
Cook up Web sites fast with CakePHP, Part 2: Bake bigger and better with CakePHP

Skill Level: Intermediate

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

12 Dec 2006

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 you how to build an online product catalog using CakePHP.

Section 1. Before you start

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.

About this series

- [Part 1](#) focuses on getting CakePHP up and running, and the basics of how to put together a simple application allowing users to register for an account and log in to the application.
- Part 2 demonstrates how to use scaffolding and Bake to get a jump start on your application, and using CakePHP's access control lists (ACLs).
- [Part 3](#) shows how to use Sanitize, a handy CakePHP class, which helps secure an application by cleaning up user-submitted data. Part 3 also covers the CakePHP security component, handling invalid requests and other advanced request authentication.

- [Part 4](#) focuses primarily on the Session component of CakePHP, demonstrating three ways to save session data, as well as the Request Handler component to help you manage multiple types of requests (mobile browsers, requests containing XML or HTML, etc).
- And [Part 5](#) deals with caching, specifically view and layout caching, which can help reduce server resource consumption and speed up your application.

About this tutorial

This tutorial shows you how to jumpstart your CakePHP application using scaffolding and Bake. You will also learn the ins and outs of using CakePHP's ACLs. You'll get a look at what scaffolding is and what it provides. Then you'll learn how to use Bake to generate the code for a scaffold, letting you tweak it as you go. Finally, you will learn about ACLs: what they are, how to create them, and how to use them in your application. This tutorial builds on the online product application *Tor* created in [Part 1](#).

Prerequisites

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. A full grasp of the MVC design pattern is not necessary, as the fundamentals will be covered during this tutorial. More than anything, you should be eager to learn, ready to jump in, and anxious to speed up your development time.

System requirements

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

1. An HTTP server that supports sessions (and preferably `mod_rewrite`). This tutorial was written using Apache V1.3 with `mod_rewrite` enabled.
2. PHP V4.3.2 or later (including PHP V5). This tutorial was written using PHP V5.0.4
3. A supported database engine (currently MySQL, PostgreSQL or using a wrapper around ADODB). This tutorial was written using MySQL V4.1.15.

You'll also need a database ready for your application to use. The tutorial will provide 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](#)).)

Section 2. Tor, so far

At the end of [Part 1](#), you were given an opportunity to put your skills to work by building some missing functionality for Tor. Login/Logout, index, the use of hashed passwords, and automatically logging a registering user were all on the to-do list. How did you do?

The login view

Your login view might look something like Listing 1.

Listing 1. Login view

```
<?php
if ($error)
{
    e('Invalid Login.');
```

}

```
?>
    Please log in.
</p>
<?php echo $html->form('/users/login') ?>
<label>Username:</label>
<?php echo $html->input('User/username', array) ?>

<label>Password:</label>
<?php echo $html->password('User/password', array) ?>

<?php echo $html->submit('login') ?>
</form>
<?php echo $html->link('register', '/users/register') ?>
```

Your index view might look something like Listing 2.

Listing 2. Index view

```
<p>Hello, <?php e($User['first_name'] . ' ' . $User['last_name']) ?></p>

<p>Your last login was on <?php e($last_login) ?></p>

<?php echo $html->link('knownusers', '/users/knownusers') ?> <?php echo
$html->link('logout', '/users/logout') ?>
```

Both of the views should look pretty straightforward. The index view just checks the session for the user's username and, if it's not set, sends him to log in. The login view doesn't set a specific error message, so someone trying to guess their way into the system doesn't know which parts are correct.

Your controller might look something like Listing 3.

Listing 3. Controller

```
<?php
class UsersController extends AppController
{
    function register()
    {
        $this->set('username_error', 'Username must be between 6 and 40 characters.');
```

```
        if (!empty($this->data))
        {
            if ($this->User->validates($this->data))
            {
                if ($this->User->findByUsername($this->data['User']['username']))
                {
                    $this->User->invalidate('username');
```

```
                    $this->set('username_error', 'User already exists.');
```

```
                } else {
                    $this->data['User']['password'] = md5($this->data['User']['password']);
                    $this->User->save($this->data);
                    $this->Session->write('user', $this->data['User']['username']);
                    $this->redirect('/users/index');
```

```
                }
            } else {
                $this->validateErrors($this->User);
            }
        }
    }

    function knownusers()
    {
        $this->set('knownusers', $this->User->findAll(null, array('id', 'username',
'first_name', 'last_name', 'last_login'), 'id DESC'));
```

```
    }

    function login()
    {
        $this->set('error', false);
        if ($this->data)
        {
            $results = $this->User->findByUsername($this->data['User']['username']);
            if ($results && $results['User']['password'] ==
md5($this->data['User']['password']))
            {
                $this->Session->write('user', $this->data['User']['username']);
                $this->Session->write('last_login', $results['User']['last_login']);
                $results['User']['last_login'] = date("Y-m-d H:i:s");
                $this->User->save($results);
                $this->redirect('/users/index');
```

```
            } else {
                $this->set('error', true);
            }
        }
    }

    function logout()
    {
        $this->Session->delete('user');
```

```
        $this->redirect('/users/login');
```

```
    }

    function index()
    {
        $username = $this->Session->read('user');
```

```
        if ($username)
        {
            $results = $this->User->findByUsername($username);
            $this->set('User', $results['User']);
            $this->set('last_login', $this->Session->read('last_login'));
```

```
        } else {
            $this->redirect('/users/login');
```

```

    }
  }
?>

```

The use of `md5()` to hash passwords and compare their hashed values means you don't have to store plaintext passwords in the database -- as long as you hash the passwords before you store them. The logout action doesn't need a view; it just needs to clear the values you put into session.

It's OK if your solutions don't look exactly like these. If you didn't get to your own solutions, update your code using the above so that you will be ready to complete the rest of this tutorial.

Section 3. Scaffolding

Right now, Tor doesn't do a whole lot. It lets people register, log in, and see who else is registered. Now what it needs is the ability for users to enter some products into the catalog or view some products from other users. A good way to get a jumpstart on this is to use scaffolding.

Scaffolding is a concept that comes from Ruby on Rails (see [Resources](#)). It's an excellent way to get some database structure built quickly to prototype the application, without writing a bunch of throwaway code. But scaffolding, as the name implies, is something that should be used to help build an application, not something to build an application around. Once the application logic starts to get complicated, the scaffolding may need to get replaced by something more solid.

Setting up the product tables

Scaffolding works by examining the database tables and creating the basic types of elements normally used with a table: lists, add/delete/edit buttons, etc. To start, you need some tables to hold product information and dealer information.

Listing 4. Creating tables to hold product information

```

CREATE TABLE 'products' (
  'id' INT( 10 ) NOT NULL AUTO_INCREMENT ,
  'title' VARCHAR( 255 ) NOT NULL ,
  'dealer_id' INT( 10 ) NOT NULL ,
  'description' blob NOT NULL ,
  PRIMARY KEY ( 'id' )
) TYPE = MYISAM ;

CREATE TABLE 'dealers' (
  'id' INT( 10 ) NOT NULL AUTO_INCREMENT ,
  'title' VARCHAR( 255 ) NOT NULL ,
  PRIMARY KEY ( 'id' )
) TYPE = MYISAM ;

```

Additionally, it will be helpful for this demonstration to insert some data into the dealers table.

```
INSERT INTO dealer (title)
VALUES ('Tor Johnson School Of Drama'), ('Chriswell\'s Psychic Friends')
```

An important note about scaffolding: Remember that note from setting up the database about foreign keys following the format `singular_id` like `user_id` or `winner_id`? In CakePHP, scaffold will expect that any field ending in `_id` is a foreign key to a table with the name of whatever precedes the `_id` -- for example, scaffolding will expect that `dealer_id` is a foreign key to the table `dealers`.

Setting up the product model

The products functionality represents a whole new set of models, views, and controllers. You'll need to create them as you did in Part 1. Create your product model in `app/models/product.php`.

Listing 5. Creating a product model

```
<?php
class Product extends AppModel
{
    var $name = 'Product';
    var $belongsTo = array ('Dealer' => array(
        'className' => 'Dealer',
        'conditions'=>,
        'order'=>,
        'foreignKey'=>'dealer_id')
    );
}
```

You'll notice the `$belongsTo` variable. This is what's known as a model association.

Model associations

Model associations tell a model that it relates in some way to another model. Setting up proper associations between your models will allow you to deal with entities and their associated models as a whole, rather than individually. In CakePHP, there are four types of model associations:

hasOne

The `hasOne` association tells the model that each entity in the model has one corresponding entity in another model. An example of this would be a user entity's corresponding profile entity (assuming a user is only permitted one profile).

hasMany

The `hasMany` association tells the model that each entity in the model has several corresponding entities in another model. An example of this would be a category model having many things that belong to the category (posts, products, etc.). In the case of Tor, a dealer entity has many products.

belongsTo

This tells a model that each entity in the model points to an entity in another model. This is the opposite of `hasOne`, so an example would be a profile entity pointing back to one corresponding user entity.

hasAndBelongsToMany

This association indicates that an entity has many corresponding entities in another model and also points back to many corresponding entities in another model. An example of this might be a recipe. Many people might like the recipe, and the recipe would have several ingredients.

The `belongsTo` variable in this case indicates that each product in the products table "belongs to" a particular dealer.

Creating the dealer model

As the association implies, a dealer model is also required. The dealer model will get used later in Tor to build out the functionality to define dealerships. Whereas the product model had an association of `belongsTo` pointing at dealer, the dealer model has an association to product of `hasMany`.

Listing 6. The dealer model has an association to product of `hasMany`

```
<?php
class Dealer extends AppModel
{
    var $name = 'Dealer';
    var $hasMany = array ( 'Product' => array(
        'className' => 'Product',
        'conditions'=>,
        'order'=>,
        'foreignKey'=>'dealer_id')
    );
}
```

You can skip adding data validation for now, but as the application evolves, you may get ideas for different types validation to add.

