Rapid Application Development with CakePHP 2.0

Jose Diaz-Gonzalez

Rapid Application Development

with CakePHP 2.0

Jose Diaz-Gonzalez

This book is for sale at http://leanpub.com/rad-cakephp-2

This version was published on 2014-11-09

Leanpub

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Jose Diaz-Gonzalez by spreading the word about this book on Twitter!

The suggested hashtag for this book is #cakephpbook.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#cakephpbook

Contents

User Authentication

Many pastebins can be used anonymously - that is, anyone can come and paste anything they desire. Our pastebin will allow users the option of logging in, allowing them to associate pastes with their own accounts, as well as allowing some level of paste management.

User Authentication

Creating new users

Before we allow users to login, we'll need to create both a users table, as well as a model for that table. We can create the table using the Migrations plugin:

```
cd /vagrant/app

# ensure cakephp is running with the proper configuration on the command line
source /etc/service-envs/app.env

# create the migration
app/Console/cake Migrations.migration generate create_users id:primary_key usern\
ame:string:unique password:string email:string:unique created modified

# run the migration against our database
app/Console/cake Migrations.migration run all
```

Now we can create our model class. Model classes are used to contain logic regarding stateful access for collections of entities from our sources of data. In this case, we'll create a `User` model that will manage all the logic regarding the retrieval and maintenance of user records. All models should go within our `Model` folder to allow CakePHP's class loader to locate the classes when necessary:

```
touch /vagrant/app/app/Model/User.php
```

A simple model class simply extends the `AppModel` class. Model classes should use the singular version of the table name - the model for our `pastes` table would be `Paste`, and the model for our `users` table is `User`:

```php
<?php
App::uses('AppModel', 'Model');

class User extends AppModel {
}
?>
```

Whenever we create a user, we want to be sure that their passwords are hashed. We can do so by adding a little bit of code to our `User` model:

```php
<?php
App::uses('AppModel', 'Model');
App::uses('BlowfishPasswordHasher', 'Controller/Component/Auth');

class User extends AppModel {
  public function beforeSave($options = []) {
    if (!empty($this->data[$this->alias]['password'])) {
      $passwordHasher = new BlowfishPasswordHasher();
      $this->data[$this->alias]['password'] = $passwordHasher->hash(
        $this->data[$this->alias]['password']
      );
    }
    return true;
  }
}
?>
```

In the above code, we define a `beforeSave` method. This is one of several model callbacks available to us , and allows us to modify data before it is persisted to our database. We use this in combination with the `BlowfishPasswordHasher` to ensure that incoming passwords are hashed. Note that the method returns `true`; if `beforeSave` returns false, then the save call will fail.

Implementing Login/Logout

CakePHP makes it extremely easy to log a user in and out using the `AuthComponent`. This class is used in the controller layer to handle user authentication information and properly authenticate the user against your backend.

To set it up, you'll want to add the following configuration to your `AppController::$components` array:

```
public $components = [
  // Other configured components here
  'Session',
  'Auth' => [
    'authenticate' => [
      'Form' => ['passwordHasher' => 'Blowfish']
    ],
    'authorize' => ['Controller'],
    'flash' => [
      'element' => 'default',
      'key' => 'flash',
      'params' => []
    ],
    'loginRedirect' => [
      'controller' => 'pastes',
      'action' => 'index'
    ],
    'logoutRedirect' => [
      'controller' => 'pages',
      'action' => 'display',
      'home'
    ],
  ]
];
```

The above configuration will:

- Attach the SessionComponent so we can save the user's state between page loads
- Configures the AuthComponent to use the controller method `isAuthorized()` to grant access to specific actions (more on this later)
- Configure the AuthComponent to redirect our users to `/pastes/index` on successful login
- Configure the AuthComponent to redirect our users to `/` - an alias for the homepage - when they logout

Next, we'll want to configure a `login` and a `logout` action to handle the logic for each method. These should go in our new `UsersController` as follows:

```php
<?php
App::uses('AppController', 'Controller');

class UsersController extends AppController {
  public function login() {
    if ($this->request->is('post')) {
      if ($this->Auth->login()) {
        return $this->redirect($this->Auth->redirectUrl());
      }
      $this->Session->setFlash(__('Invalid username or password, try again'));
    }
  }

  public function logout() {
    return $this->redirect($this->Auth->logout());
  }
}
?>
```

These actions will be available at `/users/login` and `/users/logout`, as per default CakePHP route conventions. To complete login functionality, we'll also require a view template for the `login` action as follows. You should create this file in `app/View/Users/login.ctp`:

```php
<div class="users form">
  <?php echo $this->Form->create('User'); ?>
    <fieldset>
      <legend><?php echo __('Login'); ?></legend>
      <?php echo $this->Form->input('username'); ?>
      <?php echo $this->Form->input('password'); ?>
    </fieldset>
  <?php echo $this->Form->end(__('Submit')); ?>
</div>
```

