Create an interactive production wiki using PHP, Part 1: Introduction and scaffolding

Skill Level: Intermediate

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

13 Feb 2007

This "Create an interactive production wiki using PHP" series creates a wiki from scratch using PHP, with value-added features useful for tracking production. Wikis are widely used as tools to help speed development, increase productivity, and educate others. Each part of the series develops integral parts of the wiki until it is complete and ready for primetime, with features including file uploading, a calendaring "milestone" system, and an open blog. The wiki will also contain projects whose permissions are customizable to certain users.

Section 1. Before you start

This "Create an interactive production wiki using PHP" series is designed for PHP application developers who want to to take a run at making their own custom wikis. You'll define everything about the application, from the database all the way up to the wiki markup you want to use. In the final product, you will be able to configure much of the application at a granular level, from who can edit pages to how open the blog really is.

At the end of this tutorial, you will have learned what goes into making a wiki, considerations in defining your wiki markup, potential pitfalls and challenges in dealing with file uploads, and some implications involved when setting up an environment where content is edited by a collective, rather than an individual. Some of these issues can make wikis tricky. But they can also make them great.

About this series

Part 1 of this series will draw the big picture. You will determine how you want the application to look, flow, work, and behave. You'll design the database and

rough-out some scaffolding. Part 2 focuses on the primary wiki development, including defining the markup, tracking changes, and file uploads. In Part 3, you define some users and groups, as well as a way to control access to certain aspects of individual Wiki pages and uploaded files. Part 4 deals with a Calendaring and Milestones feature to track tasks, to-dos, and progress against set goals. And in Part 5, you put together an open blog to allow discussion of production topics and concerns.

About this tutorial

PHP is a popular language choice when developing Web applications. Binary and source versions are available, and while the binaries are Windows®-specific, the source can be complied for most common platforms. PHP is widely available at most third-party hosting companies. It's easy to learn, but at the same time, it's powerful, flexible, and capable. All of these things make PHP a good choice for writing a wiki engine.

This tutorial focuses on application design. After you get your prerequisites out of the way, you'll get a picture for how the application is going to look, and jump into the database design and scaffolding parts. You're building the beginnings of your custom PHP wiki engine. But rather than refer to it as "your custom PHP wiki engine," you are going to have to give it an easy-to-remember name, such as *Criki*.

Topics include:

- What makes a wiki?
- Criki's architecture
- Database design
- Scaffolding

Prerequisites

It is assumed that you have some experience working with PHP and MySQL. We won't be doing a lot of deep database tuning, so as long as you know the basic ins and outs, you should be fine. You may find it helpful to download and install phpMyAdmin, a browser-based administration console for your MySQL database.

System requirements

Before you begin, you need to have an environment in which you can work. The general requirements are reasonably minimal:

- An HTTP server that supports sessions (and preferably mod_rewrite). This tutorial was written using Apache V1.3 with mod_rewrite enabled.

- PHP V4.3.2 or later (including PHP V5). This was written using PHP V5.0.4
- Any version of MySQL from the last few years will do. This was written using MySQL V4.1.15.

You'll also need a database and database user ready for your application to use. The tutorial will provide syntax for creating any necessary tables in MySQL.

Additionally, to save time, we will be developing Criki using a PHP framework called CakePHP. Download CakePHP by visiting CakeForge.org and downloading the latest stable version. This tutorial was written using V1.1.13. For information about installing and configuring CakePHP, check out the tutorial series titled "Cook up Web sites fast with CakePHP" (see Resources).

Section 2. Introduction

If you've ever had to work on a project with people who were scattered geographically, you've probably used something to collaborate online. Historically, collaboration software has been heavy, overbearing, and pointlessly complicated, often a hindrance to collaboration. But as wikis have risen into common usage, people have put them to good use as a tool in support of collaboration.

Wikis can be fun tools to use. They derive a lot of power from their openness and simplicity. Building your own wiki engine from scratch can be an enlightening exercise as you get a good look at what's going on behind the scenes. But before you start, you should get an idea for what a wiki typically looks like.