Creating the products controller

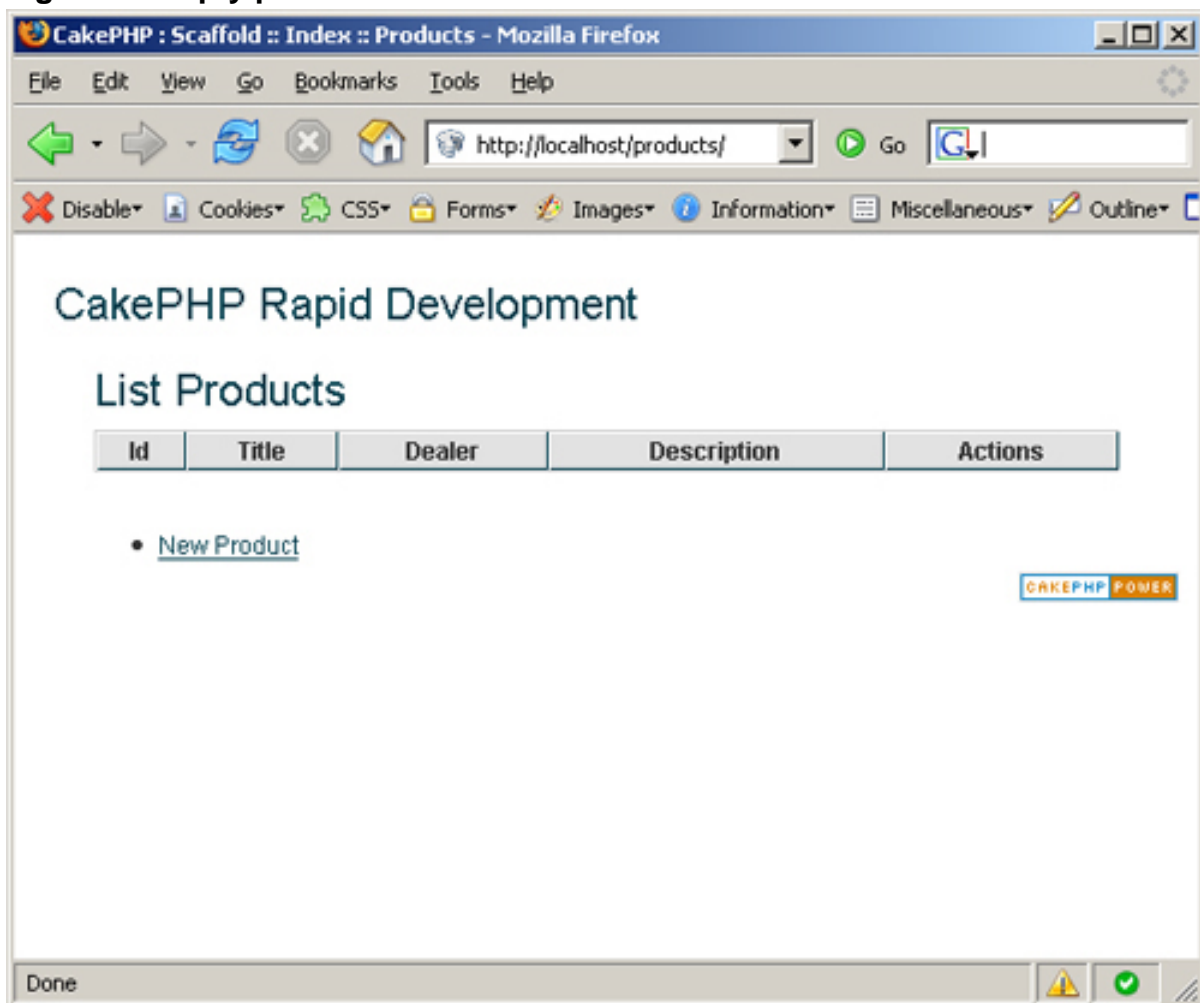
You've built and associated the models for product and dealer. Now Tor knows how the data is interrelated. Next, make your controller in `app/controllers/products_controller.php` -- but this time, add the class variable `$scaffold`.

Listing 7. Adding a class variable to your controller

```
<?php
class ProductsController extends AppController
{
    var $scaffold;
}
?>
```

Save the controller, then visit <http://localhost/products> (yes -- without creating any views, or a Dealer controller). You should see something like Figure 1.

Figure 1. Empty product list



It's really that simple. Just like that, you have an interface into your products table that lets you add, edit, delete, list, slice, and julienne your products.

Try adding a product. You should be prompted to enter a title and a description for the product, as well as select a dealer. Did that list of dealers look familiar? It should have; you inserted them into the dealer table just after you created it. Scaffolding recognized the table associations as you defined them, and auto-generated that drop-down dealer list for you.

Now go back and look at the amount of code you wrote to get all of this functionality.

How much easier could it get?

Section 4. Using the Bake code generator

It's not necessary to completely throw away everything that scaffolding gives you. By using Bake, the CakePHP code generator, you can generate a controller that contains functions that represent the scaffolding functionality and the views to go with it. For the products-related parts of Tor, this will be a huge time-saver.

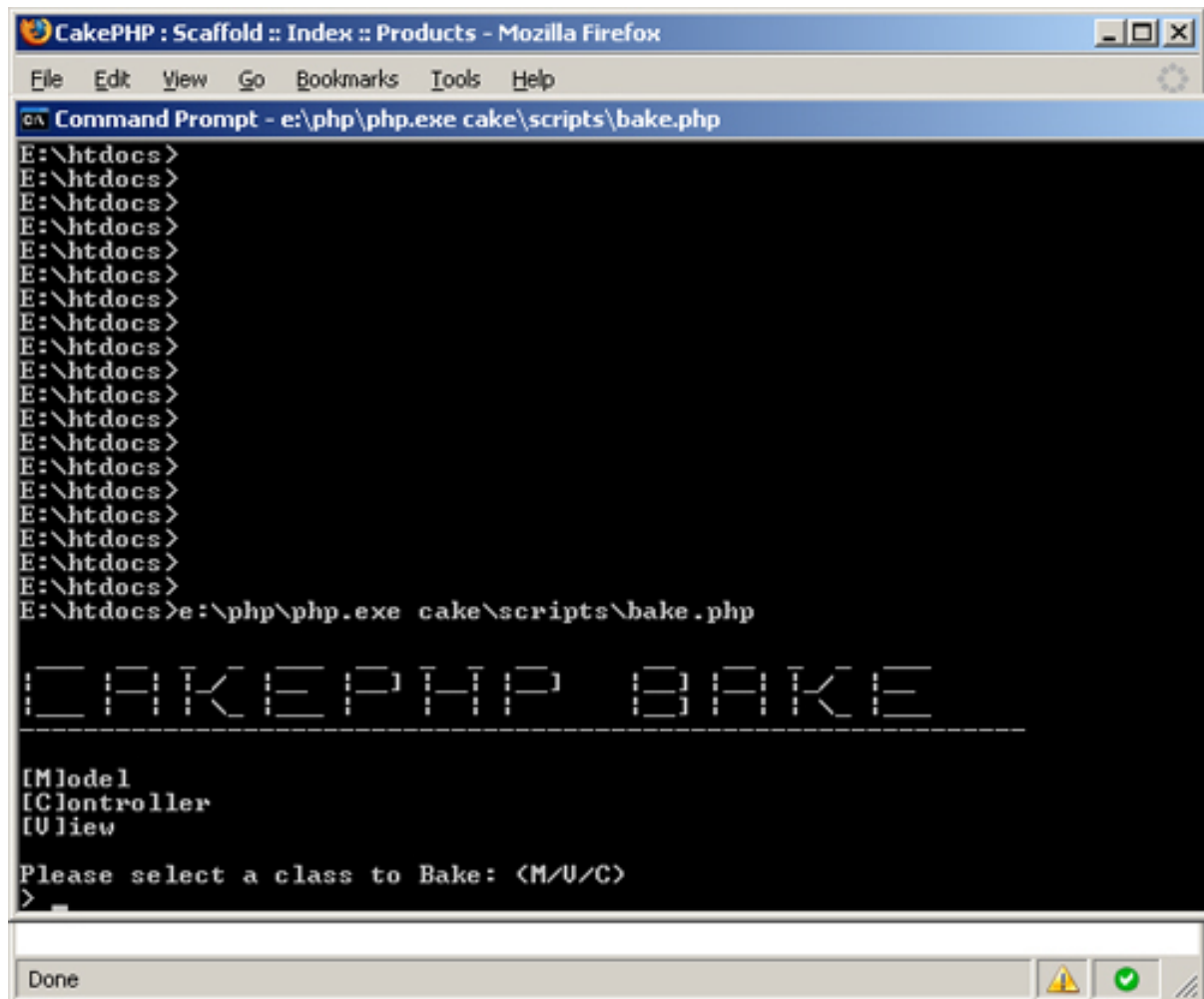
Before you proceed, make a copy of your existing app directory. Bake will overwrite the products controller, and you should always back up your files when an operation involves the word "overwrite" (or "copy," "delete," "format," or "voodoo"). You may want to up the value of `max_input_time` in the `php.ini` file when using Bake, especially in the beginning. If you take too long entering your information, Bake will time out.

If you have problems getting this to run, make sure that `php` is in your path -- if it's not, specifying the full path to your PHP executable should be sufficient.

Baking your products controller

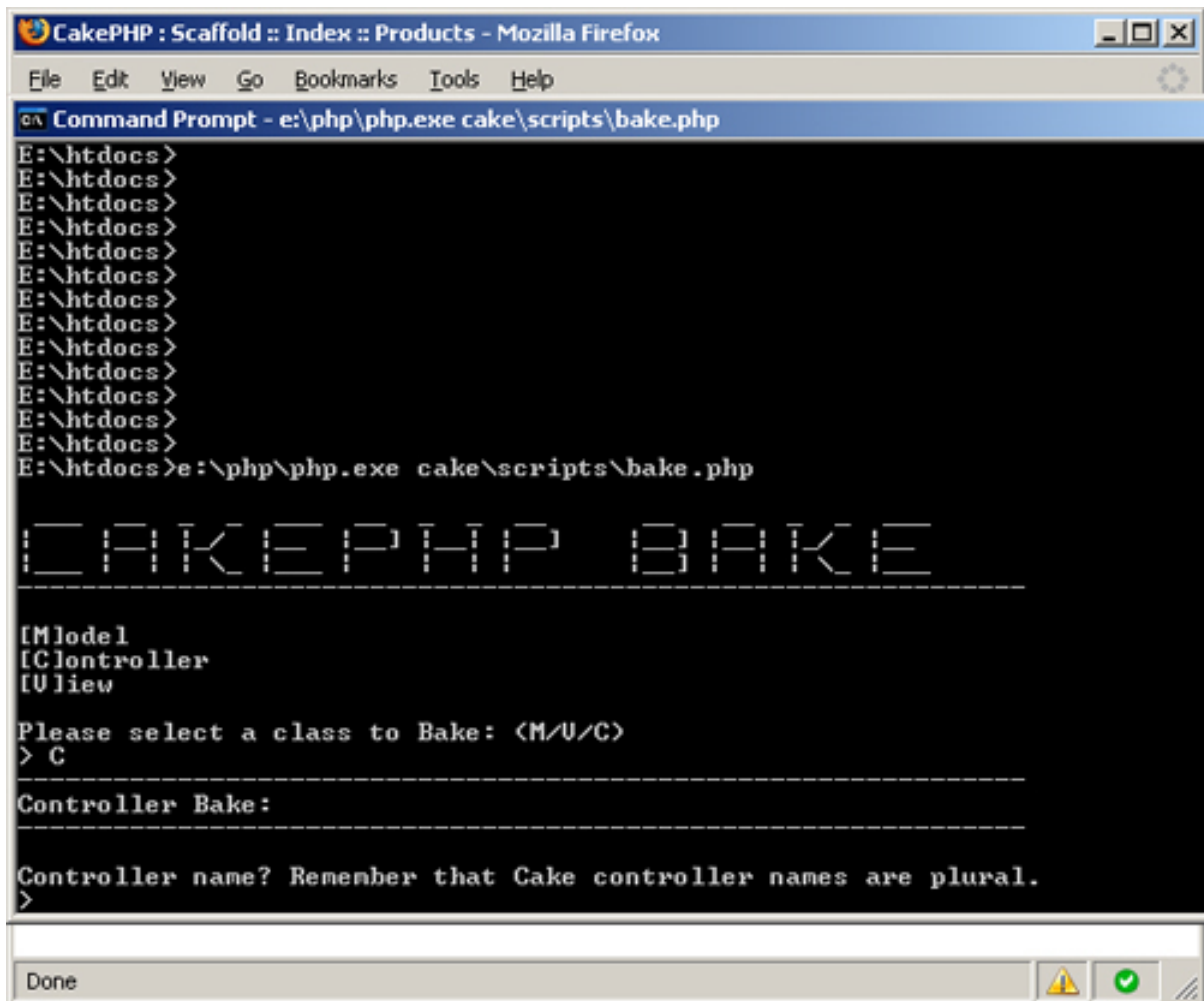
To use Bake, `cd` into the webroot where you installed CakePHP and execute the `bake.php` script: `php cake\scripts\bake.php`. You should be presented with a screen that looks like Figure 2.

