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

Updated 22 Jan 2008

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

Editor's note: This series was originally published in 2006 and 2007. Since its publication, CakePHP developers made significant changes to CakePHP, which made this series obsolete. In response to these changes and the popularity of this series, the authors revised each of its five parts to make it compliant with the version of CakePHP available in January 2008.

This "Cook up Web sites fast with CakePHP" 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).
- 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 jump-start 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 V2.2.4 with `mod_rewrite` enabled.
2. PHP V4.3.2 or later (including PHP V5). This tutorial was written using

PHP V5.2.3

3. A supported database engine. this tutorial was written using MySQL V5.0.4

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 V1.2.0. 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 (isset($error)) {
  echo('Invalid Login.');
}
?>

<p>Please log in.</p>
<?php echo $form->create('User',
array('action' => 'login')); ?>

<?php
    echo $form->input('username');
    echo $form->input('password');
?>

<?php echo $form->end('Login');?>
<?php echo $html->link('Register',
array('action' => 'register')); ?>
```

Your index view might look something like Listing 2.

Listing 2. Index view

```
<p>Hello, <?php echo($user['first_name'] . ' ' . $user['last_name']); ?></p>

<?php echo $html->link('knownusers', array('action' => 'knownusers')); ?>

<?php echo $html->link('logout', array('action' => 'logout')); ?>
```

Both of the views should look pretty straightforward. The index view just checks the session for the user's user name 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 his 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
{
    var $name = 'Users';
    var $helpers = array('Html', 'Form' );

    function register()
    {
if (!empty($this->data))
{
$this->data['User']['password'] = md5($this->data['User']['password']);
if ($this->User->save($this->data))
{
$this->Session->setFlash('Your registration information was accepted');
$this->Session->write('user', $this->data['User']['username']);
$this->redirect(array('action' => 'index'), null, true);
} else {
$this->data['User']['password'] = '';
$this->Session->setFlash('There was a problem saving this information');
}
}
    }

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

    function login()
    {
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->redirect(array('action' => 'index'), null, true);
} else {
$this->set('error', true);
}
}
    }

    function logout()
    {
$this->Session->delete('user');
$this->redirect(array('action' => 'login'), null, true);
    }
```

```
    function index()
    {
$username = $this->Session->read('user');
if ($username)
{
$results = $this->User->findByUsername($username);
$this->set('user', $results['User']);
} else {
$this->redirect(array('action' => 'login'), null, true);
}
    }

}
?>
```

The use of `md5()` to hash passwords and compare their hashed values means you don't have to store plain-text 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 jump-start 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 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 you start wishing the scaffolding acted differently, it's time to pull it down.

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, the stuff normally called Create, Read, Update, Delete (CRUD). 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 dealers (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

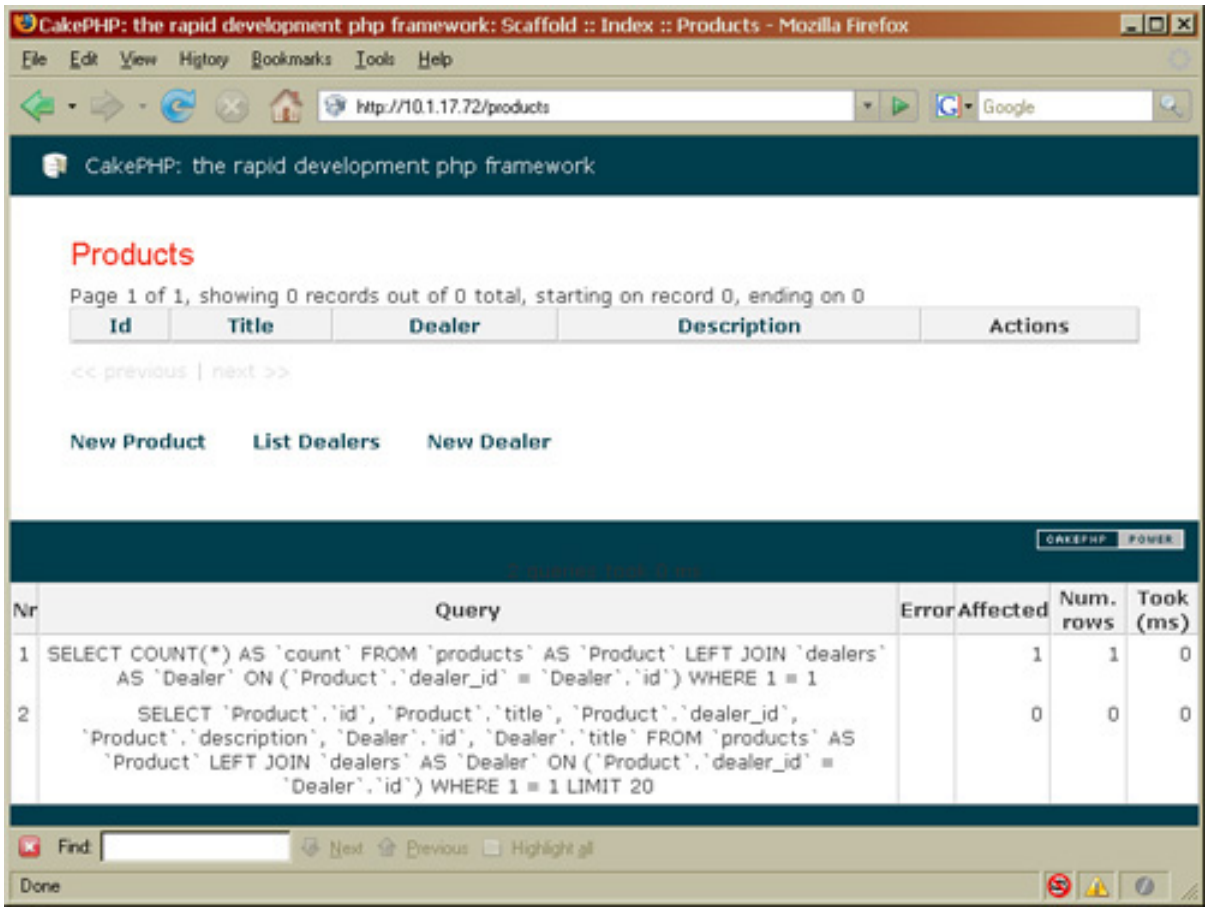
You've built and associated the models for product and dealer. Now Tor knows how the date is interrelated. Next, make your controller in `app/controllers/products_controller.php` — but this time, add the class variable `$scaffold`.

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.

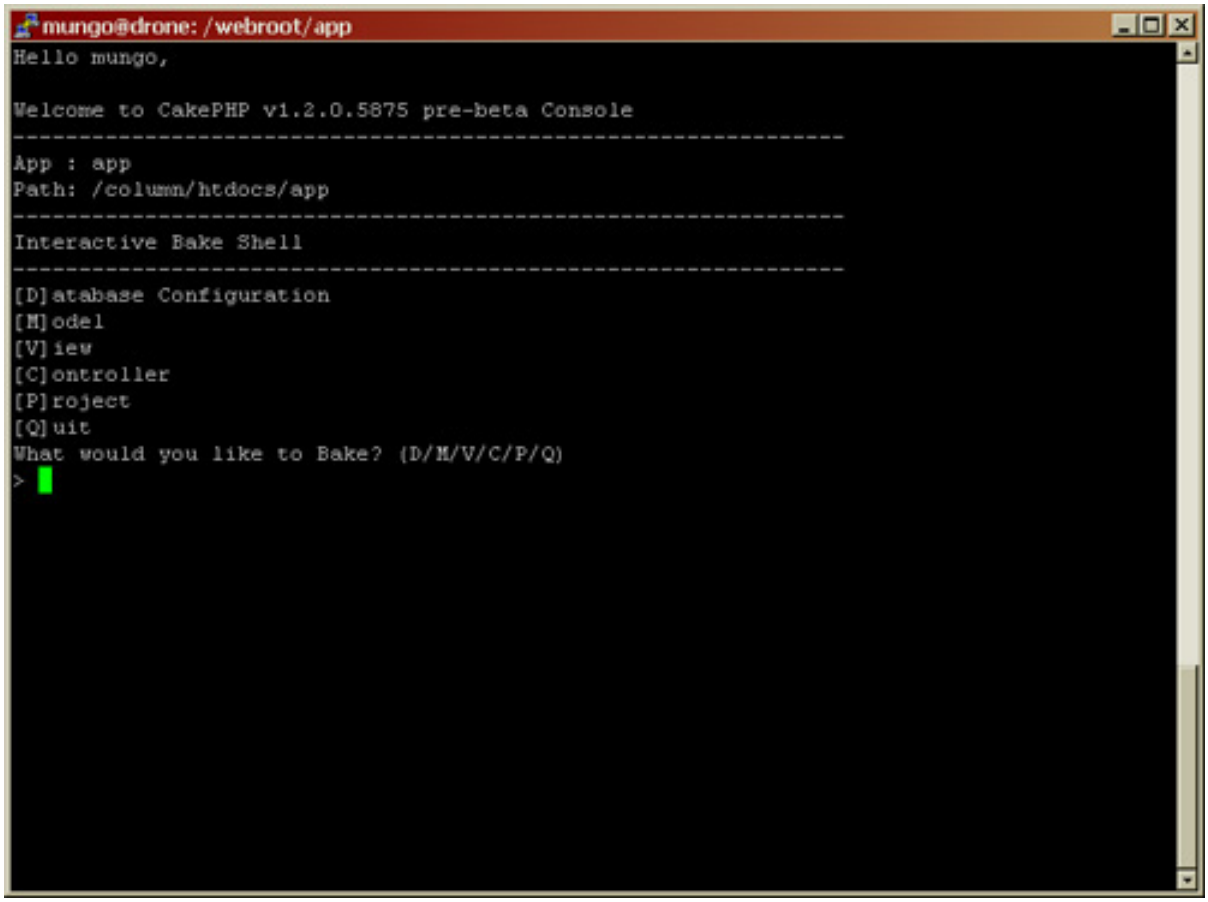
In CakePHP V1.1, Bake was a PHP script that you called directly. In CakePHP V1.2, the Bake functionality has been moved to the Cake Console, which you will be getting familiar with through the rest of this tutorial. You can simplify your life if you add the path /webroot/cake/console to your environment's `PATH` variable. This will allow you to call the Cake Console without specifying path information. You don't need to do this, and the tutorial will assume you have not. Additionally, you should run the Cake Console from your application's app directory — in this case, /webroot/app — or the Cake Console will think you are trying to do something new.