What makes a wiki?

There are many wiki flavors, and wiki engines (the software behind the wiki) have been written in just about every language suitable for the Web. But most of these have the same kinds of features:

Browser-based
A wiki typically runs in the context of a Web browser.

Open editing
Generally, anyone who can access the wiki can add to or edit the content.

Wiki markup
A wiki typically uses a sort of meta-language for formatting that acts as a shorthand version of HTML.

Simplicity

Because wikis are used for collaboration, they are generally designed as simple tools that can easily used by most anyone. The focus is on authoring and presentation of information.

Some wikis require user registration and are less open. Some allow for use of raw HTML, rather than a markup language. But these general features hold true when looking at most Wiki engines.

Section 3. Architecture

In the process of planning the development of Criki, there are several things to consider. Start by determining how the application is going to flow and consider what kind of markup will be used, how uploaded files will be stored, what levels of user permissions will be used, how users will be promoted (and demoted), and how pages will be protected (and unprotected). Considering these beforehand will give you a clearer path when coding later.

How is Criki going to flow?

Before you write a line of code, you should spend some time getting a clear picture for how your application is going to flow. You don't have to have precise determinations for UI specifics at this point (though having them won't hurt you), but you need a clear idea of what happens when you submit a page, add a user, edit text, upload a file, etc. As you go through this process, it is important to keep your target audience in mind. Sure, it may make sense to you, but you are not the only one using the application.

Types of users in Criki

What kinds of users will be using Criki? You shouldn't overthink this. Keeping in mind the open nature of wikis in general, defining every possible kind of user would be an exercise in frustration. Keep it simple:

Administrators
The highest level of user. Usually there would only be one administrator. Administrators will have full access to all users and pages within Criki.

Editors
Since Criki is to be used to manage projects, the project managers, team leaders, or tech leads might be editors. They will have some rights over some users and some rights over some pages.

Contributors

This is your average run-of-the-mill user. Contributors will have no rights over other users and minimal rights over some pages.

That should be enough user type definition. Remember, wikis derive much of their power from their openness. As you begin to use Criki more, you may find it beneficial to add other user types, but for now, these three should suffice.

Administering Criki

You are probably going to be the one administering Criki. And this can serve as a good place to start. Not because this will represent the most common of your tasks but because it will represent the most complex of your tasks. And it's the complex tasks, not the common ones, that will shape the bulk of the back end to Criki.

Start by identifying the tasks only an administrator will be able to perform:

1. Changing Criki configuration options
2. Removing a page completely from Criki
3. Removing a user from Criki
4. Removing an uploaded file from Criki
5. Promoting or Demoting any page to any access level
6. Promoting or Demoting any user to any access level

Promotion and demotion could be accomplished with a single button click. Removing a page, user, or uploaded file should have a confirmation box or the ability to recover from mistakes. For simplicity, a confirmation box will be used. Changing Criki configuration options will require its own page.

Editor tasks in Criki

There could be any number of editors using Criki. They should have access to more features than contributors, but they should not be able to access administrative tasks. Specifically, editors should be able to:

- Promote or demote any page to or from editor-level access
- Promote or demote any user to or from editor-level access

Again, promotion and demotion could be accomplished with a single button click. There's no need to overcomplicate the flow.

The security-minded might raise a hand at this point and point out the potential problems that come from this kind of structure. If two editors were to get into a

disagreement, one of them could simply demote the other, causing arguments, escalations, hurt feelings, floods, fires, earthquakes, and plagues of locust. These kinds of concerns are worth considering when dealing with a wiki, but keep in mind that the general nature of a wiki and, indeed, what gives a wiki its power is its openness. There is inherent trust in the power structure.

But there doesn't have to be. Editor rights could just as easily be controlled by configuration settings, indicating what editors can and cannot promote or demote. By giving the administrator this level of control of the application configuration, you give the administrator the ability to configure the application to meet the needs at hand.

