to clear the cache manually. You may be thinking that this is obviously necessary when the data has been updated. For example, if a product is edited, the view and edit views for that product, as well as the products index view would all have to be cleared from the cache. And you're right. They do need to be cleared.

But *you* don't have to do it. CakePHP does it for you. CakePHP knows when data is inserted, updated, or deleted, and if this affects a cached view, the cache for that view is cleared. That saves an awful lot of work.

But it may be necessary to explicitly clear the cache. For example, if an external process, such as a scheduled batch script, is updating the data in the database directly rather than using the application directly, the cache would need to be cleared manually. This can be done using the global function `clearCache`.

How to clear the cache

Just as CakePHP gave you many ways to put things into the cache, there are several ways you can clear things from the cache:

- To clear the cache of only the cached "view" view for Product 1, the syntax would be: `clearCache('products_view_1');`
- To clear the cache of all the "view" views for the products controller, the syntax would be `clearCache('products_view');`
- To clear the cache of all views for the products controller, the syntax would be `clearCache('products');`
- To clear the cache of multiple types of views, you would pass an array to `clearCache`:

```
clearCache('products', 'users_view');
```

If you want to blow the cache clean, simply call the function with no parameters: `clearCache();`. You could call this function by creating an `emptycache` controller and putting the function call in the index action. This action could then be called directly, manually, or via something like we get in the event that the process needed to be automated.

Summary

Congratulations! You completed the series. You're probably psyched and ready to bake your own application.

But before you do, in `app/config/core.php`, set `DEBUG` to 2. This will tell CakePHP to display some SQL debug information at the bottom of your views. You can find the cached pages in `app/tmp/cache/views`. Open one up and have a look at what a the cached page looks like. Delete the cached files and view a product. Make a note of how many queries were run and how long they took. Now reload. The view has been cached. How many queries ran this time? How long did they take? Get familiar with caching. Look for ways to use it in *Tor*. When you're done, throw it all away and dive into your own application.

You've learned a lot in this "[Cook up Web sites fast with CakePHP](#)" series, but nothing will put it in perspective like writing your own application from scratch in CakePHP.

Share this...

 [Digg this story](#)

 [Post to del.icio.us](#)

 [Slashdot it!](#)

Download

Description	Name	Size	Download method
Part 5 source code	os-php-cake5.source.zip	158KB	HTTP

→ [Information about download methods](#)

Resources

Learn

- Visit [CakePHP.org](#) to learn more about it.
- The [CakePHP API](#) has been thoroughly documented. This is the place to get the most up-to-date documentation for CakePHP.
- There's a ton of information available at [The Bakery](#), the CakePHP user community.
- [CakePHP Data Validation](#) uses PHP Perl-compatible regular expressions.
- Read a tutorial titled "[How to use regular expressions in PHP.](#)"
- Want to learn more about design patterns? Check out [Design Patterns: Elements of Reusable Object-Oriented Software](#).
- Check out some [Source material for creating users](#).
- Check out the [Wikipedia Model-View-Controller](#).
- Here is more useful background on the [Model-View-Controller](#).
- [Here's a whole list](#) of different types of software design patterns.
- Read about [Design Patterns](#).
- Visit IBM developerWorks' [PHP project resources](#) to learn more about PHP.
- Stay current with [developerWorks technical events and webcasts](#).
- Check out upcoming conferences, trade shows, webcasts, and other [Events](#) around the world that are of interest to IBM open source developers.
- Visit the developerWorks [Open source zone](#) for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.
- To listen to interesting interviews and discussions for software developers, be sure to check out [developerWorks podcasts](#).

Get products and technologies

- Innovate your next open source development project with [IBM trial software](#), available for download or on DVD.

Discuss

- The developerWorks PHP Developer Forum provides a place for all PHP developer discussion topics. Post your questions about PHP scripts, functions, syntax, variables, PHP debugging and any other topic of relevance to PHP developers.
- Get involved in the developerWorks community by participating in [developerWorks blogs](#).

About the author

Duane O'Brien has been a technological Swiss Army knife since the Oregon Trail was text only. His favorite color is sushi. He has never been to the moon.