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"). If you have problems getting this to run, make sure `php` is in your environment's `PATH` variable.

Baking your products controller

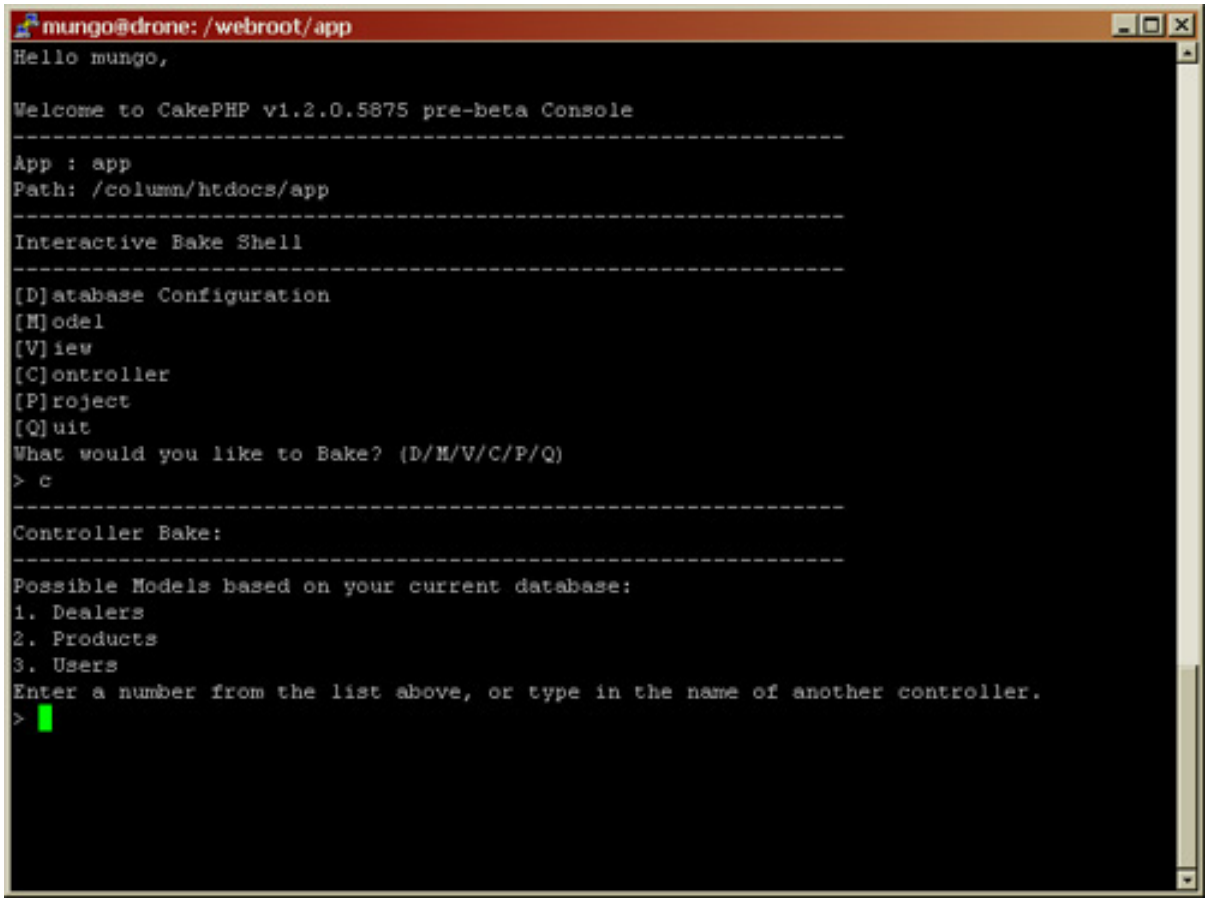
To use Bake, `cd` into the /webroot/app directory and launch the Cake Console: `../cake/console/cake bake`. You should be presented with a screen that looks like Figure 2.

Figure 2. Bake menu



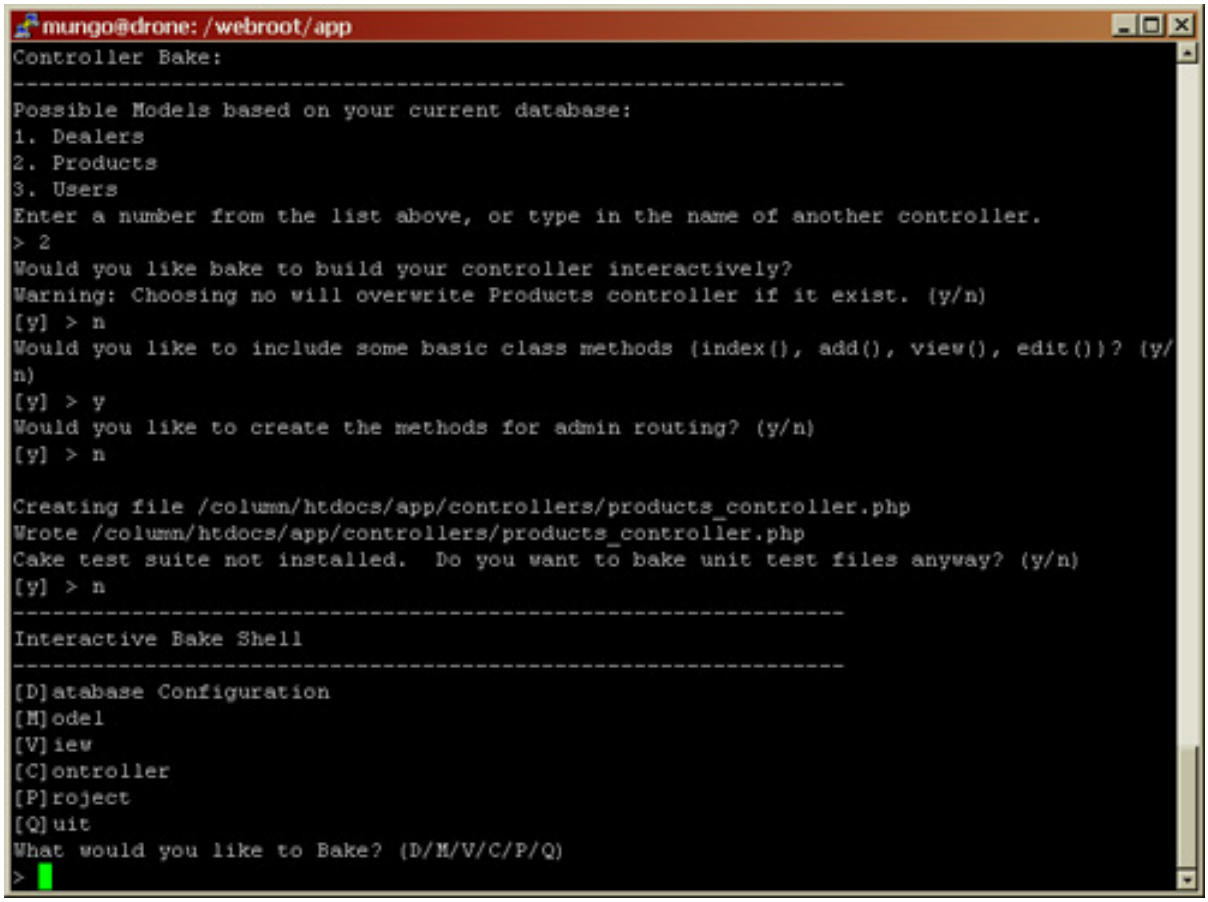
For the Tor application, the model you've written should be fine, so let's start with the controller. Press **C** to select the controller. The Cake Console will take a look at the application as it stands and present a list of possible controllers you may want to bake.

Figure 3. Controller name



In this case, you are baking the Products controller, which should be controller number two. 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 will then ask if you want to include some basic class methods (see Figure 4). Press **Y** — getting this code is the whole reason you are here. Next, Bake will ask it you want to create methods for admin routing. Press **N** — you don't need these right now. Bake should inform you that it's created the file \app\controllers\products_controller.php and ask if you want to bake some unit test files. You can skip these for now. Once you are done, you go back to the Bake menu.