User registration in Criki

User registration in Criki could take a couple different forms:

- A user could self-register by entering e-mail address, name, and password.
- Users could only be able to register using a link from an invitation sent by an administrator or editor.
- Administrators and editors could have the ability to register users directly.

In all three cases, the user registration forms will look identical -- an e-mail field, a display name, and a password (and confirmation box). The primary difference between them is who can perform the action.

In the first case, the user registration page is wide open. Anyone can access it and create an account. In the second case, the user registration page requires a valid invitation code to work properly, but the access levels for the page are the same as the first case. In the third case, access to the registration page is restricted.

In all three cases, the user registration process will flow exactly the same way: The form is filled out, the user is created, and a welcome e-mail is sent. Given the similarities of the three options, why do them all? The method used can be controlled using a configuration setting. This allows the administrator to determine which method works best.

Adding pages to Criki

Is Criki going to allow anyone to add pages to the wiki or will this be restricted to contributors? This is another place where putting the choice into a configuration setting can give the administrator power. Regardless of who can add pages to the wiki, the form to add the page will look exactly the same.

Generally, adding a page to a wiki is a two-part process. First, someone creates a link to the page that has not yet been created. When the wiki engine renders this link and sees that the page does not exist, the link is visually changed in some fashion

and is changed to point to the edit page. Then, when someone clicks the link, the edit page is loaded. When the form is submitted, the page is created.

This is a sensible approach to adding pages to Criki. You could give Criki the ability to add a page directly, but unless it is linked to by another page, the page will be an *orphan* in that it won't be visible from anywhere in the wiki unless you make a page to show all pages that are not linked to from any other page.

Once you get in the mindset that adding pages is just like editing pages -- potentially with differing access levels -- all the important questions are answered by identifying what happens when a page is edited.

Editing pages in Criki

Who can edit pages in Criki? Anyone? Only contributors? Only editors? The same people who can add pages? Once again, this can be put into a configuration setting, so the administrator can determine what works best. Generally, if someone can add a page, someone can edit a page. But that may not always be the case. In some cases, the administrator may want to restrict edits by default.

More important to consider is what happens when a page is edited? Specifically:

- Is the page locked for editing when someone clicks the **Edit** button?
- If so, what happens if a user starts an edit and closes the browser?
- If not, what happens if two users edit the same page?
- Can the user preview changes?
- Does the user have to preview changes before submission?
- Is the page history tracked?
- Are you going to provide revision diffs?
- What information are you going to track about edits?

Most of these things wouldn't be of much use as a configuration setting. For simplicity, Criki will not lock a page when a user clicks an **Edit** button. If two users edit the same page at the same time, the last one submitted will win. A full history of edits will be kept, including the user and IP address of the person who edited the page. A user can preview pages, but doesn't have to. Criki won't provide version diffs.

Reading pages in Criki

Now you are getting down to the more mundane tasks in the wiki. Reading pages will be the bulk of what your users will be doing. Reading a page should be as simple as passing the page name to a controller, verifying that the user has rights to

read the page, and retrieving the text for display.

Access verification should be a simple matter, as the permissions on the actual wiki pages will be fairly open. But some pages will be only readable by editors and administrators. Keep in mind that, since the page text was stored in the database in wiki markup, it is during page retrieval and formatting that this markup will need to be rendered into HTML.

Uploading files to Criki

Uploading files is a powerful feature, but it needs to be handled with care. Once you give users the ability to upload files to your application, you open up a world of potential security issues. For example, if you allow any user to upload any kind of file, and if the file was directly accessible via a browser, a malicious user could upload his own PHP file that could be used to manipulate the file system, delete files, elevate access rights, etc.

Therefore, it follows that file types should be restricted or uploaded files should not be directly accessible via a browser, or both. For Criki, you will allow the administrator to control what file types can be uploaded via a configuration setting. The files will be stored on the file system. Taking this approach puts both the power and the responsibility firmly in the hands of the administrator.

If a file is uploaded that has the same name as an existing file, the existing file will be backed up and marked with a revision number.