Figure 2. Bake menu



For the Tor application, the model you've written should be fine, so let's start with the controller. Press **C** to select controller. You will be prompted to enter the name of the controller.

Figure 3. Controller name



```
CakePHP : Scaffold :: Index :: Products - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
on Command Prompt - e:\php\php.exe cake\scripts\bake.php
E:\htdocs>
E:\htdocs>
E:\htdocs>
E:\htdocs>
E:\htdocs>
E:\htdocs>
E:\htdocs>
E:\htdocs>
E:\htdocs>
E:\htdocs>
E:\htdocs>
E:\htdocs>
E:\htdocs>
E:\htdocs>e:\php\php.exe cake\scripts\bake.php

CAKEPHP BAKE
-----
[M]odel
[C]ontroller
[V]iew

Please select a class to Bake: (M/U/C)
> C

Controller Bake:
-----

Controller name? Remember that Cake controller names are plural.
>

Done
```

Keep in mind that in CakePHP, controller names are plural. Enter the name `products` and press **Enter** to continue. Bake will ask if you want to build the controller interactively. For now, press **N** to let Bake make all the decisions on its own, but later on, you should try building a controller interactively to get a feel for what Bake can do for you beyond this.

Bake should inform you that it's created the file `\app\controllers\products_controller.php`.

Figure 4. Controller created

```

CakePHP : Scaffold :: Index :: Products - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
on Command Prompt
E:\htdocs>
E:\htdocs>
E:\htdocs>
E:\htdocs>
E:\htdocs>e:\php\php.exe cake\scripts\cake.php

CAKEPHP BAKE
-----

[M]odel
[C]ontroller
[U]iew

Please select a class to Bake: <M/U/C>
> C

-----
Controller Bake:
-----

Controller name? Remember that Cake controller names are plural.
> products

Would you like bake to build your controller interactively?
Warning: Choosing no will overwrite Products controller if it exist. <y/n>
[y] > n

Creating file \app\controllers\products_controller.php
Wrote \app\controllers\products_controller.php
E:\htdocs>

```

That's all there is to it. Open up `products_controller.php` and take a look at what Bake did for you. It should look something like Listing 8.

Listing 8. `products_controller.php` after Bake

```

<?php
class ProductsController extends AppController
{
    //var $scaffold;
    var $name = 'Products';
    var $helpers = array('Html', 'Form' );

    function index() {
        $this->Product->recursive = 0;
        $this->set('products', $this->Product->findAll());
    }

    function add() {
        if(empty($this->data)) {
            $this->render();
        } else {
            $this->cleanUpFields();
            if($this->Product->save($this->data)) {
                $this->Session->setFlash('The Product has been saved');
                $this->redirect('/products/index');
            } else {
                $this->Session->setFlash('Please correct errors below.');
```

```

    }

    function edit($id) {
        if(empty($this->data)) {
            $this->data = $this->Product->read(null, $id);
        } else {
            $this->cleanUpFields();
            if($this->Product->save($this->data)) {
                $this->Session->setFlash('The Product has been saved');
                $this->redirect('/products/index');
            } else {
                $this->Session->setFlash('Please correct errors below.');
```

This controller Bake has generated contains the same basic functionality you got from including scaffolding, but now you can tweak it to fit your needs. Pretty slick. Ready to go again?

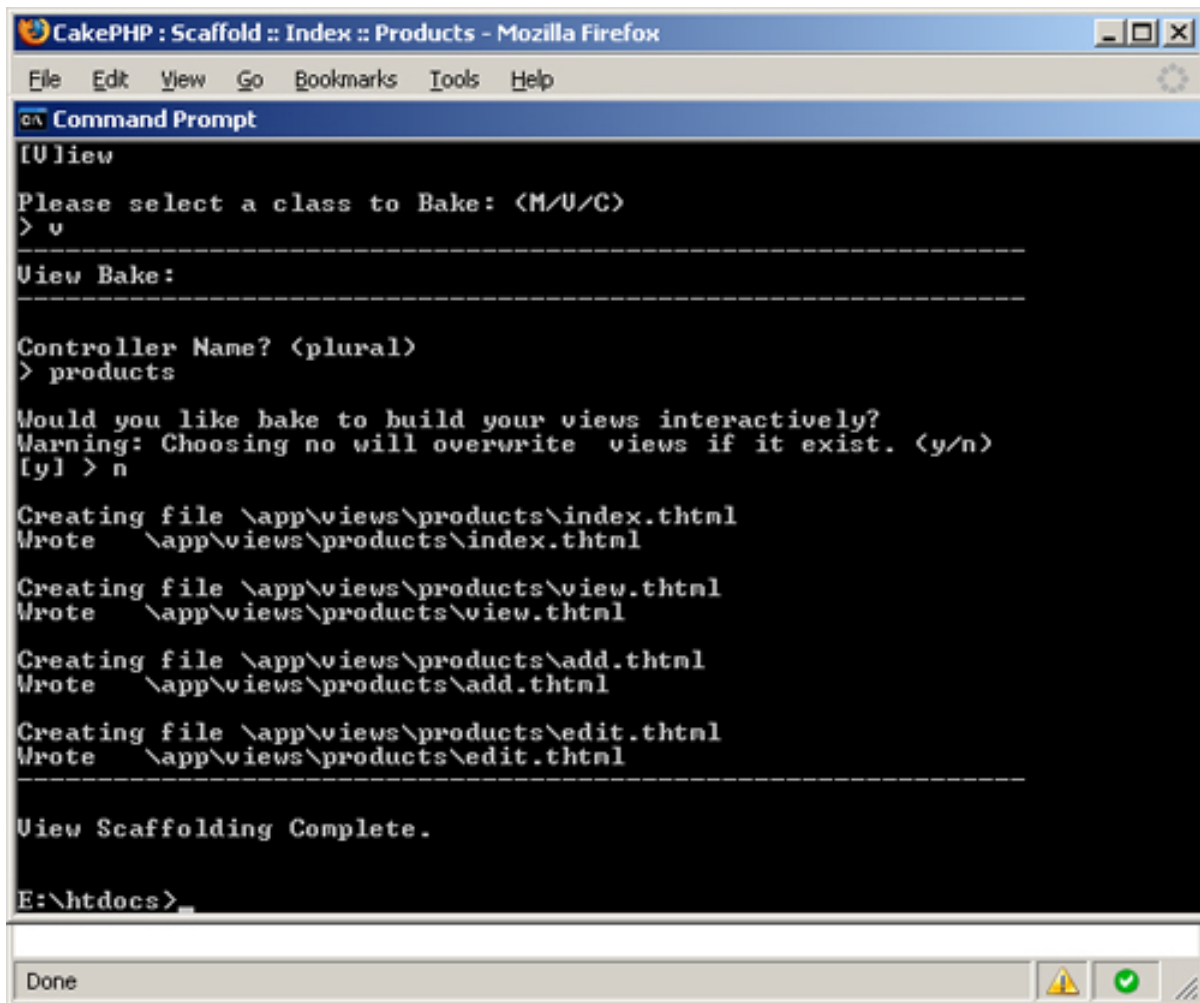
Baking your products views

Now that you've baked the products controller, all Tor needs is some product views. Bake will do those for you, too. Start as before: `php cake\scripts\bake.php`.

The initial Bake menu should look just like it did when you baked the controller. This time, though, it's time to bake some views. Press **V** to select views. You will be prompted to enter the name of the controller for which you are baking views. As before, enter the name `products` and press **Enter** to continue. Bake will ask if you want to build the views interactively. For now, press **N** to let Bake make all the decisions on its own. You can come back later and play around with interactive baking.

Bake should inform you that it's created the views.

Figure 5. Bake informs you that it has created the views



Open the index and take have a look. It should look something like Listing 9.

Listing 9. The index

```
<h2>List Products</h2>

<table cellpadding="0" cellspacing="0">
<tr>
    <th>Id</th>
    <th>Title</th>
    <th>Dealer Id</th>
    <th>Description</th>
    <th>Actions</th>
</tr>
<?php foreach ($products as $product): ?>
<tr>
    <td><?php echo $product['Product']['id'] ?></td>
    <td><?php echo $product['Product']['title'] ?></td>
    <td><?php echo $product['Product']['dealer_id'] ?></td>
    <td><?php echo $product['Product']['description'] ?></td>
    <td>
        <?php echo $html->link('View','/products/view/' .
        $product['Product']['id'])?>
        <?php echo $html->link('Edit','/products/edit/' .
        $product['Product']['id'])?>
        <?php echo $html->link('Delete','/products/delete/' .
        $product['Product']['id'],
        null, 'Are you sure you want to delete: id ' . $product['Product']['id'])?>
    </td>
</tr>
```

```
</tr>
<?php endforeach; ?>
</table>

<ul class="actions">
    <li><?php echo $html->link('New Product', '/products/add'); ?></li>
</ul>
```

Take a look in the those other views, as well. That's a whole lot of writing you didn't have to do. You'll be tweaking these views later in this tutorial to help lock down Tor.

Take it for a test drive

You've baked a controller and the necessary views for the products functionality. Take it for a spin. Start at <http://localhost/products> and walk through the various parts of the application. Add a product. Edit one. Delete another. View a product. It should look exactly like it did when you were using scaffolding.

Bake bigger and better

This isn't the end of what Bake can do for you by a long-shot. There will be a couple exercises at the end of the tutorial to let you venture out on your own. Keep in mind that the code generated by Bake is intended to be your starting point, not the end of your development work. But it's a tremendous time-saver if used properly.

Section 5. Access control lists

So far, Tor is wide open in terms of access. For example, anyone can add, edit, or delete products, etc. It's time to lock down some of this functionality. To do that, we will use CakePHP's ACL functionality.

What is an ACL?

An *ACL* is, in essence, a list of permissions. That's all it is. It is not a means for user authentication. It's not the silver bullet for PHP security. An ACL is just a list of who can do what.

The *who* is usually a user (but it could be something like a controller). The *who* is referred to as an access request object (ARO). The *do what* in this case is going to typically mean "execute some code". The *do what* is referred to as an access control object (ACO).

Therefore, an ACL is a list of AROs and the ACOs they have access to. Simple, right? It should be. But it's not.

As soon as the explanation departed from "it's a list of who can do what" and started throwing all those three-letter acronyms (TLAs) at you, things may have gone downhill. But an example will help.