Figure 4. Controller created



That's all there is to it. Exit the Cake Console, open up app/controllers/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

```
class ProductsController extends AppController {

    var $name = 'Products';
    var $helpers = array('Html', 'Form' );

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

    function view($id = null) {
if (!$id) {
  $this->Session->setFlash('Invalid Product.');
  $this->redirect(array('action'=>'index'), null, true);
}
$this->set('product', $this->Product->read(null, $id));

}

function add() {
if (!empty($this->data)) {
   $this->cleanUpFields();
   $this->Product->create();
   if ($this->Product->save($this->data)) {
 $this->Session->setFlash('The Product has been saved');
 $this->redirect(array('action'=>'index'), null, true);
   } else {
```

```
  $this->Session->setFlash('The Product could not be
saved. Please, try again.');
   }
}
$dealers = $this->Product->Dealer->generateList();
$this->set(compact('dealers'));
    }

function edit($id = null) {
if (!$id && empty($this->data)) {
    $this->Session->setFlash('Invalid Product');
    $this->redirect(array('action'=>'index'), null, true);
}
if (!empty($this->data)) {
    $this->cleanUpFields();
    if ($this->Product->save($this->data)) {
   $this->Session->setFlash('The Product has been saved');
   $this->redirect(array('action'=>'index'), null, true);

} else {

$this->Session->setFlash('The Product could not be
saved. Please, try again.');
}
}
if (empty($this->data)) {
    $this->data = $this->Product->read(null, $id);
}
$dealers = $this->Product->Dealer->generateList();
$this->set(compact('dealers'));
    }

function delete($id = null) {
   if (!$id) {
   $this->Session->setFlash('Invalid id for Product');
   $this->redirect(array('action'=>'index'), null, true);
   }
   if ($this->Product->del($id)) {
   $this->Session->setFlash('Product #'.$id.' deleted');
   $this->redirect(array('action'=>'index'), null, true);
}
}
}
?>
```

This controller you generated using the Cake Console 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?

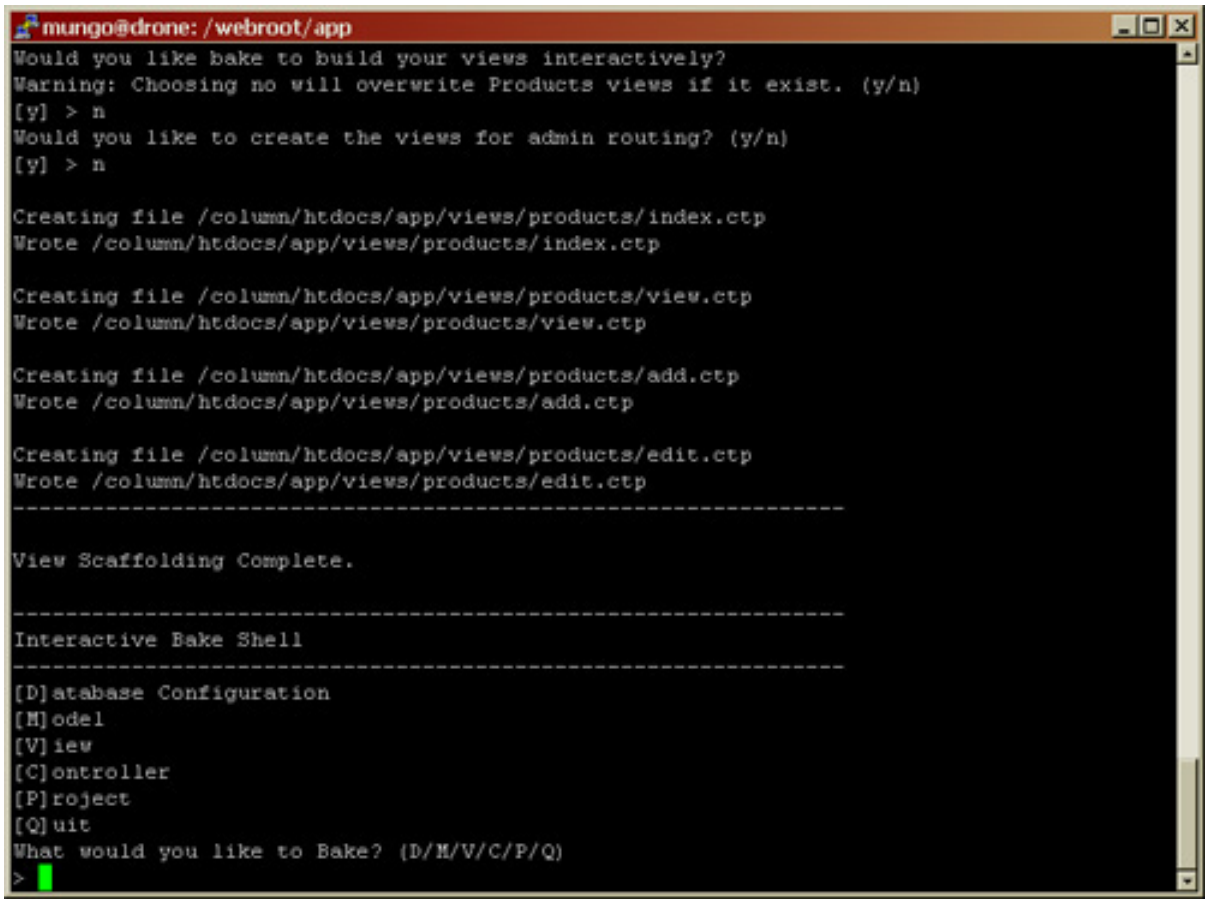
Baking your products views

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 on your /webroot/app directory: `../cake/console/cake bake`.

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 get another list of possible views to be baked. Products should still be number two on the list. Bake will ask if you want to build the views interactively. For now, press **N** to let Bake make all the decisions. Bake will also ask if you want to bake the views for admin routing. Skip those for now, too. You can come back later and play around with interactive baking and the admin routing options.

Once you get past those two options, Bake should inform you that it's created the views.

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



Exit the Cake Console and open the app/views/products/index.ctp view and take a look. It should look something like Listing 9.

Listing 9. The index

```
<div class="products">
<h2><?php __('Products');?></h2>
<p>
<?php
echo $paginator->counter(array(
'format' => __('Page %page% of %pages%, showing %current%
records out of %count% total, starting on record %start%, ending on %end%', true)
));
?></p>
<table cellpadding="0" cellspacing="0">
<tr>
<th><?php echo $paginator->sort('id');?></th>
<th><?php echo $paginator->sort('title');?></th>
<th><?php echo $paginator->sort('dealer_id');?></th>
<th><?php echo $paginator->sort('description');?></th>
<th class="actions"><?php __('Actions');?></th>
</tr>
<?php
$i = 0;
foreach ($products as $product):
$class = null;
if ($i++ % 2 == 0) {
```

```
$class = ' class="altrow"';
}
?>
<tr<?php echo $class;?>>
<td>
      <?php echo $product['Product']['id'] ?>
</td>
<td>
<?php echo $product['Product']['title'] ?>
</td>
<td>
      <?php echo $html->link(__($product['Dealer']
                                                ['title'], true), array('controller'=>
'dealers', '
                                                action'=>'view',
$product['Dealer']['id'])); ?>
</td>
<td>
<?php echo $product['Product']['description'] ?>
</td>
<td class="actions">
     <?php echo $html->link(__('View', true),
                                                array('action'=>'view',
$product['Product']['id'])); ?>
     <?php echo $html->link(__('Edit', true),
                                                array('action'=>'edit',
$product['Product']['id'])); ?>
      <?php echo $html->link(__('Delete', true),
                                                array('action'=>'delete',
$product['Product']['id']),
                                                null, sprintf(__('Are you sure you want to
delete
                                                #%s?', true),
                                                $product['Product']['id'])); ?>

</td>
</tr>
<?php endforeach; ?>
</table>
</div>
<div class="paging">
<?php echo $paginator->prev('<< '.__('previous', true),
                  array(), null, array('class'=>'disabled'));?>
 |      <?php echo $paginator->numbers();?>
<?php echo $paginator->next(__('next', true).' >>',
                  array(), null, array('class'=>'disabled'));?>
</div>
<div class="actions">
<ul>
  <li><?php echo $html->link(sprintf(__('New %s', true),
                              __('Product', true)), array('action'=>'add')); ?></li>
  <li><?php echo $html->link(sprintf(__('List %s', true),
                            __('Dealers', true)), array('controller'=> 'dealers',
                            'action'=>'index')); ?> </li>
  <li><?php echo $html->link(sprintf(__('New %s',  true),
                            __('Dealer', true)), array('controller'=> 'dealers',
                                'action'=>'add')); ?> </li>
</ul>
</div>
```

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 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.