What markup will Criki use?

No two wiki engines use exactly the same markup. Since you are writing Criki from scratch, you can tailor the markup to suit your specific needs. For our purposes, the markup will be kept fairly simple:

- Wrapping text in three equal signs (===) will cause the text to be wrapped in <h3> tags.
- Wrapping text in three single quotes (''') will cause the text to be wrapped in <i> tags.
- Wrapping text in three exclamation points (!!!) will cause the test to be wrapped <b> tags.
- Wrapping text in three underscores (___) will cause the text to be wrapped in <u> tags.
- Wrapping text in three ampersands (&&&) will cause the text to be wrapped in <pre> tags.
- Wrapping text in triple braces ([[[like this]]]) will cause the text to be turned into a link. If the text does not appear to be a URL, the link will be

interpreted as a link to another page on the wiki, stripping out nonalphanumerics. An alternate name may be provided following a pipe (|). For example:

- [[[ftp://ftp.yourdomain.com]]] will output a link to ftp.yourdomain.com with the text "ftp.yourdomain.com"
- [[[ftp://ftp.yourdomain.com|my ftp site]]] will output a link to ftp.yourdomain.com with the text "my ftp site"
- [[[how to do it]]], [[[howtodoit]]], [[[HowToDoIt]]] and [[[how_to_do_it]]] will all output links to the wiki page "howtodoit." The text of the link will be whatever was wrapped in the braces.
- [[[howtodoit|Instructions]]] will output a link to the wiki page "howtodoit" with the text "Instructions."

- Consecutive lines beginning with an asterisk (*) will be rendered as an unordered list (<ul>).
- Consecutive lines beginning with a pound sign (#) will be rendered as an ordered list (<ol>).
- Three dashes (---) on a line by themselves will be rendered as an <hr /> tag.
- URLS beginning with 'http://' will automatically turned into hyperlinks.

When applying the markup, it will be important to consider the order in which you apply the various markups. Specifically, the &&& markup should be applied first, as it is most likely to contain code fragments and comments that could be interpreted as markup. You should experiment with the application of the various markups to determine what happens when markups are nested. Criki should handle nested markup gracefully. That is to say: It's not as important to render the markup properly as it is to *not* render the markup badly.

For example, consider the following lines.

```
===I don't want to fight
*unless I have to===
```

What's the proper way to render these lines? Taken literally, they would be rendered something like Listing 1.

Listing 1. Improper rendering of lines

```
<h3>I don't want to fight
<ul>
<li>unless I have to</li>
</ul>
</h3>
```

That's probably not going to look the way it's intended to or the way the user wanted

it to. But it's not as simple as grabbing the biggest match for your patterns (greedy pattern matching). Consider the following lines:

Listing 2. Lines to be rendered

```
===I don't want to fight===
*unless I have to
*And I don't want to
===If you have to fight===
*Fight Dirty
*Win
```

Greedy pattern matching might render the text as shown below.

Listing 3. Poorly rendered lines

```
<h3>I don't want to fight===
*unless I have to
*and I don't want to
===If you have to fight</h3>
<ul>
<li>Fight Dirty</li>
<li>Win</li>
</ul>
```

Again -- not what you're after. You don't have to solve this problem right now, but it's a good idea to keep it in the back of your mind, so you can chew on it while you design the rest of the Criki.

Section 4. How will Criki do all that?

So you know basically how you want Criki to work. Now you can start working out how you are going to put it together. Since we're doing this in CakePHP, we'll be using a standard Model-View-Controller (MVC) design.

MVC overview

Model-View-Controller (MVC) refers to a specific method of designing an application. While a full discussion of MVC is outside the scope of this tutorial, the main points will be touched on briefly.

MVC breaks an application into three distinctive pieces: Model, View, and Controller. Each part of the application is responsible for specific tasks.

The Model is primarily concerned with saving and retrieving data. All database interaction takes place within the Model. The Model takes data from the Controller and saves it into the database, and the Model retrieves data from the database and

passes it on to the Controller.