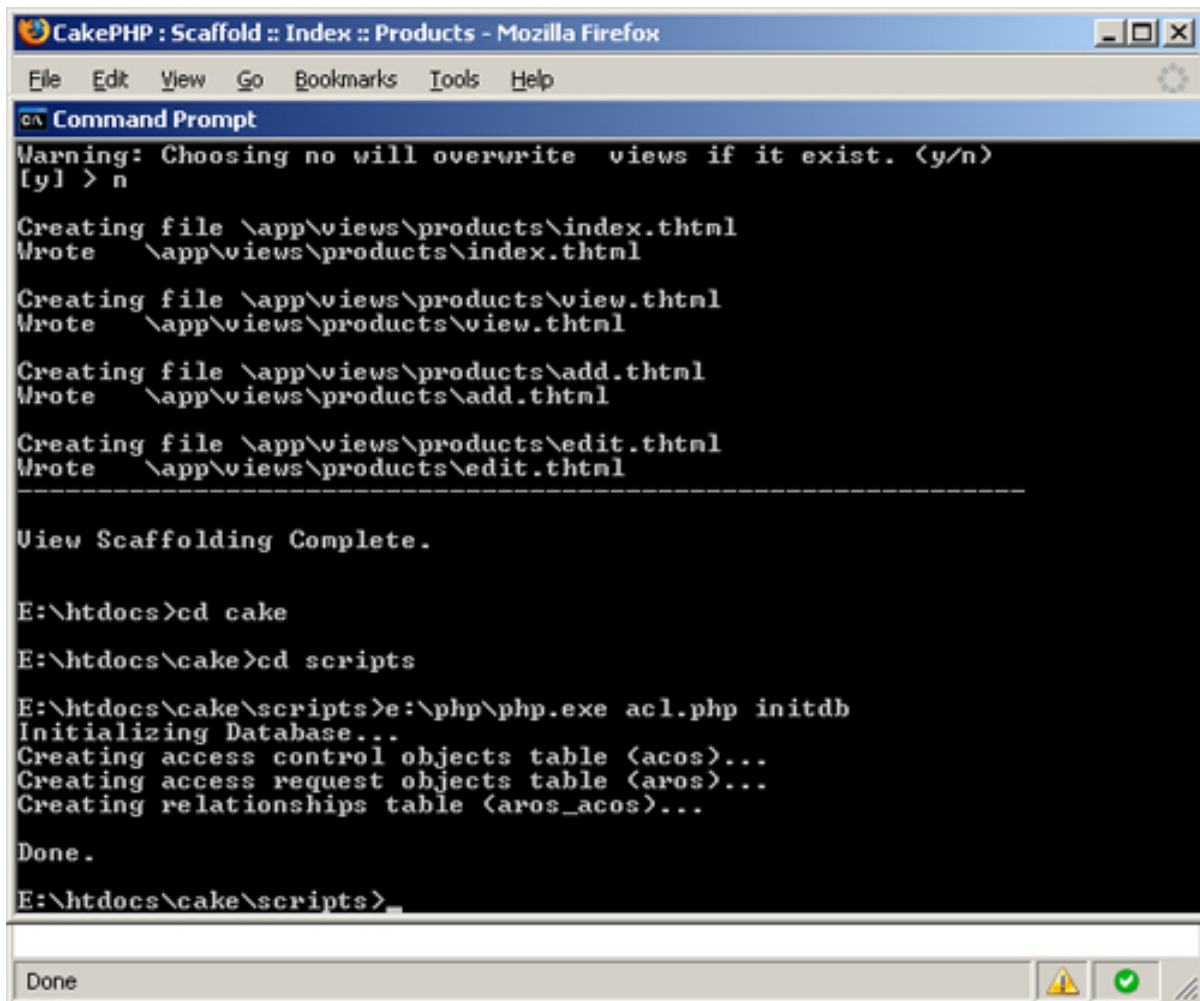
Imagine there's a party going on at some nightclub. Everyone who's anyone is there. The party is broken up into several sections -- there's a VIP lounge, a dance floor, and a main bar. And of course, a big line of people trying to get in. The big scary bouncer at the door checks a patron's ID, looks at the List, and either turns the patron away or lets him come into the section of the party to which he has been invited.

Those patrons are AROs. They are requesting access to the different sections of the party. The VIP lounge, dance floor, and main bar are all ACOs. The ACL is what the big scary bouncer at the door has on his clipboard. The big scary bouncer is CakePHP.

Creating the ACL table

CakePHP provides a command-line script to set up a database table to be used to store ACL information. Since you've already configured a database for Tor, running this script should be very straight-forward. At the command line, change into the `cake\scripts` and run the following command: `php acl.php initdb`. The `acl.php` file will tell you it has made three databases: `acos`, `aros`, and `aros_acos`.

Figure 6. `acl.php` output



```
CakePHP : Scaffold :: Index :: Products - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
on Command Prompt
Warning: Choosing no will overwrite views if it exist. (y/n)
[y] > n

Creating file \app\views\products\index.thtml
Wrote \app\views\products\index.thtml

Creating file \app\views\products\view.thtml
Wrote \app\views\products\view.thtml

Creating file \app\views\products\add.thtml
Wrote \app\views\products\add.thtml

Creating file \app\views\products\edit.thtml
Wrote \app\views\products\edit.thtml

-----

View Scaffolding Complete.

E:\htdocs>cd cake
E:\htdocs\cake>cd scripts
E:\htdocs\cake\scripts>e:\php\php.exe acl.php initdb
Initializing Database...
Creating access control objects table (acos)...
Creating access request objects table (aros)...
Creating relationships table (aros_acos)...

Done.

E:\htdocs\cake\scripts>
```

That's all it takes to get started. Now it's time to start defining your AROs and your ACOs.

Defining access request objects

So you have the ACL database tables. And you have an application that lets users self-register. How do you create the AROs for your users?

It makes the most sense to add this to the registration portion of the application. That way, when new users sign up, their corresponding ARO is automatically created for them. This does mean you'll have to manually create a couple AROs for the users you've already created, but CakePHP makes that easy, as well.

Defining groups

In CakePHP (and when using ACLs, in general), users can be assigned to groups for the purpose of assigning or revoking permissions. This greatly simplifies the task of permission management, as you do not need to deal with individual user permissions, which can grow into quite a task if your application has more than a few

users.

For Tor, you're going to define two groups. The first group, called users, will be used to classify everyone who has simply registered for an account. The second group, called dealers, will be used to grant certain users additional permissions within Tor.

You will create both of these groups using the CakePHP command-line `acl.php` script, much like you did to create the ACL database. To create the groups, execute the commands below from the `cake/scripts` directory.

```
php acl.php create aro 0 null Users
php acl.php create aro 0 null Dealers
```

After each command, CakePHP should display a message saying the ARO was created.

```
New Aro 'Users' created.
New Aro 'Dealers' created.
```

The parameters you passed in (for example, '0 null Users') are `link_id`, `parent_id`, and `alias`. The `link_id` parameter is a number that would normally correspond to the database ID of, for example, a user. Since these groups will never correspond to a database record, you passed in 0. The `parent_id` parameter would correspond to a group to which the ARO should belong. As these groups are top-level, you passed in null. The `alias` parameter is a string used to refer to the group.

Adding ARO creation to registration

Adding ARO creation to the user registration piece of Tor isn't hard. It's just a matter of including the right component and adding a couple lines of code. To refresh your memory, the `register` function from `users_controller.php` should look something like Listing 10.

Listing 10. Adding ARO creation to Tor user registration

```
function register()
{
    $this->set('username_error', 'Username must be between 6 and 40 characters.');
```

```
    if (!empty($this->data))
    {
        if ($this->User->validates($this->data))
        {
            if ($this->User->findByUsername($this->data['User']['username']))
            {
                $this->User->invalidate('username');
```

```
                $this->set('username_error', 'User already exists.');
```

```
            } else {
                $this->data['User']['password'] = md5($this->data['User']['password']);
                $this->User->save($this->data);
                $this->Session->write('user', $this->data['User']['username']);
                $this->redirect('/users/index');
```

```
            }
        }
    }
}
```

```

    } else {
        $this->validateErrors($this->User);
    }
}
}

```

To start using CakePHP's ACL component, you will need to include the component as a class variable.

Listing 11. Including the components as a class variable

```

<?php
class UsersController extends AppController
{
    var $components = array('Acl');
    ...
}

```

The `$components` array simply contains a list of CakePHP components to include, by name. There are other components available, such as the security component, which will be covered in a later tutorial. In this case, the only one you need is ACL.

Now you have access to all the functionality provided by the ACL component. Creating an ARO for your user is as simple as creating an ARO object and invoking the `create` method. This method takes the same three parameters you passed at the command line to create the groups: `link_id`, `parent_id`, and `alias`. To create the ARO for your user, you also need to know what the user's ID is once it has been saved. You can get this from `$this->User->id` after the data has been saved.

Putting it all together, your `register` function now might look something like Listing 12.

Listing 12. Register function

```

function register()
{
    $this->set('username_error', 'Username must be between 6 and 40 characters.');
```

```

    if (!empty($this->data))
    {
        if ($this->User->validates($this->data))
        {
            if ($this->User->findByUsername($this->data['User']['username']))
            {
                $this->User->invalidate('username');
```

```

                $this->set('username_error', 'User already exists.');
```

```

            } else {
                $this->data['User']['password'] = md5($this->data['User']['password']);
                if ($this->User->save($this->data))
                {
                    $aro = new Aro();
                    $aro->create($this->User->id, 'Users',
$this->data['User']['username']);
                    $this->Session->write('user', $this->data['User']['username']);
                    $this->redirect('/users/index');
```

```

                } else {
                    $this->flash('There was a problem saving this information', '/users/register');
```

```

                }
            }
        } else {