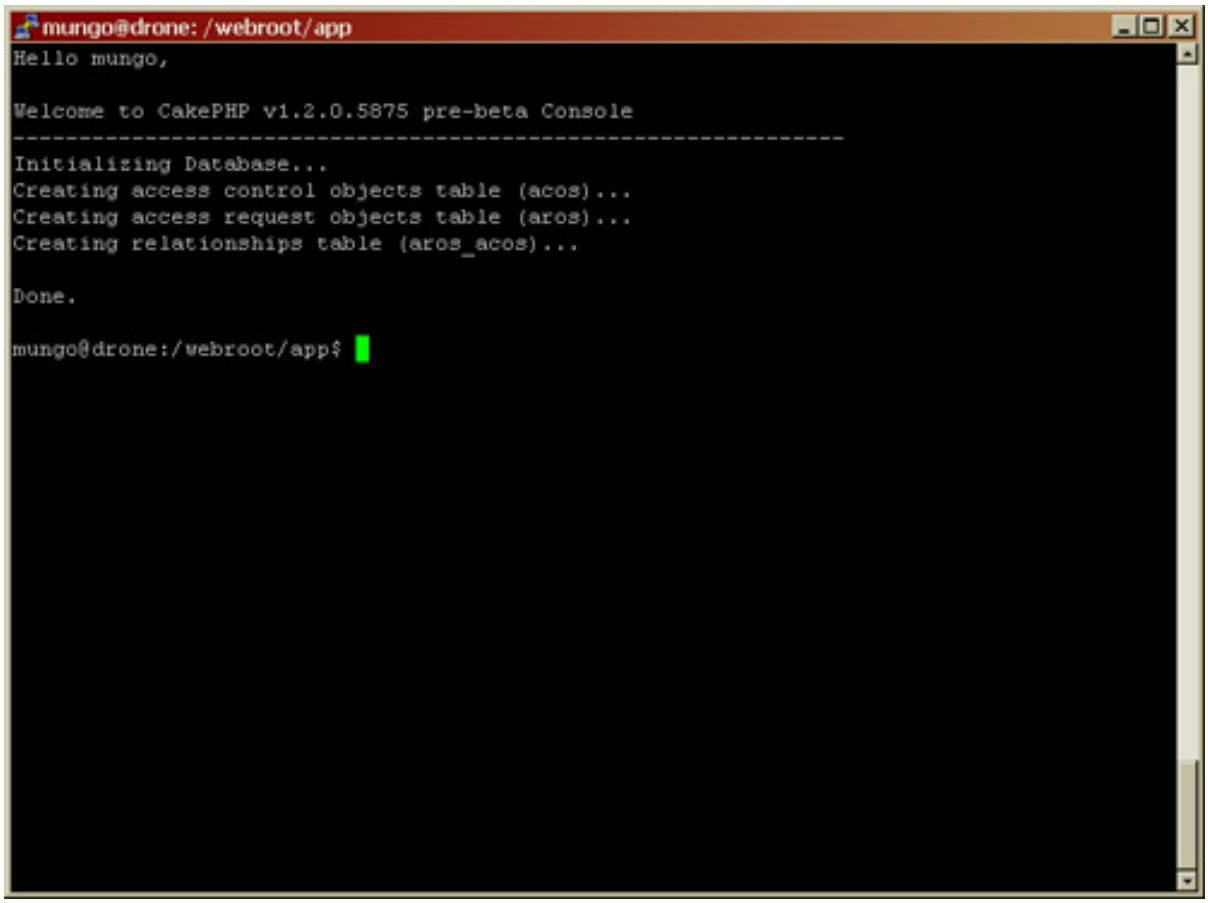
Imagine there's a party going on at some nightclub. Everyone who's anyone is there. The party is broken up into several sections — there's a VIP lounge, a dance floor, and a main bar. And, of course, a big line of people trying to get in. The big scary bouncer at the door checks a patron's ID, looks 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

In CakePHP V1.1, ACL management worked like baking, via a PHP script you called directly. In CakePHP V1.2, ACL management is part of the Cake Console. Using the Cake Console, you can set up a database table to be used to store ACL information. At the command line, from the /webroot/app directory, run the following command: `../cake/console/cake acl initdb`. The Cake Console will tell you it has made three databases (see Figure 6): acos, aros, and aros_acos.

Figure 6. ACL shell output



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

Defining AROs

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 then 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 Cake Console, much like you did to create the ACL database. To create the groups, execute the commands below from the /webroot/app 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, 'root Users') are parent, and node. The `parent` parameter would correspond to a group to which the ARO should belong. As these groups are top-level, you passed in root. The `node` 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. Original register action

```
    function register()
    {
if (!empty($this->data))
{
$this->data['User']['password'] = md5($this->data['User']
                                ['password']);
if ($this->User->save($this->data))
{
$this->Session->setFlash('Your registration information
                                    was accepted');
```

```
$this->Session->write('user', $this->data['User']
                                  ['username']);
$this->redirect(array('action' => 'index'), null, true);
} else {
$this->data['User']['password'] = '';
$this->Session->setFlash('There was a problem saving
                                  this information');
}
}
    }
```