The Controller is primarily concerned with logic. This is where the bulk of the application workload is done. The Controller takes data from the Model, applies logic, and passes output to the View to be displayed to the user. The Controller also takes input from user, applies logic, and makes decisions based on the input (such as passing the input to the Model to be saved or passing other information to the View to be displayed).

The View is primarily concerned with presentation. This is where the HTML lives. The View takes output from the Controller, and formats it for display to the user. In MVC, application logic is separated completely from presentation logic.

Taking the MVC approach has many benefits. If you need to switch to a new database vendor, for example, you can update the Model code without having to wade through presentation and application logic. To change the look of the application, you can update the View code without having to touch the application logic or database code. Keeping the three layers separate generally makes each layer and, thus, the application as a whole, easier to maintain.

Designing the database

Now that you have a fairly good idea for how you want Criki to be used and how it will be put together, you can start thinking about how Criki will go about accomplishing its various tasks. Looking back through the list of tasks, you should start to get an idea for how the database is going to look. You're going to need:

1. **A table to hold user data** -- This will need to hold at least a user name (unique), e-mail (unique), password, and access level. Other information you may want would include a numeric ID for the user, creation date, and last login date.

2. **A table to hold page data** -- This will need to hold at least a page name (unique), page content, page access level, last-modified date, user who last edited the page, and the IP address of that user (for when user registration is not required). Other information you may want would include a numeric ID for the page, as well as maybe a revision number and last-accessed timestamp.

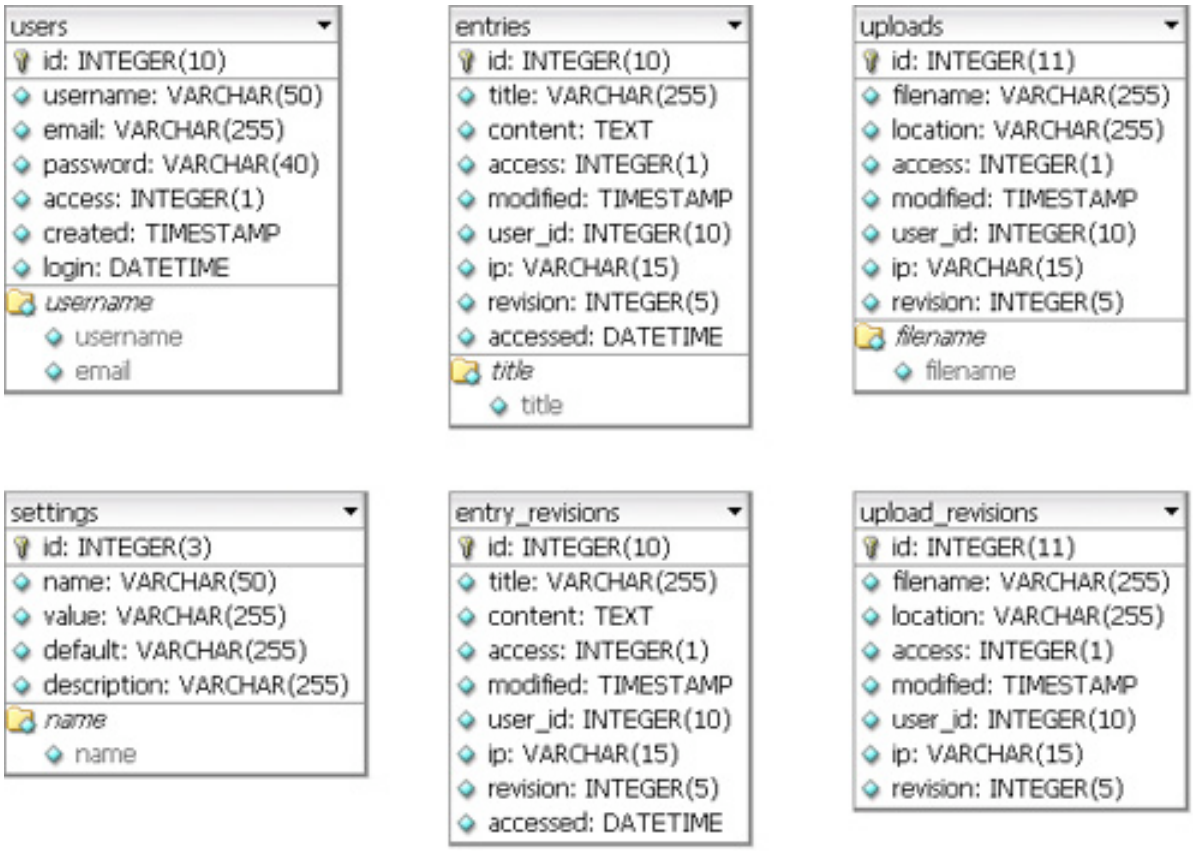
3. **A table to hold uploaded file data** -- You could probably get by without this, but by putting uploaded file information into its own table, you eliminate the need to walk the directory where files have been uploaded, while making it easier to store things like file access rights. This table will need to hold at least a file name (unique), location, revision number, and access rights. Other information you may want would include a numeric ID for the file, user who uploaded the file, and IP address of the user uploaded the file (for times when file upload does not require user registration).