```

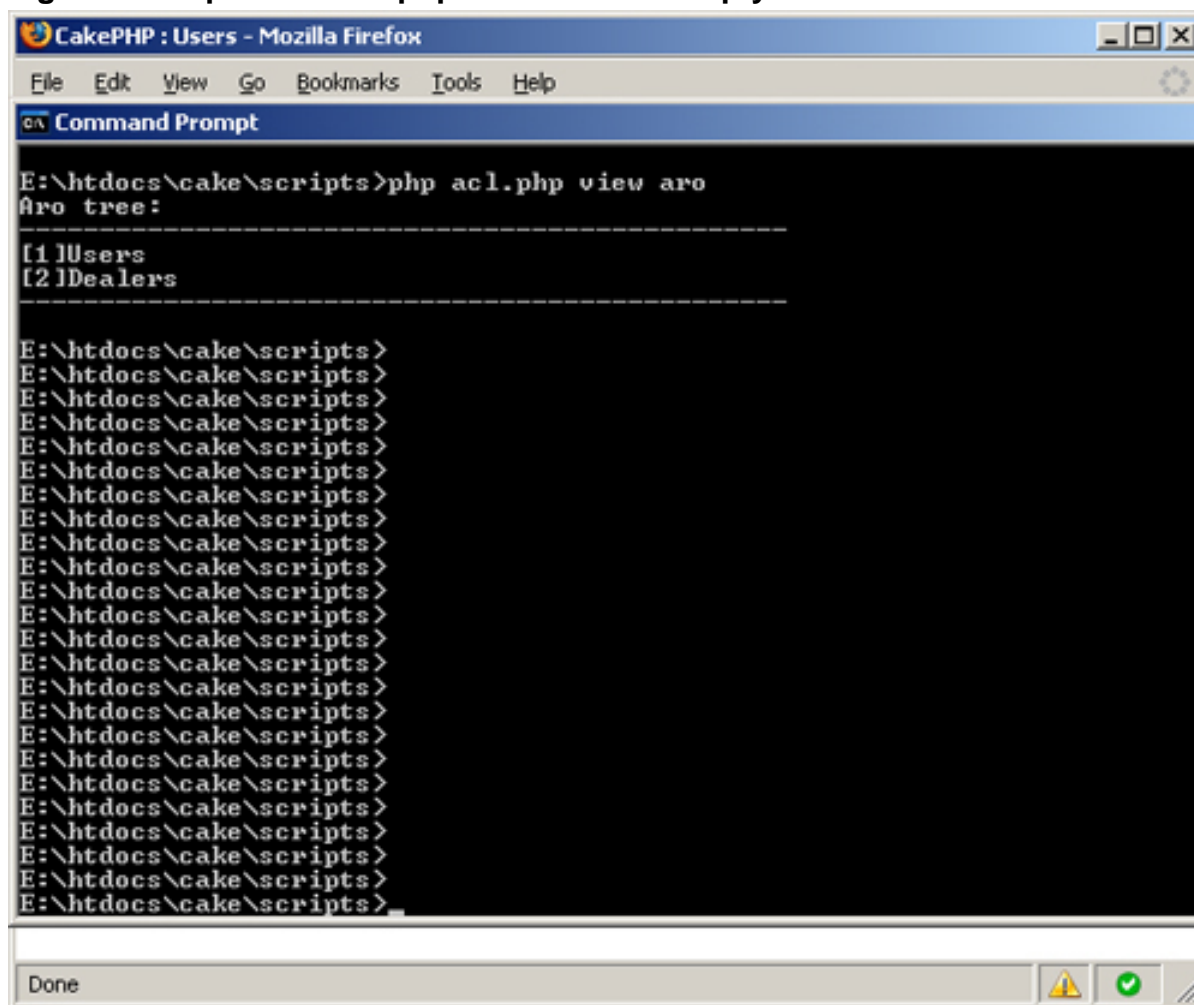
```
        $this->validateErrors($this->User);  
    }  
}  
}
```

You'll note that this `register` function includes a check to verify that the user data was saved successfully before proceeding with creation of the ARO.

Try it out

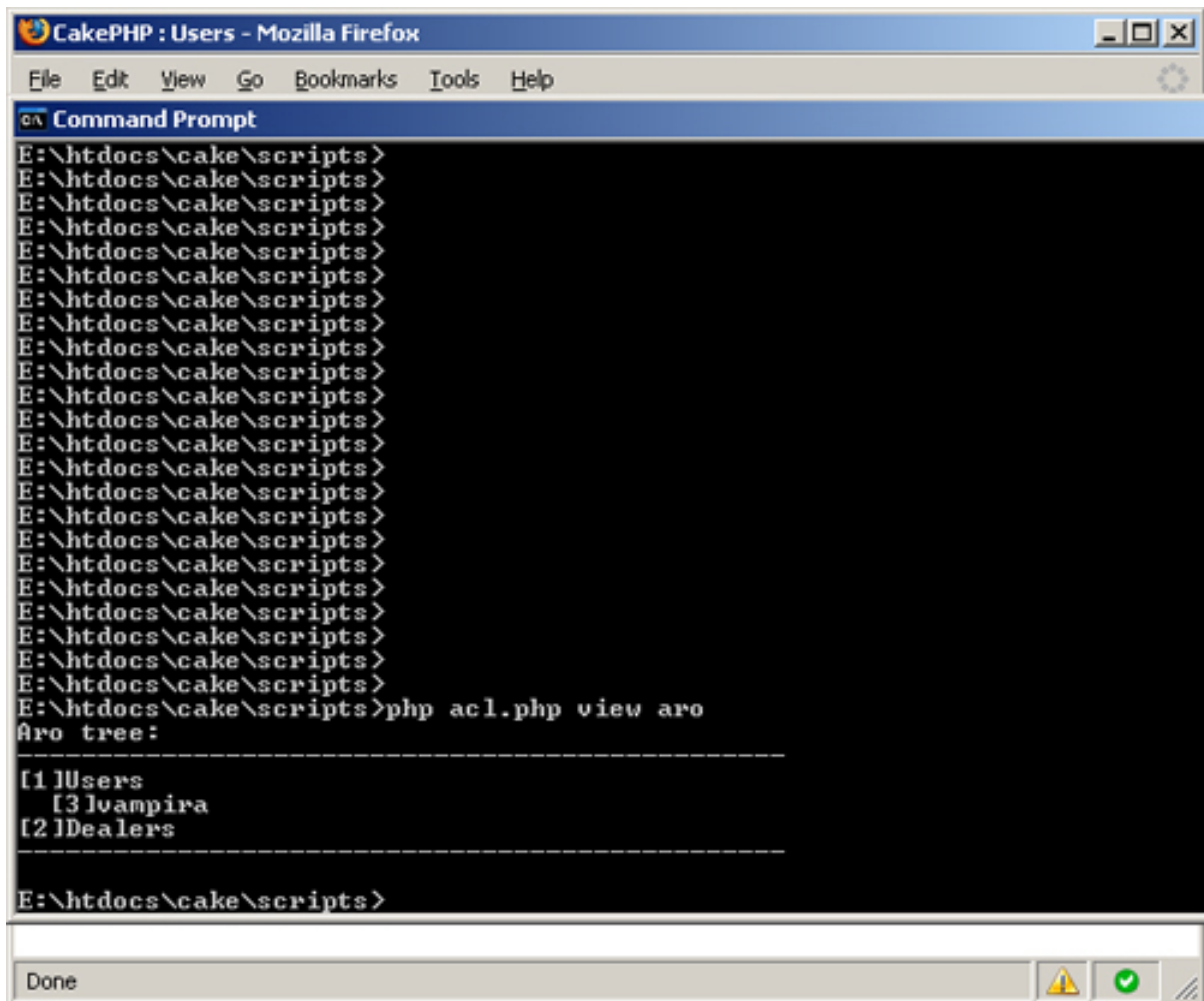
That should be all you need to get your AROs up and running. To verify, start back at the command line in `cake/scripts` and use the `acl.php` script to view the ARO tree: `php acl.php view aro`. Your output should look something like Figure 7.

Figure 7. Output from `acl.php view aro` with empty list



Now go to <http://localhost/users/register> and sign up a new user. Once you're done, rerun the `php acl.php view aro` command. Your output should look something like Figure 8.

Figure 8. Output from `acl.php view aro` with a user



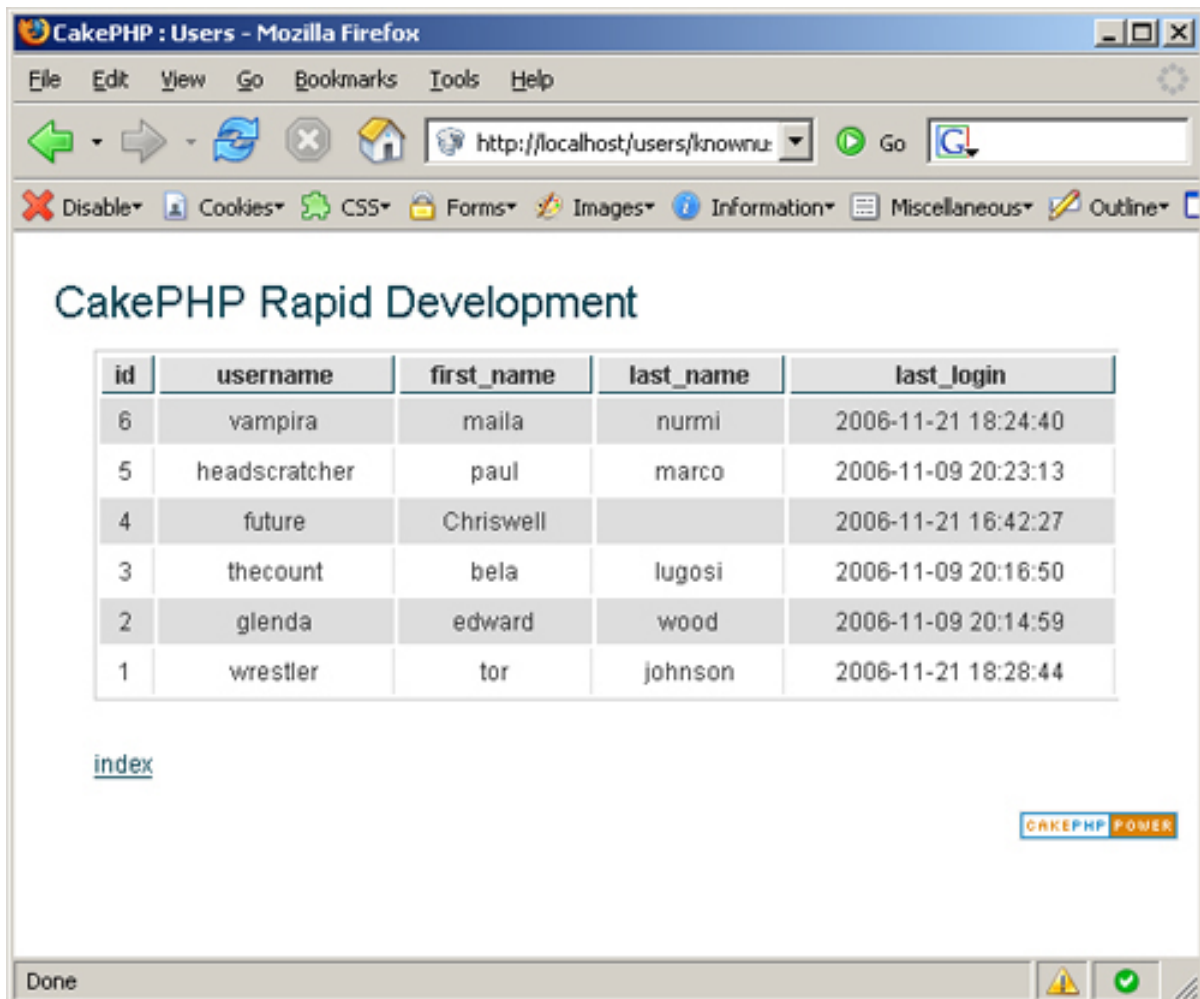
From now on, whenever someone registers for a new account, he will automatically have an ARO created for him. That ARO will belong to the users group.

Don't be confused by the numbers preceding the groups and users in the ARO list. That number is not the `link_id` for the ARO you've created.

Creating AROs for existing users

Now that new Tor users are getting their AROs created, you need to go back and create AROs for the existing users. You will do this with the command-line script, in almost exactly the same way you created the groups. Start by using that user you just created to visit <http://localhost/users/knownusers> to get a list of users that have been created.