We previously mentioned that we would have define an `isAuthorized()` method to control access to specific actions in our app. This method is called once a user has been logged in - that is to say, this method is *not* executed for anonymous users. Anonymous users are by default not allowed to access any page on the site by the AuthComponent other than the login action. If you want to allow them access, you will need to add a line similar to the following to your controller's `beforeFilter()` (place this logic in your AppController):

```
public function beforeFilter() {
  parent::beforeFilter();
  // other beforeFilter() logic here

  // Allow users access to display action
  // in the PagesController
  $this->Auth->allow('display');
}
```

For logged in users, we can define an `isAuthorized()` method to allow access to a specific page. Since this is called after the `beforeFilter()`, it has access to any changed state from that part of a request, and also has access to the request in general. If the method returns `true`, a user is allowed access to execute their request. If the method returns false, one of several things can occur:

- If the app is not configured to redirect on auth error, a ForbiddenException is thrown that can be caught by your Application's Error Handler
- If the app is configured to redirect and has a refering page that is on the same domain as your app, the user will be redirected to the refering page
- If the app is configured to redirect but there is no valid referer, the user will be redirected to app's homepage.

For our use, we'll add the following to the `AppController` for now:

```
public function isAuthorized($user = null) {
  return true;
}
```

The above will allow logged in users access to any action in the app. In the next chapter we will cover simple access controls, but for now this is sufficient.

Registering users

Now that we have a functioning login system, we need a way for users to register for our service. We can create a very simple register action using the Crud plugin. We will first need to map `UsersController::register()` to the `Crud.Add` action in the `UsersController::beforeFilter()`:

```
public function beforeFilter() {
  parent::beforeFilter();
  $this->Crud->mapAction('register', 'Crud.Add');
  $this->Auth->allow('register');
}
```

We also added an `Auth::allow()` call to ensure non-authenticated users can actually register themselves.

Next, we'll want to modify the action to ensure only the `username`, `email`, and `password` fields are saved and nothing more. We also want to redirect registered users to the login page, as well as customize the flash messages that are shown to users when they submit forms:

```
public function register() {
  // Whitelists only the desired fields for saving
  $this->Crud->action()->config('saveOptions', [
    'fieldList' => [
      'User' => ['username', 'email', 'password'],
    ]
  ]);

  // Redirect to /users/login after registering
  $this->Crud->action()->config('redirect', [
    'post_add' => [
      'reader' => 'request.data',
      'key' => '_add',
      'url' => [
        'controller' => 'users',
        'action' => 'login'
      ],
    ]
  ]);

  // Updates the flash messages to be pertinent to the current user
  $this->Crud->action()->config('messages', [
    'success' => ['text' => '{name} was successfully registered'],
    'error' => ['text' => 'Could not register {name}']
  ]);

  return $this->Crud->execute();
}
```

Note that we haven't blocked any already logged in users from accessing the register action. We can do that by modifying the `UsersController::isAuthorized()` method as follows:

```
public function isAuthorized($user = null) {
  if ($this->request->action == 'register') {
    return false;
  }

  return parent::isAuthorized($user);
}
```

Lastly, we'll need to bake a template for this action. We can create a single form action using the `action` argument, and alias it as `register` via the `alias` argument:

```
cd /vagrant/app

# ensure cakephp is running with the proper configuration on the command line
source /etc/service-envs/app.env

# create the `register` view from the form bake template
app/Console/cake bake view Users form register
```

Views that are baked via the default CakePHP templates will include a basic scaffolded sidebar. This may be inappropriate for your application, and if so, feel free to remove it from the baked views.

You should now be able to register new users and sign in as them. We'll push these changes up in the meantime:

```
cd /vagrant/app
git add app/Config/Migration
git add app/Controller/AppController.php
git add app/Controller/UsersController.php
git add app/Model/User.php
git add app/View/Users/
git commit -m "Implemented user login/logout and registration"
git push origin master
git push heroku master
```

Authorizing Paste Access

Now that we have implemented user authentication within our application, we'll want to allow users to access to viewing pastes as well as restricting editing to their own pastes. To do so, we'll need a way to tie a specific user to the paste they created. We also want to allow anyone to view a paste, regardless of whether they are logged in or not.