4. **A table to hold page history** -- This will look exactly like the page table, with the exception that page name need not be unique.

5. **A table to hold file history** -- This will look exactly like the file table, with the exception that page name need not be unique.

6. **A table to hold configuration settings** -- This would probably work best as an entity-attribute-value (EAV) table. An EAV table is designed to hold an arbitrary number of values by using generic column names such as "id" and "value," rather than specific column names like "enable_fileupload" or "access_level." By storing the configuration settings in this fashion, you allow yourself room to easily add additional configuration options to Criki in future releases, while at the same time making it easier for the administrator to change the configuration options via a browser interface, rather than via a configuration file. For Criki, you're going to add a field called "description" that will be used as the label for the configuration field. See Figure 1.

Figure 1. Criki tables



Six tables -- that's all you should need to get the basics of Criki in place. Go ahead and run the SQL script included with the source code now. Then we'll use some built-in CakePHP shortcuts to help us get a leg up on the application.

Section 5. Scaffolding

You have the database created. It's time to get cracking. CakePHP ships with a script called Bake that will help get you up and running, but before you bake your controllers and views, you need to create your models.

Creating the models

All the models you create will go in the app\models directory where you installed Cake. Keep in mind that in CakePHP, models are singular, while database table names are plural. For example, the basic model for the entries table would be called entry.php and would look like this:

Listing 4. Basic model entries table

```
<?php
class Entry extends AppModel {

        var $name = 'Entry';

}
?>
```

Yes -- CakePHP is smart enough to know that the plural form of "entry" is "entries."

Save the entry model and create models for entry_revision, settings, upload, upload_revision, and user. All the models will look almost identical to the one above -- just a class definition and a *$name*. If you're not sure you got them right, check the models in the accompanying source code.

Baking the controllers

Now that you have the database and models in place, you can use the Bake code generator tool that ships with CakePHP to get a quick scaffold up for Criki. Before you proceed, you should back up the app directory for your CakePHP installation, as Bake will overwrite some files that exist there. If you are doing all of your work on a fresh install, you shouldn't need to back anything up. Additionally, if this is your first time using the Bake code generator, you may need to change the value of max_input_time in the php.ini file, as Bake might time out if you take too long.