Figure 9. Output from acl.php view aro with a user



CakePHP Rapid Development

id	username	first_name	last_name	last_login
6	vampira	maila	nurmi	2006-11-21 18:24:40
5	headscratcher	paul	marco	2006-11-09 20:23:13
4	future	Chriswell		2006-11-21 16:42:27
3	thecount	bela	lugosi	2006-11-09 20:16:50
2	glenda	edward	wood	2006-11-09 20:14:59
1	wrestler	tor	johnson	2006-11-21 18:28:44

[index](#)

CAKEPHP POWER

Done

Then, for each user, you need to execute the `create aro` command like you did for creating the groups. For `link_id`, specify the ID of the user. For `parent_id`, null. You will need to set the parent with a separate command. For `alias`, specify the username. For example, from Figure 9, to create an ARO for Tor Johnson, you would execute the following (again, from the `cake/scripts` directory): `php acl.php create aro 1 null wrestler`. And to set the parent for Tor, execute the `setParent` command, passing in the ID of the ARO and the parent ID: `php acl.php setParent aro wrestler Users`.

Make sure you run these commands for each user in your `knownusers` list, except for the user you created to test ARO creation during user registration. Be sure that you are specifying the right user ID and username for each user. When you're done, the results of `php acl.php view aro` should look something like Figure 10.

Figure 10. Output from `acl.php view ARO` with a user


```

CA 'manual human interface'

G:\htdocs\cake\scripts>g:\php\php acl.php setParent aro glenda Users
Node parent set to Users

G:\htdocs\cake\scripts>g:\php\php acl.php setParent aro thecount Users
Node parent set to Users

G:\htdocs\cake\scripts>g:\php\php acl.php setParent aro future Users
Node parent set to Users

G:\htdocs\cake\scripts>g:\php\php acl.php setParent aro headscratcher Users
Node parent set to Users

G:\htdocs\cake\scripts>g:\php\php acl.php view aro
Aro tree:
-----
[1]Users
  [8]headscratcher
  [7]future
  [6]thecount
  [5]glenda
  [4]wrestler
  [3]vampira
[2]Dealers
-----

G:\htdocs\cake\scripts>
G:\htdocs\cake\scripts>
G:\htdocs\cake\scripts>
G:\htdocs\cake\scripts>
G:\htdocs\cake\scripts>
G:\htdocs\cake\scripts>
G:\htdocs\cake\scripts>
G:\htdocs\cake\scripts>
G:\htdocs\cake\scripts>
G:\htdocs\cake\scripts>

```

You can get help on some of the other things that `acl.php` can do by running `php acl.php help` at the command line. The `acl.php` script is a helpful tool, but does not appear to be complete as of CakePHP V1.1.8.3544.

Section 6. Defining ACOs

Now that Tor has its AROs defined, it's time to identify and define your ACO. In this case, you're going to define ACOs to represent products, organizing the ACOs into groups as you did for your AROs.

Adding ACO definition to the products controller

You are going to add the initial ACO definition to the products controller in the `add` function, similar to what you did with ARO definition in user registration. Right now, the `add` function is exactly what Bake gave you. It should look something like Listing 13.

Listing 13. The add function

```
function add() {
    if(empty($this->data)) {
        $this->set('dealerArray',
        $this->Product->\
        Dealer->generateList());
        $this->render();
    } else {
        $this->cleanUpFields();
        if($this->Product->save($this->data)) {
            $this->Session->setFlash('The Product
has been saved');
            $this->redirect('/products/index');
        } else {
            $this->Session->setFlash('Please
correct errors below. ');
            $this->set('dealerArray',
        $this->Product->\
        Dealer->generateList());
        }
    }
}
</code>
```

Once again, CakePHP makes adding the definition for your ACOs very simple. You start by adding the `$components` class variable to the controller, like you did for the users controller.

Listing 14. Adding `$components` class variable to the controller

```
<?php
class ProductsController extends AppController
{
    var $components = array('Acl');
    ...
}
```

Creating an ACO looks almost exactly like creating an ARO. You create an ACO object and call the `create` method, passing in a `link_id` (in this case, a product ID), a `parent_id` (you will create groups representing your dealers here), and an `alias` (in this case, the product title by itself probably isn't a good idea because it's not unique; instead, you will amend the product id to the beginning of the product title). Putting these pieces into your `add` function, it should look something like Listing 15.

Listing 15. New add function

```
function add() {
    if(empty($this->data)) {
        $this->set('dealerArray', $this->Product->Dealer->generateList());
        $this->render();
    } else {
        $this->cleanUpFields();
        if($this->Product->save($this->data)) {
            $aco = new Aco();
            $product_id = $this->Product->getLastInsertID();
            $aco->create($product_id, $this->data['Product']['dealer_id'],
$product_id.'-'.$this->data['Product']['title']);
            $this->Session->setFlash('The Product has been saved');
            $this->redirect('/products/index');
        } else {

```



```
$this->Session->setFlash('Please correct errors below.');
```

```
$this->set('dealerArray', $this->Product->Dealer->generateList());
```

```
}
}
```

```
}
```

That should be all you need to auto-create the ACOs for the products created in Tor. Before you continue, you should create ACOs for the existing products and groups.

Adding ACO definitions for the dealers

The `acl.php` script can be used to define ACOs in much the same way it was used to define the AROs for existing users. It will be helpful to pull up that products list that CakePHP baked for you at <http://localhost/products>.

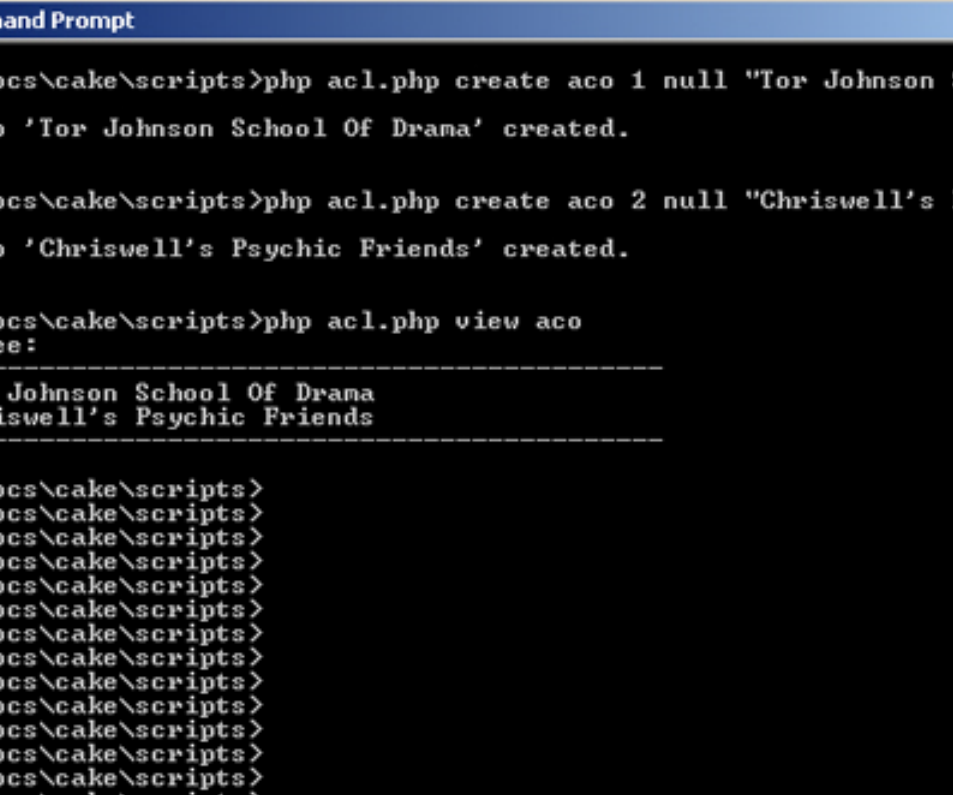
Once again, from the command line, in the `cake/scripts` directory, you will run some `create` commands.

Start by creating groups to represent the dealers you created way back when you created the dealer table. But this time, specify that you are creating an ACO and provide the dealer ID.

```
php acl.php create aco 1 null "Tor Johnson School Of Drama"
php acl.php create aco 2 null "Chriswell's Psychic Friends"
```

You can run `php acl.php view aco` to verify that the groups look as expected.

Figure 11. ACO dump with dealers no products



The screenshot shows a Windows XP desktop with two windows open. The top window is Mozilla Firefox, displaying the CakePHP Scaffold index page for 'Products'. The bottom window is a Windows Command Prompt, showing the execution of several PHP commands to create and view records in a database. The commands and their outputs are as follows:

```
E:\htdocs\cake\scripts>php acl.php create aco 1 null "Tor Johnson School Of Drama"
New Aco 'Tor Johnson School Of Drama' created.

E:\htdocs\cake\scripts>php acl.php create aco 2 null "Chriswell's Psychic Friends"
New Aco 'Chriswell's Psychic Friends' created.

E:\htdocs\cake\scripts>php acl.php view aco
Aco tree:
-----
[1]Tor Johnson School Of Drama
[2]Chriswell's Psychic Friends
-----

E:\htdocs\cake\scripts>
E:\htdocs\cake\scripts>
E:\htdocs\cake\scripts>
E:\htdocs\cake\scripts>
E:\htdocs\cake\scripts>
E:\htdocs\cake\scripts>
E:\htdocs\cake\scripts>
E:\htdocs\cake\scripts>
E:\htdocs\cake\scripts>
E:\htdocs\cake\scripts>
E:\htdocs\cake\scripts>
E:\htdocs\cake\scripts>
E:\htdocs\cake\scripts>
E:\htdocs\cake\scripts>
E:\htdocs\cake\scripts>
```

The taskbar at the bottom shows the 'Done' button and two icons: a yellow triangle with an exclamation mark and a green checkmark.

Next, delete the existing products from the products table. You should be able to do this by going to the products index (<http://localhost/products/index>) and clicking **Delete** next to each product.

There are a couple reasons to delete these products before continuing. A defect in `acl.php` at V1.1.8.3544 makes granting permissions at the command line problematic. You could write or modify an action to grant permissions on existing products, rather than deleting existing products. But because you have only created a couple of products thus far, deleting them is the easiest solution.

Don't test out that new product add function just yet. Now that you have ACOs created for your existing dealers and you've deleted the existing products, you're ready to proceed with setting up some permissions.

Section 7. Assigning permissions

Now Tor has a bunch of AROs representing users, and a bunch of ACOs

representing products, grouped by dealer. It's time to glue them together by defining some permissions.

How do permissions work?

You are going to specifically define who has the rights to work with the products. You will do this by explicitly allowing an ARO (in this case, a user) full rights on an ACO (in this case, a product), and an action. The actions can be read (meaning the user can view database information), create (the user can insert information into the database), update (the user can modify information), delete (the user can delete information from the database), or *, which means the user can perform all actions. Each action must be granted individually -- allowing delete does not imply allowing create or even view.

By default, once you check permissions for something, if there is no defined permission, CakePHP assumes DENY.

Defining policies

Defining permission policies is more than just writing and executing code. You need to think about what your ACL is actually trying to accomplish. Without a clear picture of what you are trying to protect from whom, you will find yourself constantly redefining your permissions.