To start using CakePHP's ACL component, you 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. Components are to Controllers as Helpers are to Views. 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. You can create an ARO by invoking the `create` method on the ACL's `ARO` object (that will be easier to read in Listing 12). This method takes the same parameters you would normally pass when you are calling the `create` method from a model, which is essentially what you are doing. In this case, you specify the name of the alias (the user name), the model that the ACL points to (user), the `foreign_key` for the record (the new user's ID), and the `parent_id` (the ID of the parent node — in this case, the Users ARO group, which is found using the `findByAlias` line below). 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()
    {
if (!empty($this->data))
{
$this->data['User']['password'] = md5($this->data['User']
                        ['password']);
if ($this->User->save($this->data))
{
$this->Session->setFlash('Your registration information
                                  was accepted');
$this->Session->write('user', $this->data['User']['username']);
$parent = $this->Acl->Aro->findByAlias('Users');
$this->Acl->Aro->create(array(
```

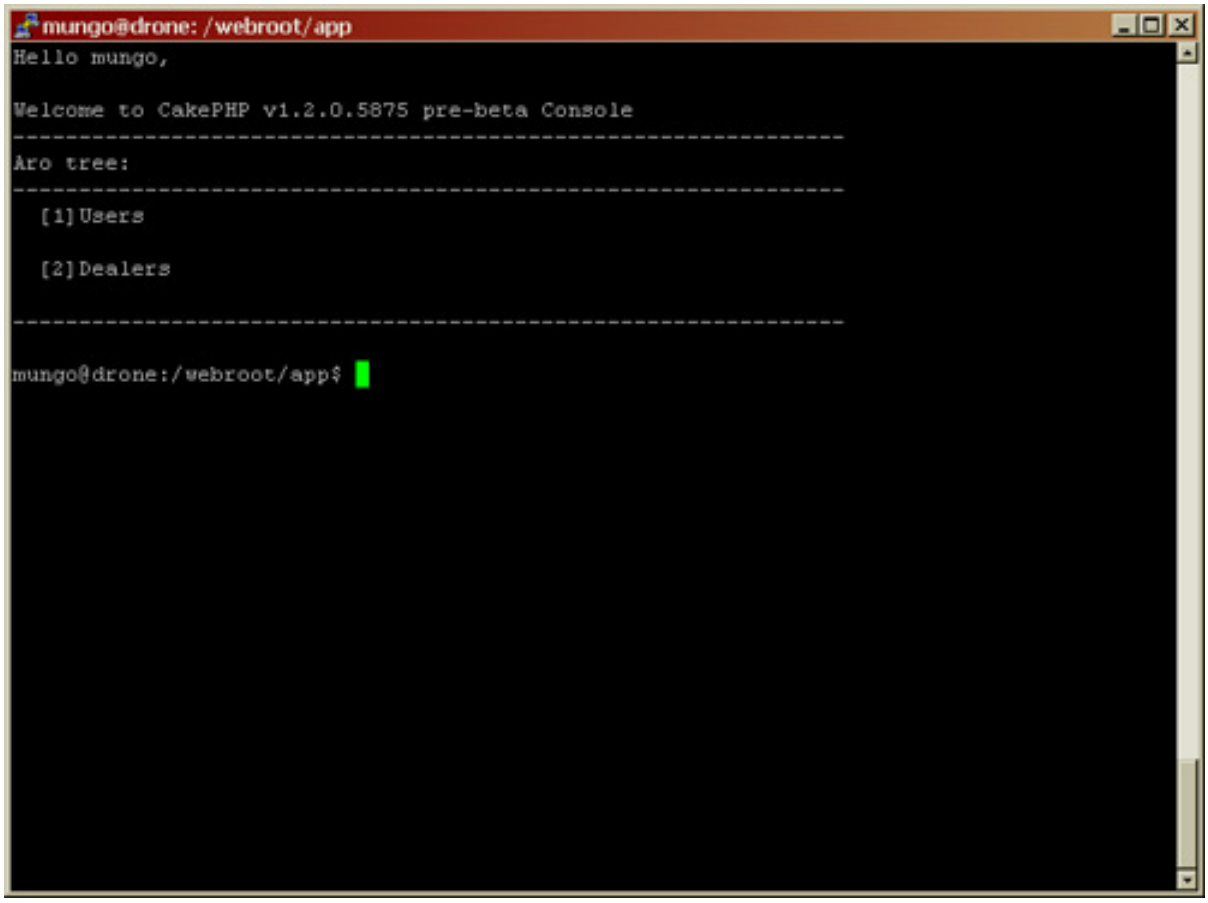
```
    'alias' => $this->data['User']['username'],
    'model' => 'User',
    'foreign_key' => $this->User->id,
    'parent_id' => $parent['Aro']['id'])
);
$this->Acl->Aro->save();
$this->redirect(array('action' => 'index'), null, true);
} else {
$this->data['User']['password'] = '';
$this->Session->setFlash('There was a problem saving
                                  this information');
}
}
    }
```

You'll note that the ARO is not created until the save has succeeded.

Try it out

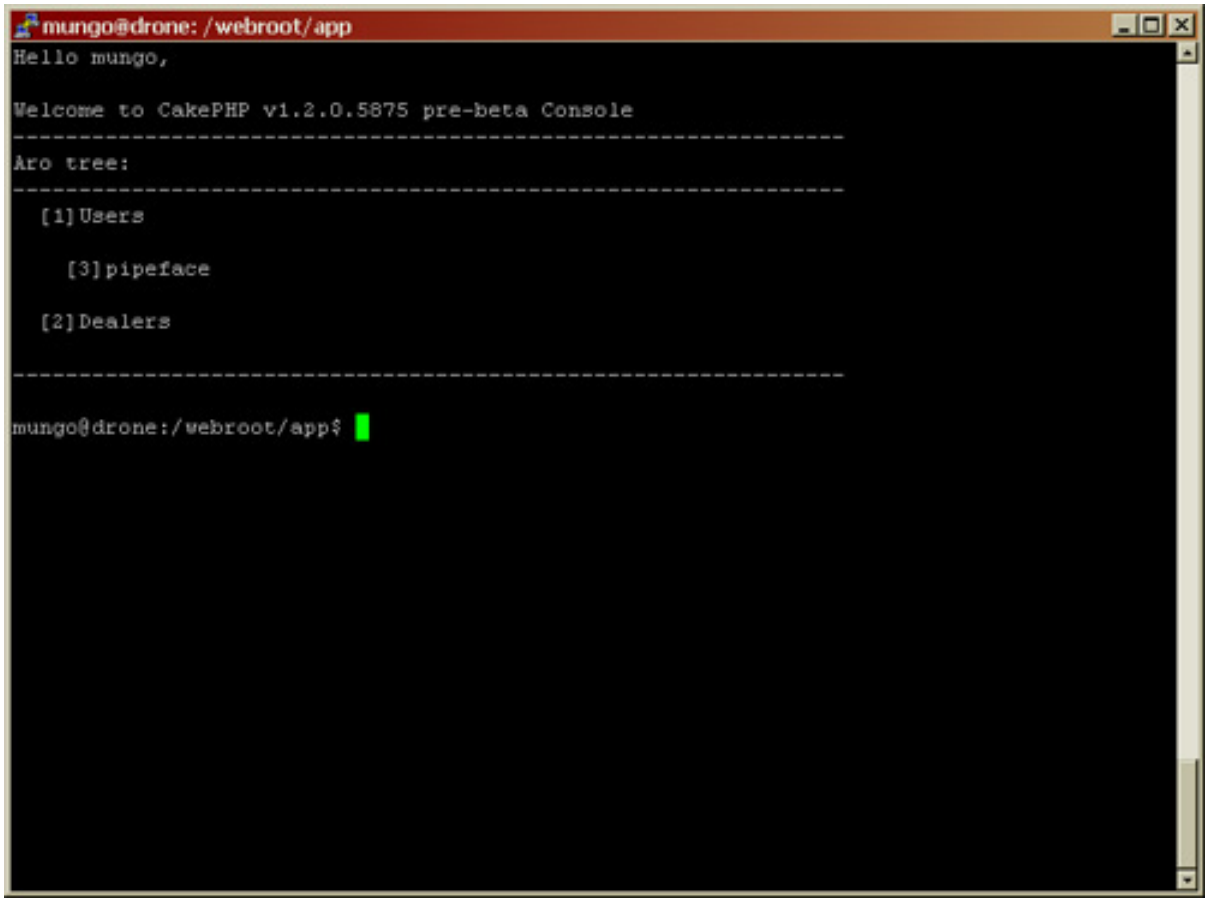
That should be all you need to get your AROs up and running. To verify, start back at the command line in webroot/app and ask the Cake Console to view the ARO tree: `../cake/console/cake acl view aro`. Your output should look something like Figure 7.

Figure 7. Output from ACL shell view aro with empty list



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

Figure 8. Output from ACL shell view aro with a user

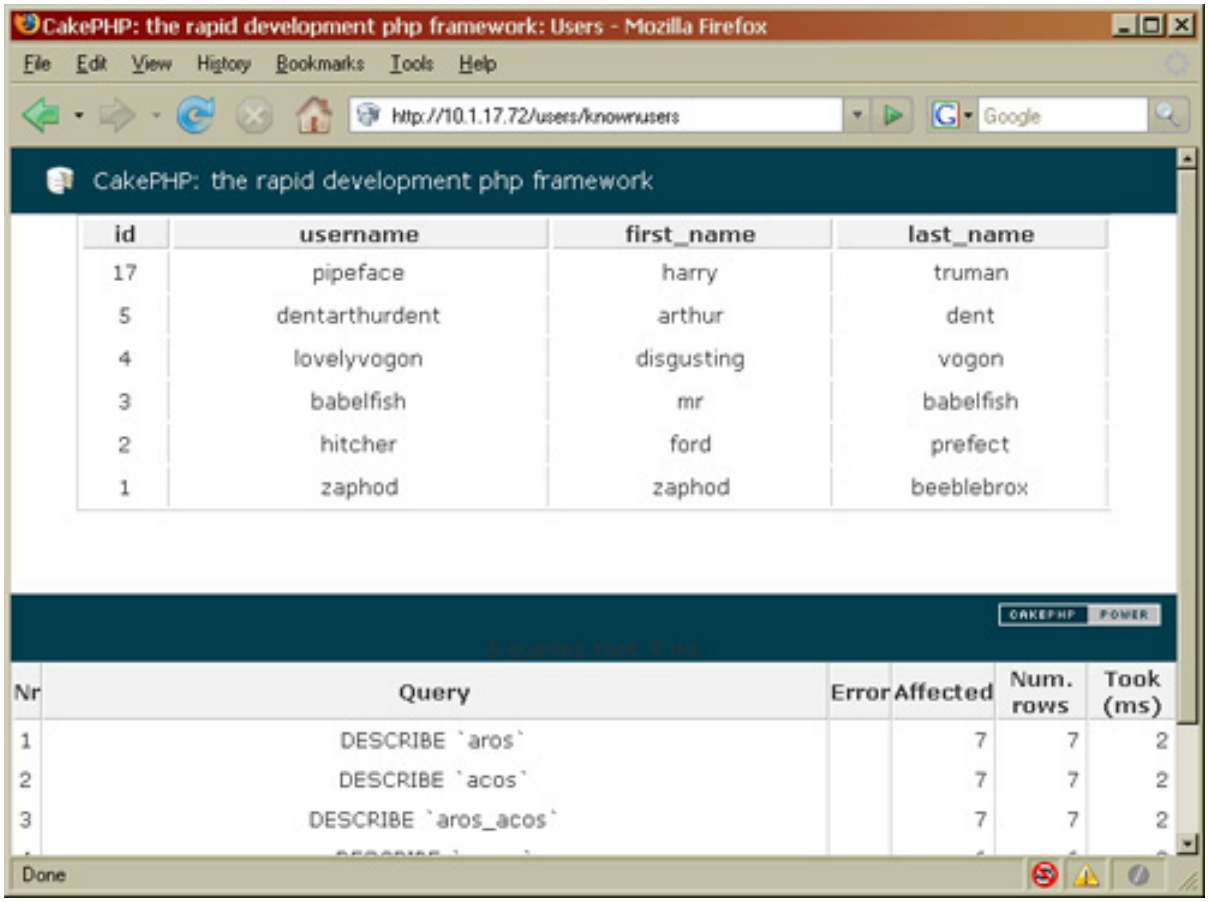


From now on, whenever someone registers for a new account, he will automatically have an ARO created for him. That ARO will belong to the users group.

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 Cake Console, 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.

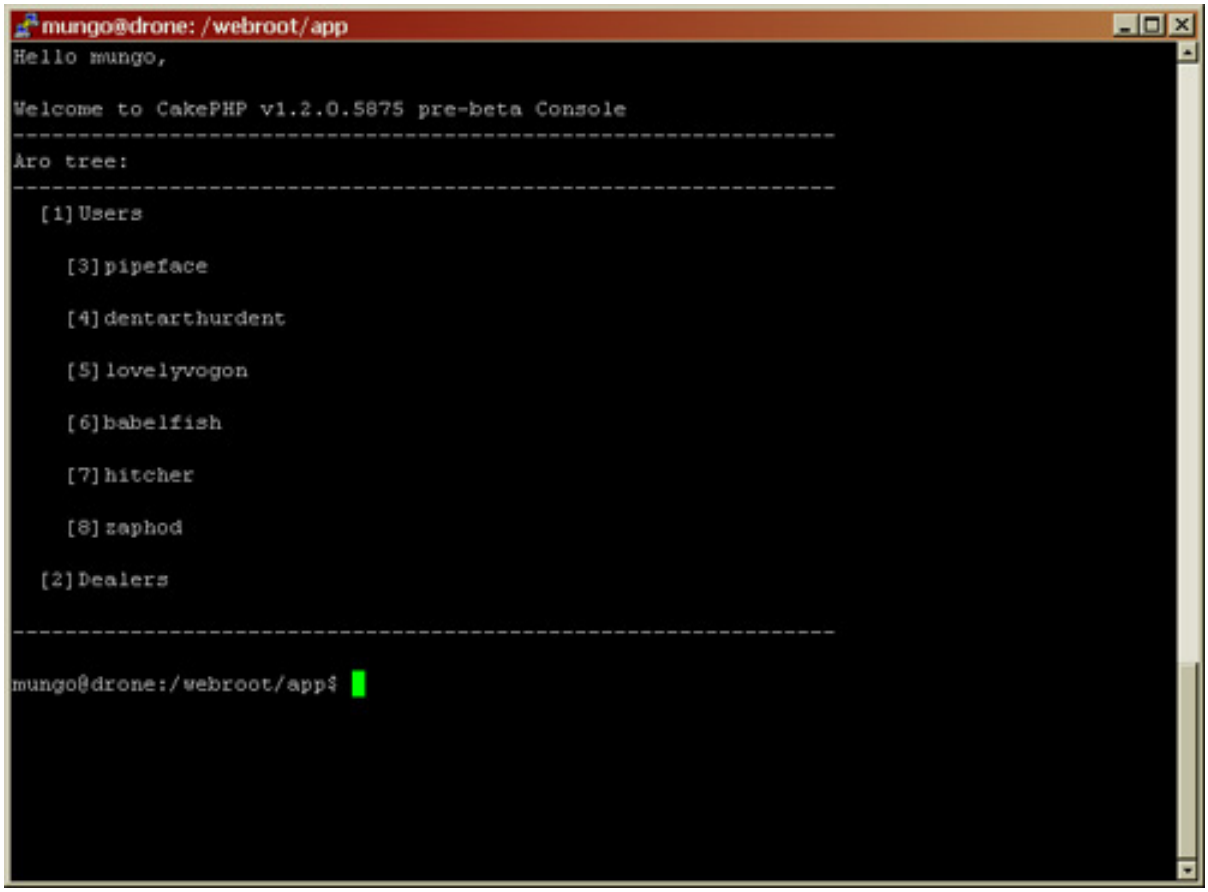
Figure 9. Output from ACL shell view ARO with a user



Then, for each user, you need to execute the create ARO command like you did for creating the groups. For parent, specify 'Users'. For node, specify the user name. For example, from Figure 9, to create an ARO for `dentarthurdent`, you would execute the following (again, from the /webroot/app directory): `../cake/console/cake acl create aro Users dentarthurdent`.

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 user name for each user. When you're done, the results of `../cake/console/cake acl` should look something like Figure 10.

Figure 10. Output from ACL shell ARO with a user



You can get help on some of the other things Cake Console can do with ACLs by running `../cake/console/cake acl help` at the command line.

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->cleanUpFields();
$this->Product->create();
if ($this->Product->save($this->data)) {
$this->Session->setFlash('The Product
has been saved');
$this->redirect(array('action'=>'index'),
null, true);
} else {
$this->Session->setFlash('The Product
could not be
saved. Please, try again.');
}
}
$dealers =
$this->Product->Dealer->generateList();
$this->set(compact('dealers'));
}
```