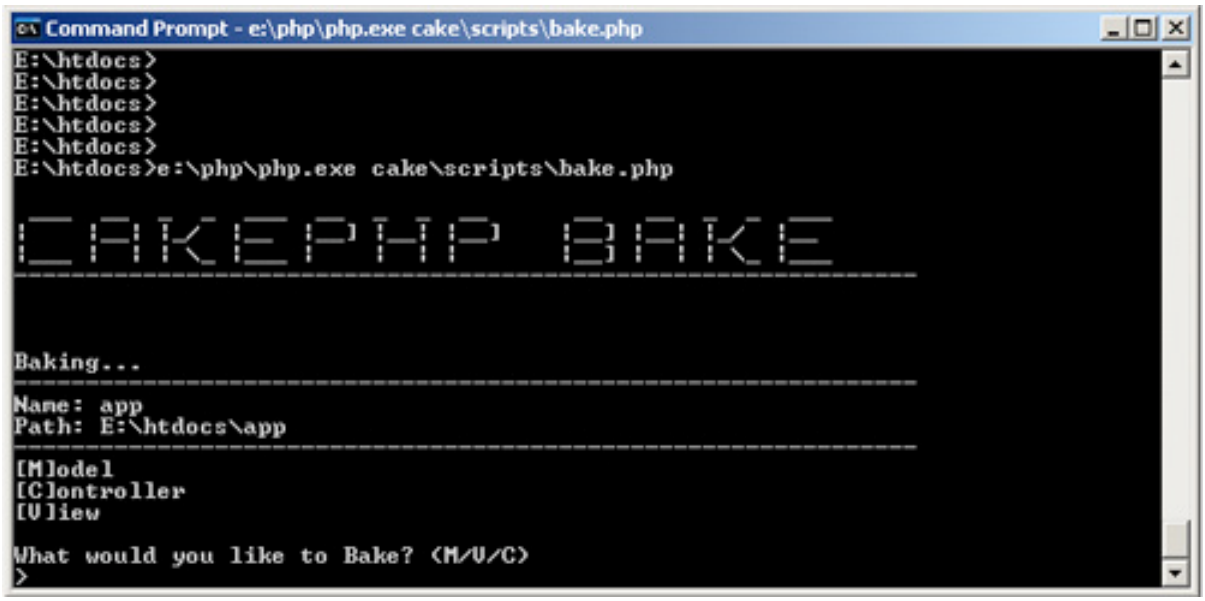
In this case, running Bake is a snap. cd into the directory where you installed CakePHP and execute the bake.php script located in cake\scripts\. For example, if you installed CakePHP in the directory \htdocs, you need to cd into \htdocs and run `php cake\scripts\bake.php`.

If you have problems getting Bake to run, make sure you are in the directory where

you installed Cake and that the php.exe is in a directory located in your path. If it's not, you should just be able to specify the full path to the php.exe binary.

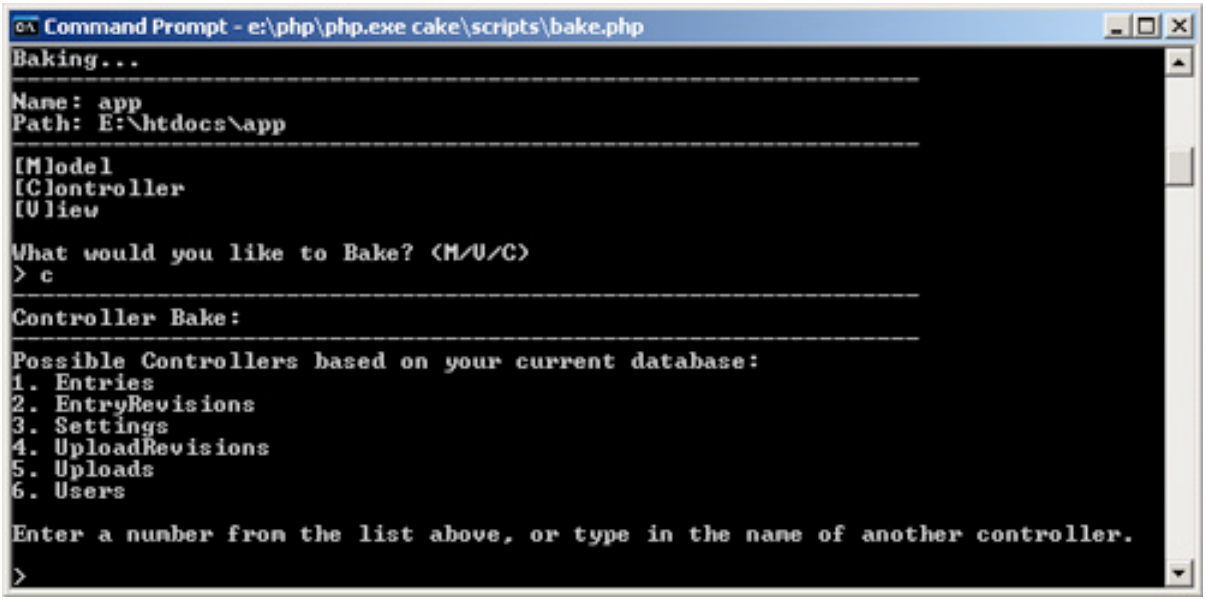
To start, you'll be presented with the Bake menu.