Tor has users and products. For the purpose of this tutorial, you are going allow the user who created the product full permissions to edit and delete the product. Any user will be able to view the product unless explicitly denied access.

Adding permission definition to product add

Now that the existing products are taken care of, Tor needs to know how to assign permissions when a product is created. This can be accomplished by adding two lines to the controller. One line adds view permissions for the Users and another line adds full permissions for the creating user. Granting permissions looks something like this: `$this->Acl->allow(ARO, ACO, TYPE);`.

If you do not specify a TYPE (create, read, update, or delete), CakePHP will assume you are granting full permission. Your new add function in the products controller should look like this:

Listing 16. New add function in the products controller

```
function add() {
    if(empty($this->data)) {
        $this->set('dealerArray', $this->Product->Dealer->generateList());
        $this->render();
    } else {
        $dealer_id = $this->data['Product']['dealer_id'];
```

```

$product_alias = '-'.$this->data['Product']['title'];
$this->cleanUpFields();
if($this->Product->save($this->data)) {
    $product_id = $this->Product->id;
    $aco = new Aco();
    $product_alias = $product_id.$product_alias;
    $aco->create($product_id, $dealer_id, $product_alias);
    $this->Acl->allow('Users', $product_alias, 'read');
    $this->Acl->allow($this->Session->read('user'), $product_alias, '*');
    $this->Session->setFlash('The Product has been saved');
    $this->redirect('/products/index');
} else {
    $this->Session->setFlash('Please correct errors below.');
```

Try logging in as *wrestler* and adding a couple of products, just to see that nothing got broken along the way. You're almost done. You've defined your AROs, your ACOs, and you have assigned permissions. Now Tor needs check permissions when performing the various product-related actions.

Section 8. Putting your ACLs to work

You've laid all the pieces out. It's time to put your ACLs to work. When you're done, any user will be allowed to view products in Tor, but only the user who created the product will be able to edit or delete it.

You are going to add a couple lines to each action in the products controller. These lines will check the user for access and permit or deny the action based on the permissions.

Letting only users view products

Start with the view action. Add a line to check access to the product, displaying a message if the action is not allowed.

Listing 17. Adding a line to check access to the product

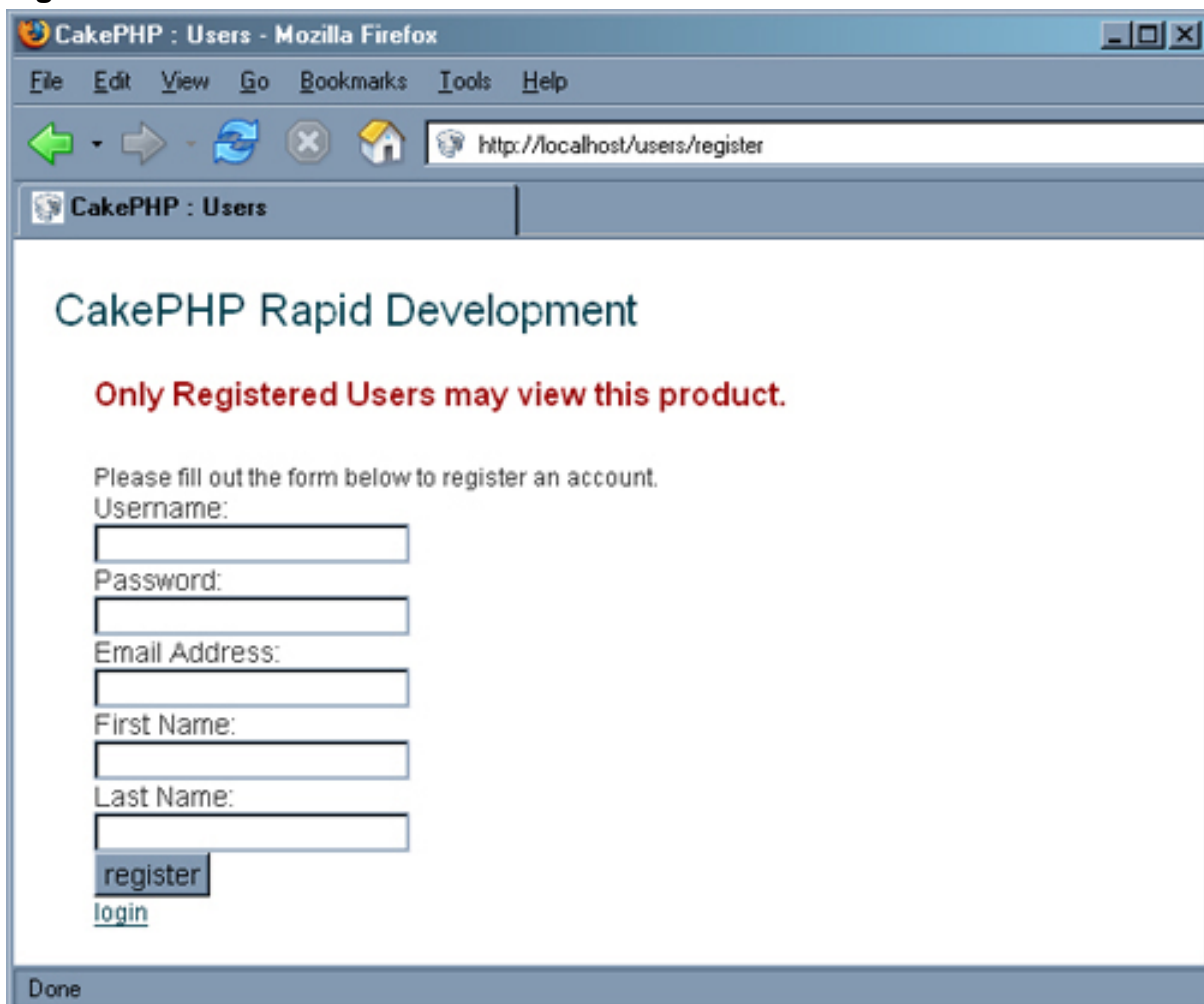
```

function view($id) {
    $product = $this->Product->read(null,
    $id);
    if
    ($this->Acl->check($this->Session->read('user'),
    \
    $id.'-'. $product['title'],
    'read'))
    {
        $this->set('product', $product);
    } else {
        $this->Session->setFlash('Only
    Registered Users \
```

```
    may view this product.');
```

Save the file, make sure you are logged out of Tor and visit the products list at <http://localhost/products>. When you click on any of the products, you should get redirected to the User Registration page.

Figure 12. Redirection



CakePHP : Users - Mozilla Firefox

File Edit View Go Bookmarks Tools Help

http://localhost/users/register

CakePHP : Users

CakePHP Rapid Development

Only Registered Users may view this product.

Please fill out the form below to register an account.

Username:

Password:

Email Address:

First Name:

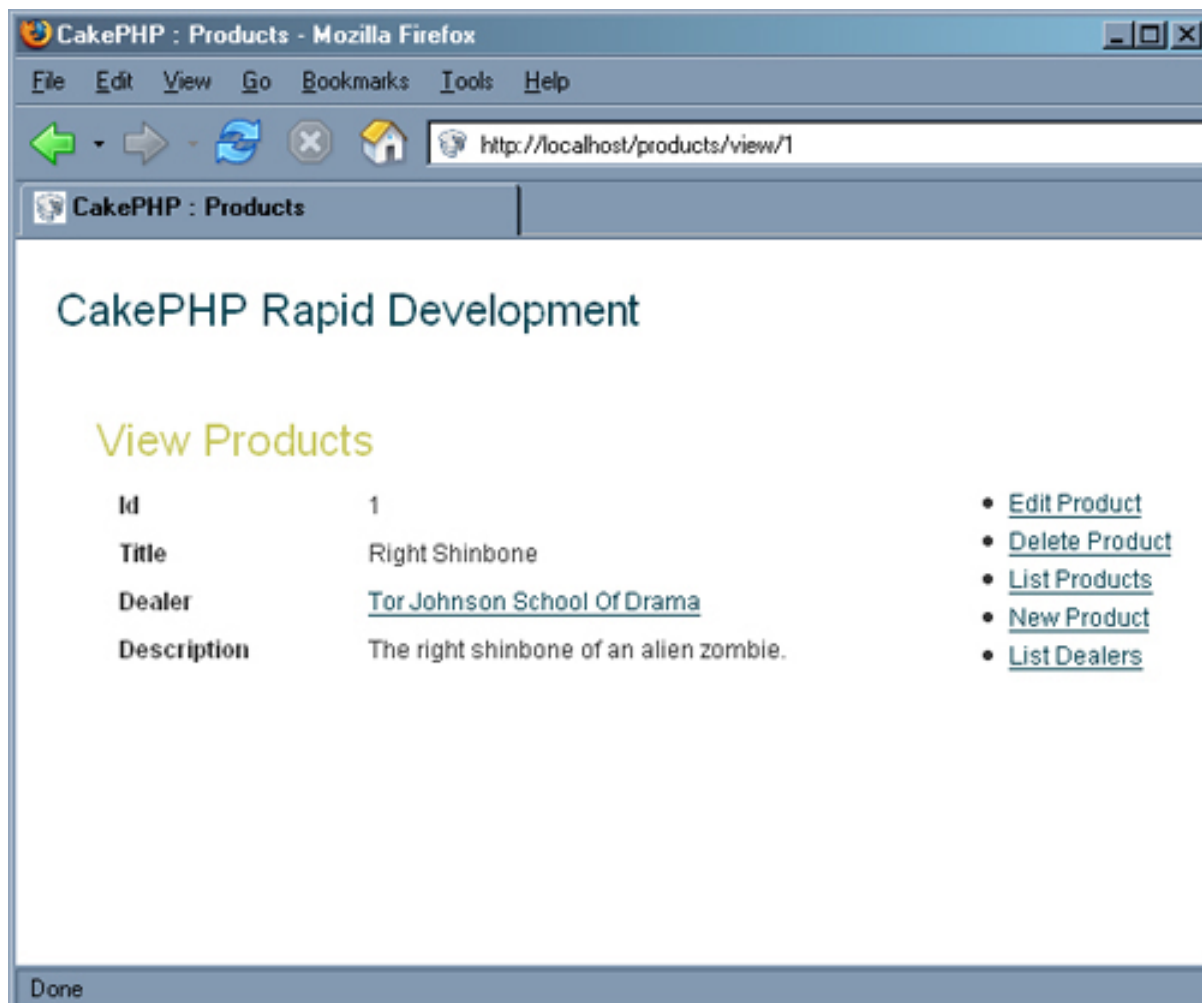
Last Name:

[login](#)

Done

Now log in using any account and try it again. This time you should be able to view the product.

Figure 13. Viewing the product



That tackles the first part of the permissions. Now you need to tell Tor to deny edit and delete access to anyone but the user who created the product.

Letting only the product creator edit or delete a product

The process is much the same for the edit and delete actions in the products controller.