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 call the `create` method on the ACL's `ACO` object. This time, the alias needs to be more than just the product's title, since that may not be unique. Instead, you can use a combination of the product's ID and the product's title for the alias. The model will be Product, the `foreign_key` will be the new product's ID, and the `parent_id` will be the ID of the dealer who inserted the product (you haven't set this up yet, but you will shortly). 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->cleanUpFields();
$this->Product->create();
if ($this->Product->save($this->data)) {
$dealer = $this->Product->Dealer->read(null,
$this->data['Product']['dealer_id']);
$parent = $this->Acl->Aco->findByAlias($dealer
                                                    ['Dealer']['title']);
$this->Acl->Aco->create(array(
'alias' =>
                $this->Product->id.'-'.$this->data['Product']['title'],
'model' => 'Product',
'foreign_key' => $this->Product->id,
'parent_id' => $parent['Aco']['id'])
);
$this->Acl->Aco->save();
$this->Session->setFlash('The Product
                                                  has been saved');
$this->redirect(array('action'=>'index'),
                                                  null, true);
```

```
} else {
$this->Session->setFlash('The Product could not
                                                        be saved. Please, try
again.');
}
}
$dealers = $this->Product->Dealer->generateList();
$this->set(compact('dealers'));
}
```

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

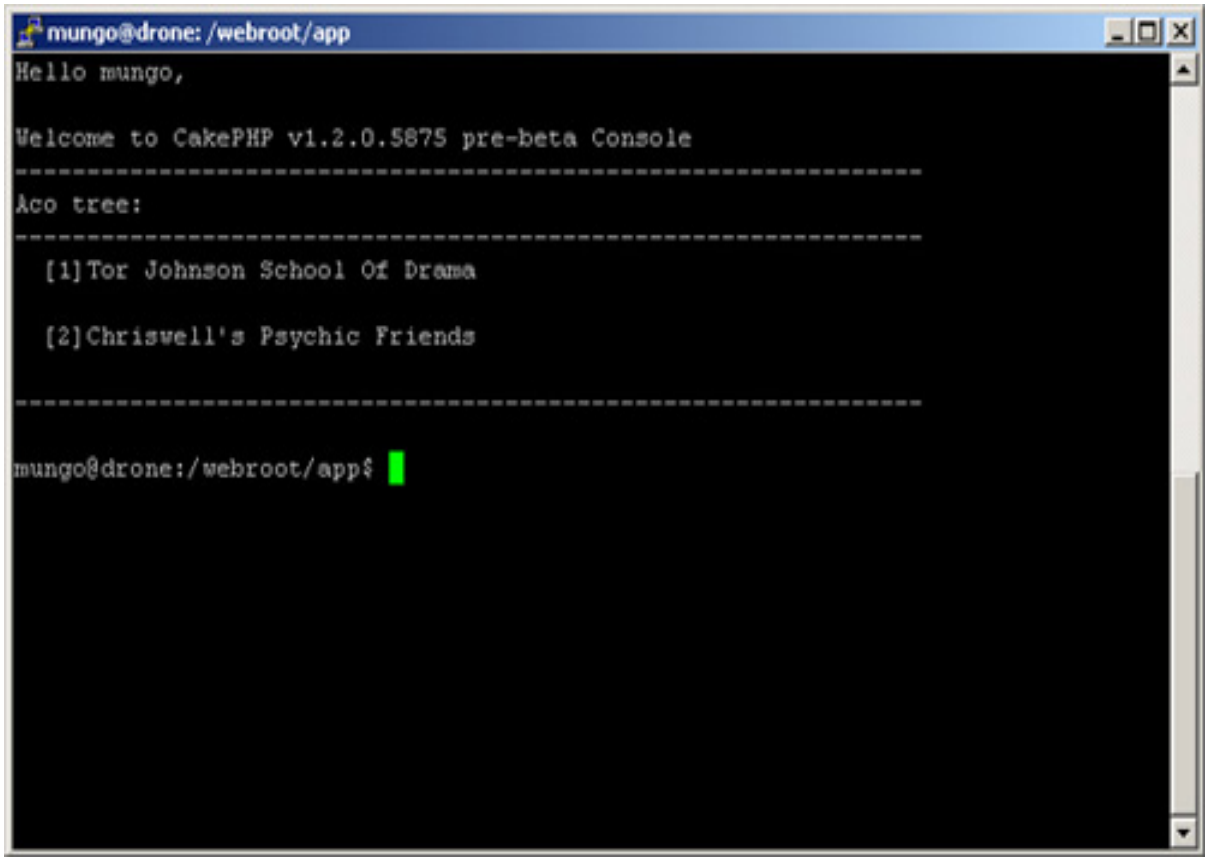
The Cake Console can be used to define ACOs in much the same way it was used to define the AROs for existing users. It will be helpful to pull up that products list that CakePHP baked for you at http://localhost/products.

Once again, from the command line, in the /webroot/app 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.

```
../cake/console/cake acl create aco root "Tor Johnson School Of Drama"
../cake/console/cake acl create aco root "Chriswell's Psychic Friends"
```

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

Figure 11. ACO dump with dealers no products



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.

Because you have only created a couple of products thus far, recreating them is the shortest path to adding the ACOs you want. 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 the stage is set to create some 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

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 what is shown in Listing 16.

Listing 16. New add function in the products controller

```
function add() {if (!empty($this->data)) {
  $this->cleanUpFields();
  $this->Product->create();
  if ($this->Product->save($this->data)) {
  $dealer = $this->Product->Dealer->read(null,
                                  $this->data['Product']['dealer_id']);
  $parent = $this->Acl->Aco->findByAlias($dealer
                                   ['Dealer']['title']);
  $this->Acl->Aco->create(array(
       'alias' => $this->Product->id.'-'.$this->data
                                        ['Product']['title'],
       'model' => 'Product',
       'foreign_key' => $this->Product->id,
       'parent_id' => $parent['Aco']['id'])
);
  $this->Acl->Aco->save();
  $this->Acl->allow('Users',
                  $this->Product->id.'-'.$this->data['Product']['title'],
```

```
                                    'read');
  $this->Acl->allow($this->Session->read('user'),
                                          $this->Product->id.'-'.$this->data['Product']
                                             ['title'],'*');
  $this->Session->setFlash('The Product has been saved');