Anonymous Access

To allow non-logged in users access to paste viewing, we will need to modify our `PastesController::beforeFilter` as we did in the AppController above:

```
public function beforeFilter() {
  parent::beforeFilter();
  // other beforeFilter() logic here

  // Allow all users access to index, view, and p actions
  $this->Auth->allow('index', 'view', 'private');

  // Also allow anonymous paste creation
  $this->Auth->allow('add');
}
```

Modeling relationships

Now that we've allowed anonymous users the ability to create and read pastes, we need to ensure that we properly track pastes that belong to a particular user. Let's start by using the `Migrations` plugin to create a `user_id` field on the `pastes` table. Doing so will allow us to track pastes on a per-user basis.

```
cd /vagrant/app

# ensure cakephp is running with the proper configuration on the command line
source /etc/service-envs/app.env

# create the migration
app/Console/cake Migrations.migration generate add_user_to_pastes user_id:intege\
r:index

# run the migration against our database
app/Console/cake Migrations.migration run all
```

Once the database migration has completed, we will need to modify our models to add this new relationship. In our case, a User record may have one or more pastes, while a Paste may belong to exactly one user. The relationship is therefore one-to-many - or `hasMany` - where one User can have many Pastes. The inverse relationship is many-to-one - or `belongsTo` - where many Pastes can belong to one user.

> CakePHP also supports other types of relationships. To see more details, refer to the documentation online[1]

[1] http://book.cakephp.org/2.0/en/models/associations-linking-models-together.html

To specify the relationship in the User model, we can use the `$hasMany` property as follows:

```
<?php
App::uses('AppModel', 'Model');

class User extends AppModel {
  public $hasMany = ['Paste'];

  // other code here
}
?>
```

Similarly, we would use the `$belongsTo` property in the Paste model:

```
<?php
App::uses('AppModel', 'Model');

class Paste extends AppModel {
  public $belongsTo = ['User'];

  // other code here
}
?>
```

Linking the two models allows us to query by one model and include related data for the other model like so:

```
// Will find user 1 and all of their pastes
$this->User->find('first', [
  'conditions' => ['User.id' => 1],
  'contain' => ['Paste'],
]);

// Will find all pastes as well as users that created them
// Will find user 1 and all of their pastes
$this->Paste->find('all', [
  'contain' => ['User'],
]);
```

This is an extremely powerful way of exposing querying for extra data necessary for a given page without constructing convoluted for loops or manual joins.

There are some cases where a join is optimal or desired, in which case the CakePHP documentation has an excellent section on using joins to retrieve related data[2]

Matching up a user to a paste

Whenever a user tries to create a paste, we will want to associate their `user_id` with the new paste. We can do so by creating a modified `PastesController::add()` action as follows:

```
public function add() {
  // previous add code here

  // Get the current user_id
  $user_id = $this->Auth->user('id');

  // Hook into the initialize Crud event and pass in the user_id
  $this->Crud->on('initialize', function(CakeEvent $event) use ($user_id) {
    // Get a shorter reference to the request object
    $request = $event->subject->request;

    // Only modify the data if it is a POST or PUT request
    if ($request->is(['post', 'put'])) {
      // Set the user_id in the posted data
      $request->data['Paste']['user_id'] = $user_id;
    }
  });

  return $this->Crud->execute();
}
```

With the above change, a user_id will be attached to all new pastes. Next we'll make sure that only a user can edit and delete their own post by changing the find requirements on that action:

[2] http://book.cakephp.org/2.0/en/models/associations-linking-models-together.html#joining-tables

```
public function edit() {
  $this->Crud->on('beforeRedirect', [$this, '_beforeRedirectSave']);
  $this->Crud->on('beforeFind', [$this, '_onBeforeFind']);
  return $this->Crud->execute();
}

public function delete() {
  $this->Crud->on('beforeFind', [$this, '_onBeforeFind']);
  return $this->Crud->execute();
}

public function _onBeforeFind(CakeEvent $event) {
  // Get the current user_id
  $user_id = $this->Auth->user('id');

  // do not give non-logged in users edit/deletion abilities
  if (empty($user_id)) {
    $event->stopPropagation();
  }

  // Scope the find to the current user
  $event->subject->query['conditions']['Paste.user_id'] = $user_id;
}
```

Instead of passing an anonymous function, we used a controller class method. Any callable is valid for Crud event handling, which uses the CakePHP event handling under the hood.

Now that the dust has settled, here is the state of our application:

- Users can register/login/logout
- Anonymous users can view pastes, as well as create anonymous pastes
- Logged in users will have pastes associated with their users
- Only a logged in user will be able to edit/delete their pastes

Deploying the code

Now that everything is set, lets commit and deploy the code:

```
cd /vagrant/app
git add app/Config/Migration
git add app/Controller/PastesController.php
git add app/Model/User.php
git add app/Model/Paste.php
git commit -m "Scope pastes to users and handle private pastes properly"
git push origin master
git push heroku master
```

Summary

- CakePHP does *not* do automatic password hashing for you. This is a major change from 1.x, so please keep it in mind if you see oddities in login code.
- Component configuration is normally handled within the Controller class property `$components`. You can also modify this at runtime within your `beforeFilter()`, but it is nicer to have it all defined in one place.
- The `AuthComponent` handles logged in and anonymous user access separately, so keep this in mind when trying to allow access to a certain type of user.
- CakeEvent handling can be *terminated* using the `stopPropagation()` method, which is useful for disallowing certain requests from continuing.