Listing 18. The edit action

```
function edit($id) {
    $product = $this->Product->read(null, $id);
    if ($this->Acl->check($this->Session->read('user'),
        $id.'-'. $product['Product']['title'],
        'update'))
    {
        if(empty($this->data)) {
            $this->data = $this->Product->read(null, $id);
            $this->set('dealerArray', $this->Product->Dealer->generateList());
        } else {
            $this->cleanUpFields();
            if($this->Product->save($this->data)) {
                $this->Session->setFlash('The Product has been saved');
                $this->redirect('/products/index');
            } else {
```

```

        $this->Session->setFlash('Please correct errors below.');
```

```

    $this->set('dealerArray', $this->Product->Dealer->generateList());
    }
} else {
    $this->Session->setFlash('You cannot modify this product.');
```

```

    $this->redirect('/products/index');
}
}
}

```

For the delete controller, you should add an additional line -- one to delete the ACO for the product. Your delete action will look like Listing 19.

Listing 19. Delete action

```

function delete($id) {
    $product = $this->Product->read(null, $id);
    if ($this->Acl->check($this->Session->read('user'),
        $id.'-'.$product['Product']['title'], 'delete'))
    {
        if($this->Product->del($id)) {
            $aco = new Aco();
            $aco->delete($id.'-'.$product['Product']['title']);
            $this->Session->setFlash('The Product deleted: id '.$id.);
            $this->redirect('/products/index');
```

```

        }
    } else {
        $this->Session->setFlash('You cannot delete this product.');
```

```

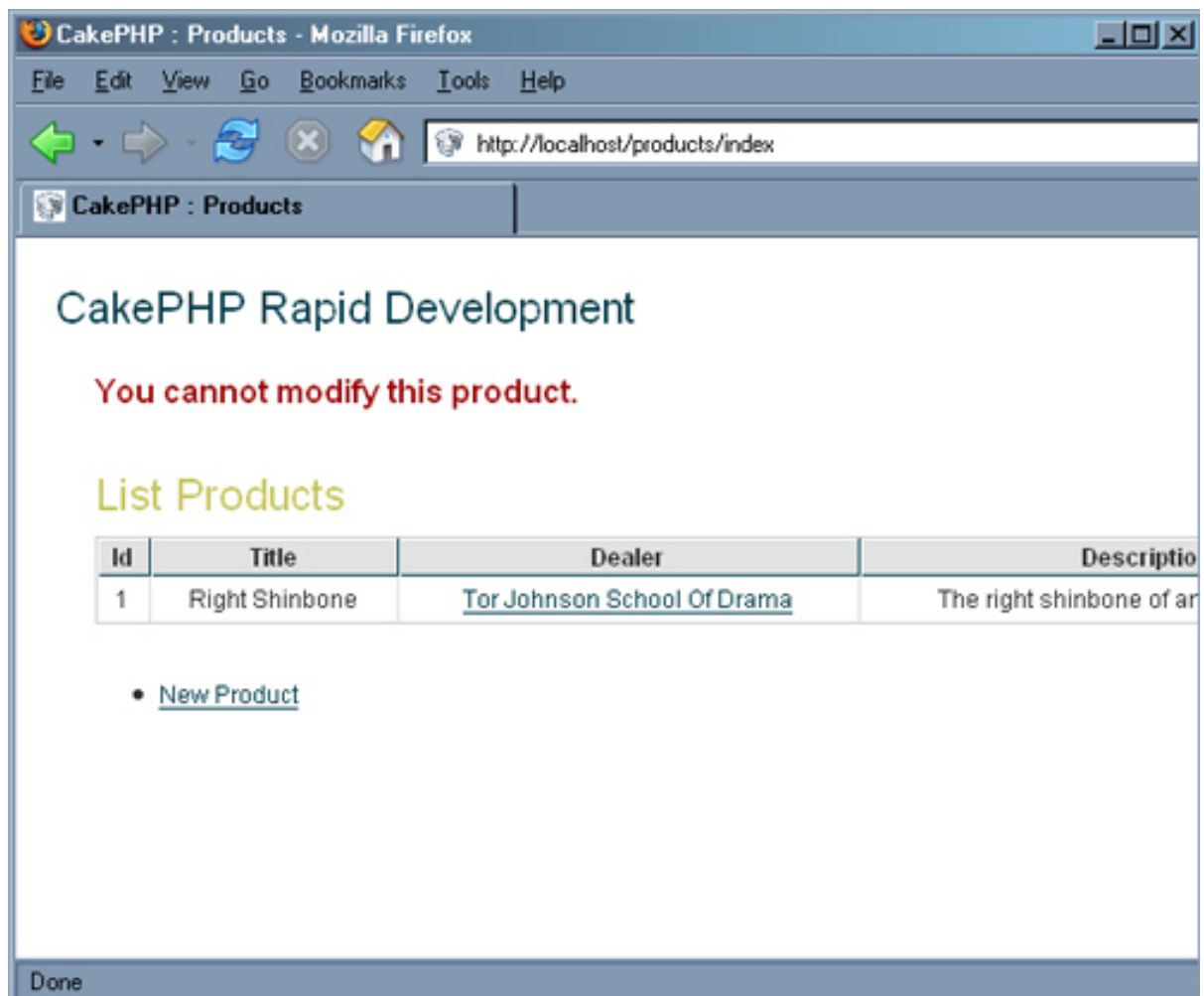
        $this->redirect('/products/index');
    }
}
}

```

You don't have to go through and revoke the corresponding permissions because CakePHP does it for you.

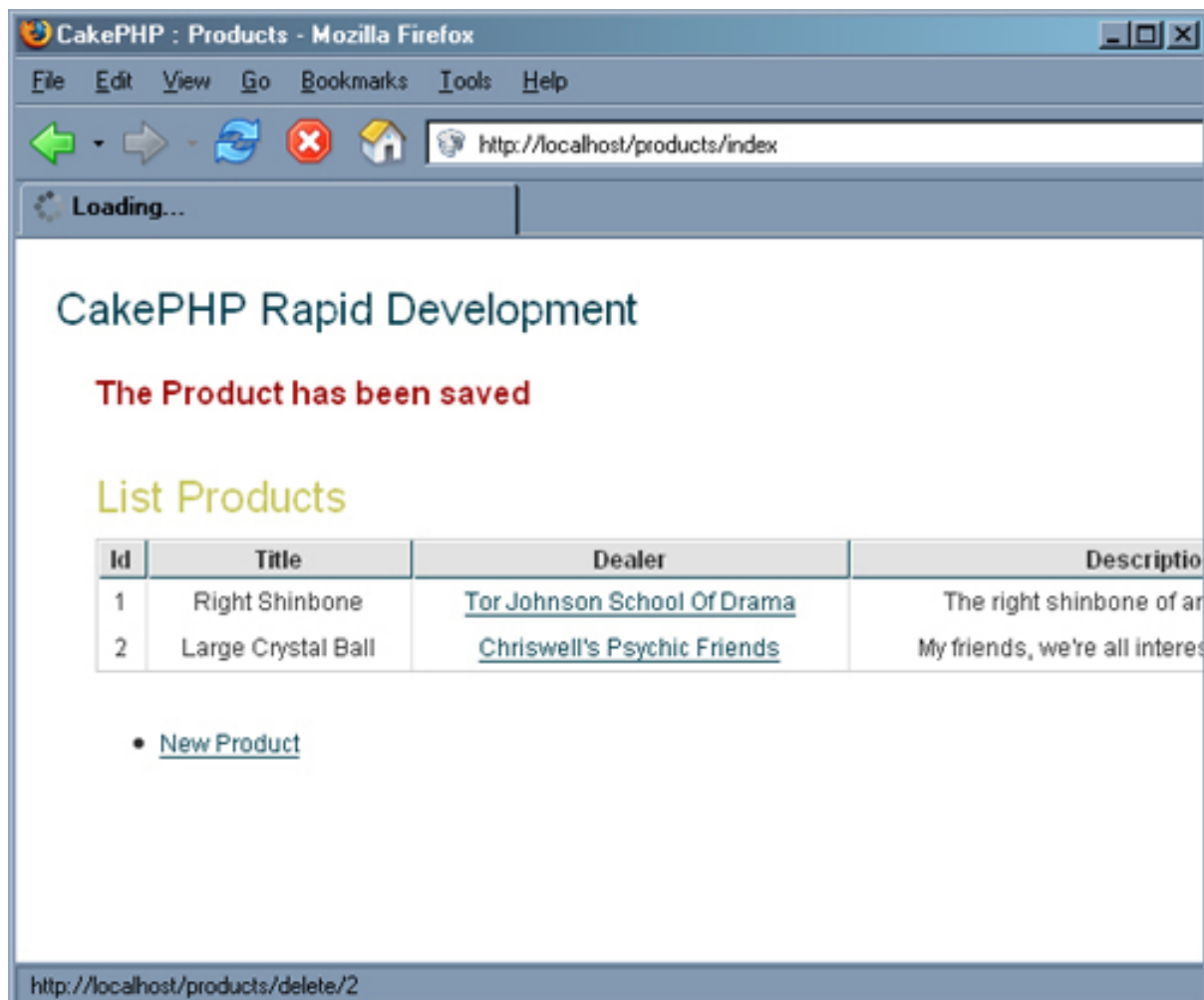
Save the products controller and try it out. Start by logging out at <http://localhost/users/logout>, then go back to your products list at <http://localhost/products/> and try to edit or delete a product. You should get directed back to the products list with a message.

Figure 14. Failed edit or delete



Now log in as the user *wrestler* and try to edit a product. Then delete it. You should have no trouble.

Figure 15. Successful edit or delete



Log out again at <http://localhost/users/logout> and log in as the user *future* and try to edit or delete another product, and you will find you are unable to. While you're here, create a new product, then try to modify or delete the product as another user.

Section 9. Filling in the gaps

With CakePHP, you can build out parts of your application quickly and easily, using scaffolding and Bake. Using ACLs, you can exercise a great deal of control over many aspects of your application. There's more that needs to be done for Tor. Here are some exercises to try.

Dealers

As you may have noticed from the products views that Bake built, there are links in the index view that point to dealers. Like you did with products, use Bake to build a controller and views for dealers. Don't build a model, as you already have one

defined and related to products.

Modify the dealer's add action to verify that the dealer name is unique.

ACLs

There's a bug in the `add` action for the products controller. It doesn't check to see who can create a product. This functionality should only be available to users. Fix the bug.

Once you have dealers built, using the ACL skills you have learned, protect all dealer functionality from anyone not belonging to the dealers group.

Once that is complete, using ACLs, allow any user to create a dealer. You will note that the ACOs that are created for products go into ACO groups representing the dealers. How would you set up ACLs so that any member of the dealership could change a product, but only the product creator could delete the product?

Views

In the products index view, come up with a way to only display **Edit** and **Delete** buttons for products the user can edit or delete.

Section 10. Summary

While scaffolding is a great way to get a quick look at your application, Bake is the way to go when it comes to getting some structure in place quickly. Using CakePHP's ACLs, you can exercise a great deal of control at a fairly granular level within your application. These are just a couple ways that CakePHP helps make your life easier and speed up your development.

[Part 3](#) shows how to use Sanitize, a handy CakePHP class, which helps secure an application by cleaning up user-submitted data.

Downloads

Description	Name	Size	Download method
Part 2 source code	os-php-cake2.source	616 KB zip	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](#), also known as the "Gang Of Four" book.
- 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

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.