  $this->redirect(array('action'=>'index'), null, true);
 } else {
  $this->Session->setFlash('The Product could not be saved.
                                      Please, try again.');
}
}
$dealers = $this->Product->Dealer->generateList();
$this->set(compact('dealers'));
}
```

OK — now you can try adding some products. Try logging in as one of your users and adding a couple of products, just to see that nothing got broken along the way. You can use the Cake Console to view the ACOs you create when you add a new product. 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, and 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.

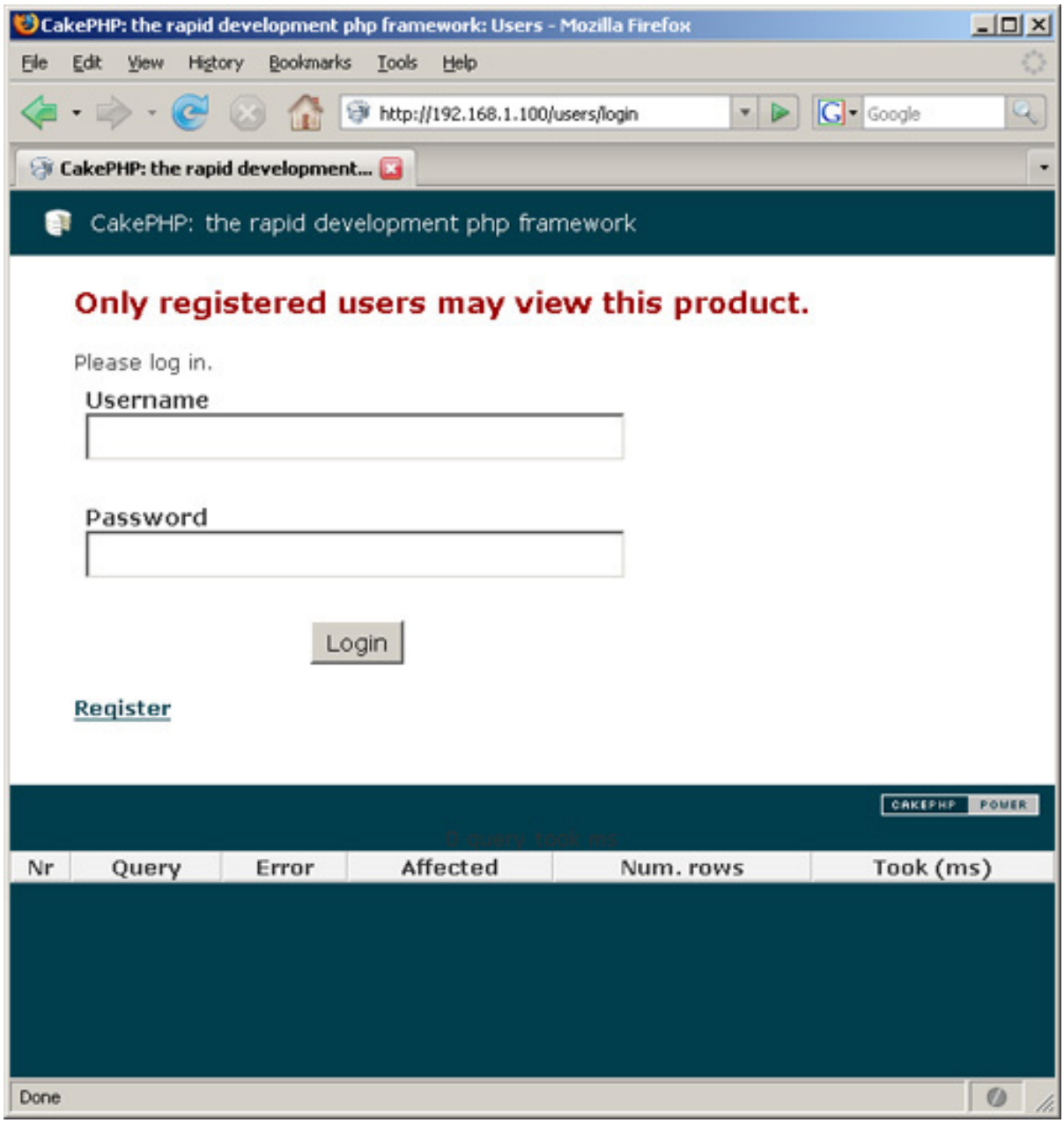
Listing 17. Adding a line to check access to the product

```
function view($id = null) {
if (!$id) {
  $this->Session->setFlash('Invalid
Product.');
  $this->redirect(array('action'=>'index'),
null, true);
}
$product = $this->Product->read(null, $id);
if
($this->Acl->check($this->Session->read('user'),
                              $id . '-' .
```

```
$product['Product']['title'], 'read')) {
$this->set('product', $product);
} else {
   $this->Session->setFlash('Only registered
users may view this product.');
   $this->redirect(array('action'=>'index'),
null, true);
      }
}
```

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, as shown in Figure 12.

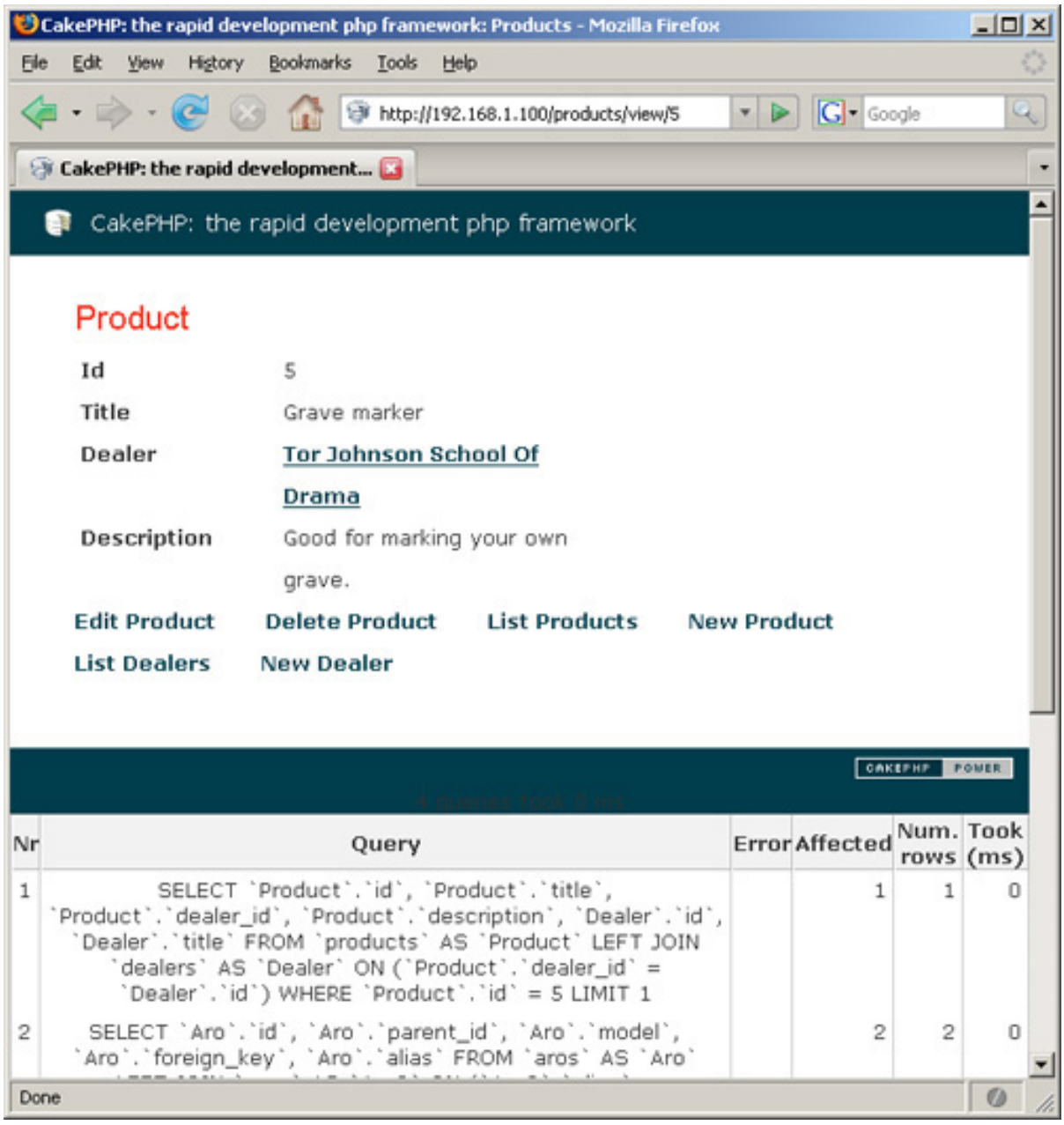
Figure 12. Redirection



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

the product, like what you see in Figure 13.

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 for controlling permissions is much the same for the `edit` and `delete` actions in the products controller.

Listing 18. The edit action

```
function edit($id = null) {
      $product = $this->Product->read(null, $id);
      if ($this->Acl->check($this->Session->read('user'),
        $id.'-'.$product['Product']['title'], 'update')) {
  if (!$id && empty($this->data)) {
       $this->Session->setFlash('Invalid Product');
       $this->redirect(array('action'=>'index'), null, true);
  }
  if (!empty($this->data)) {
       $this->cleanUpFields();
       if ($this->Product->save($this->data)) {
   $this->Session->setFlash('The Product has been saved');
   $this->redirect(array('action'=>'index'), null, true);
} else {
$this->Session->setFlash('The Product could
                                                    not be saved.
                                                Please, try again.');
}
}
  if (empty($this->data)) {
   $this->data = $this->Product->read(null, $id);
  }
  $dealers = $this->Product->Dealer->generateList();
  $this->set(compact('dealers'));
} else {
$this->Session->setFlash('You cannot edit this product.');
$this->redirect(array('action'=>'index'), null, true);
}
}
```

For the delete controller, you should add a couple lines to delete the ACO for the product being deleted. Your `delete` action will look like Listing 19.

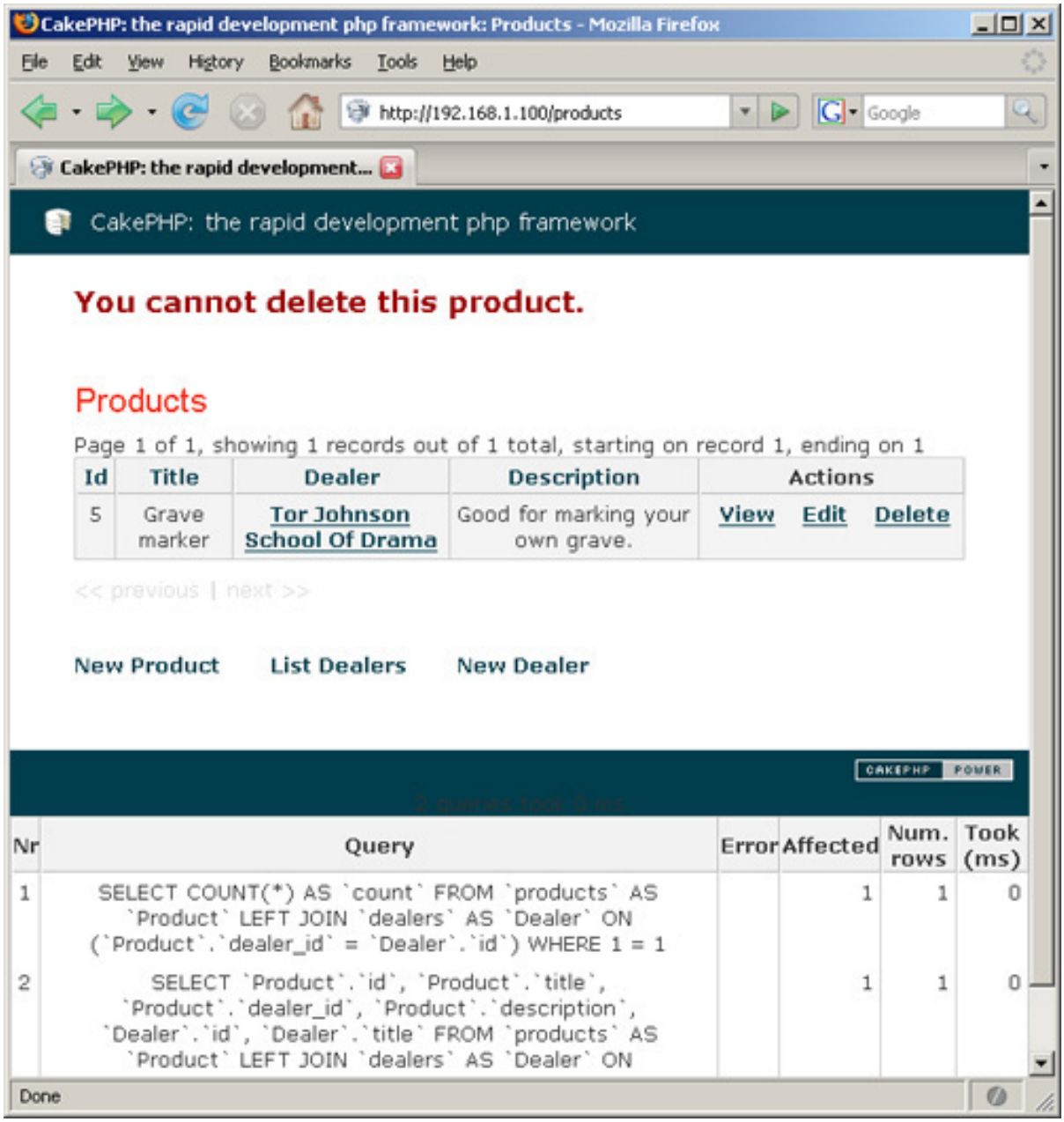
Listing 19. The delete action

```
function delete($id = null) {
    if (!$id) {
$this->Session->setFlash('Invalid id for Product');
$this->redirect(array('action'=>'index'), null, true);
    }
    $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 = $this->Acl->Aco->findByAlias($id.'-'
                                .$product['Product']['title']);
   $this->Acl->Aco->delete($aco['Aco']['id']);
   $this->Session->setFlash('Product #'.$id.' deleted');
   $this->redirect(array('action'=>'index'), null, true);
 }
    } else {
 $this->Session->setFlash('You cannot delete this product.');
 $this->redirect(array('action'=>'index'), null, true);
    }
 }
```

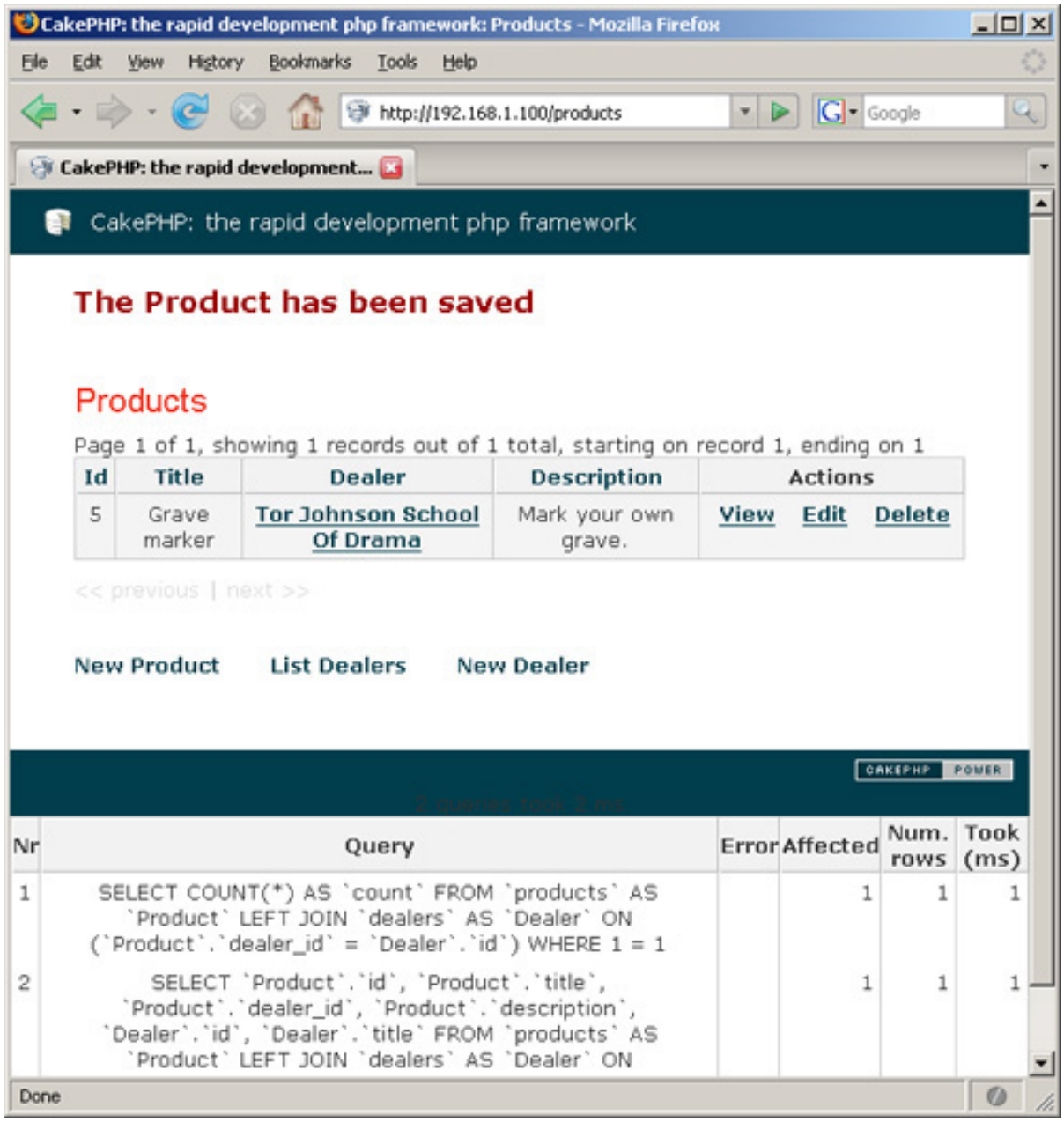
Save the products controller and try it out. Start by logging out at http://localhost/users/logout, then go back to your products list at http://localhost/products/ and try to edit or delete a product. You should get directed back to the products list with a message.

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 a different user. Then try to edit or delete another product, and you will find you are unable to take any meaningful action. 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 the Cake Console 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.zip	74KB	HTTP

Information about download methods

Resources

Learn

- Visit CakePHP.org to learn more.
- The CakePHP API has been thoroughly documented. This is the place to get the most up-to-date documentation.
- 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 Wikipedia's Model-View-Controller.
- Here is more useful background on the Model-View-Controller.
- Check out a list of software design patterns.
- Read more about Design Patterns.
- PHP.net is the central resource for PHP developers.
- Check out the "Recommended PHP reading list."
- Browse all the PHP content on developerWorks.
- Expand your PHP skills by checking out IBM developerWorks' PHP project resources.
- To listen to interesting interviews and discussions for software developers, check out developerWorks podcasts.
- Using a database with PHP? Check out the Zend Core for IBM, a seamless, out-of-the-box, easy-to-install PHP development and production environment that supports IBM DB2 V9.
- 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.
- Watch and learn about IBM and open source technologies and product functions with the no-cost developerWorks On demand demos.

Get products and technologies

- Innovate your next open source development project with IBM trial software, available for download or on DVD.
- Download IBM product evaluation versions, and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

Discuss

- Participate in developerWorks blogs and get involved in the developerWorks community.
- Participate in the developerWorks PHP Forum: Developing PHP applications with IBM Information Management products (DB2, IDS).

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.