Figure 2. Bake menu



Since you have already built the models, you can jump right in to building the controllers.

Figure 3. Building the controllers



Start from the top and work your way down:

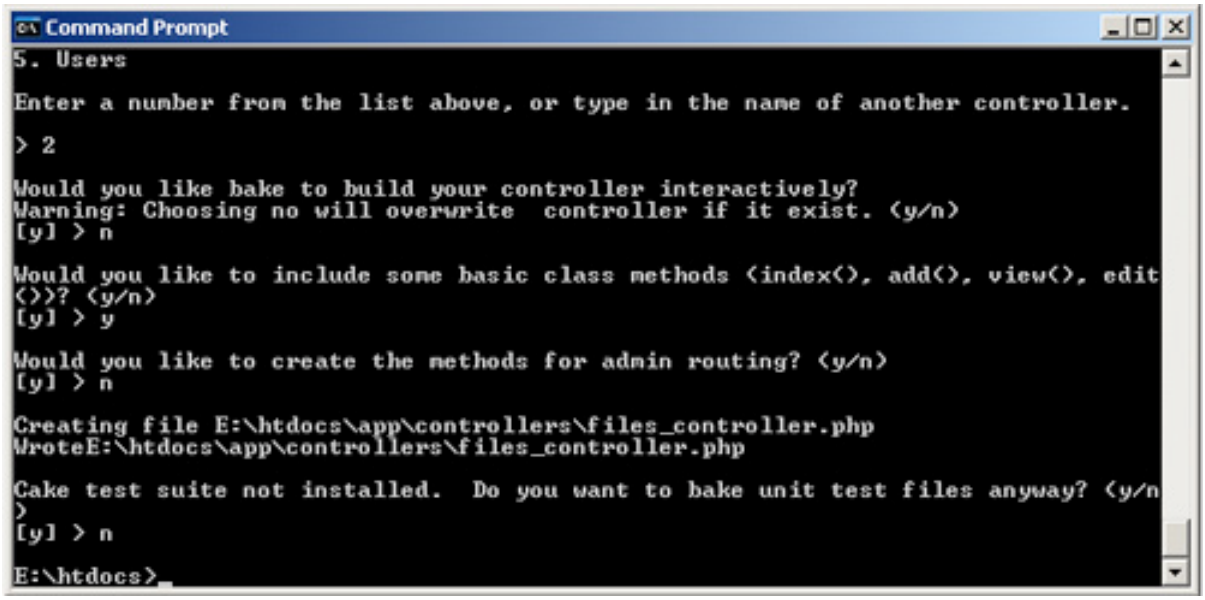
1. Bake will ask, "Would you like bake to build your controller interactively?" Unless you feel like going down that route, press **n** to skip interactive baking.

2. Bake will then ask, "Would you like to include some basic class methods?" Press **y** for yes.

3. Bake will then ask, "Would you like to create the methods for admin routing?" Go ahead and press **n** for no.

4. Finally, Bake may ask if you want to build unit test files. The unit test files will not be covered in the tutorial series. Press **n** and Bake will exit.

Figure 4. Exiting Bake

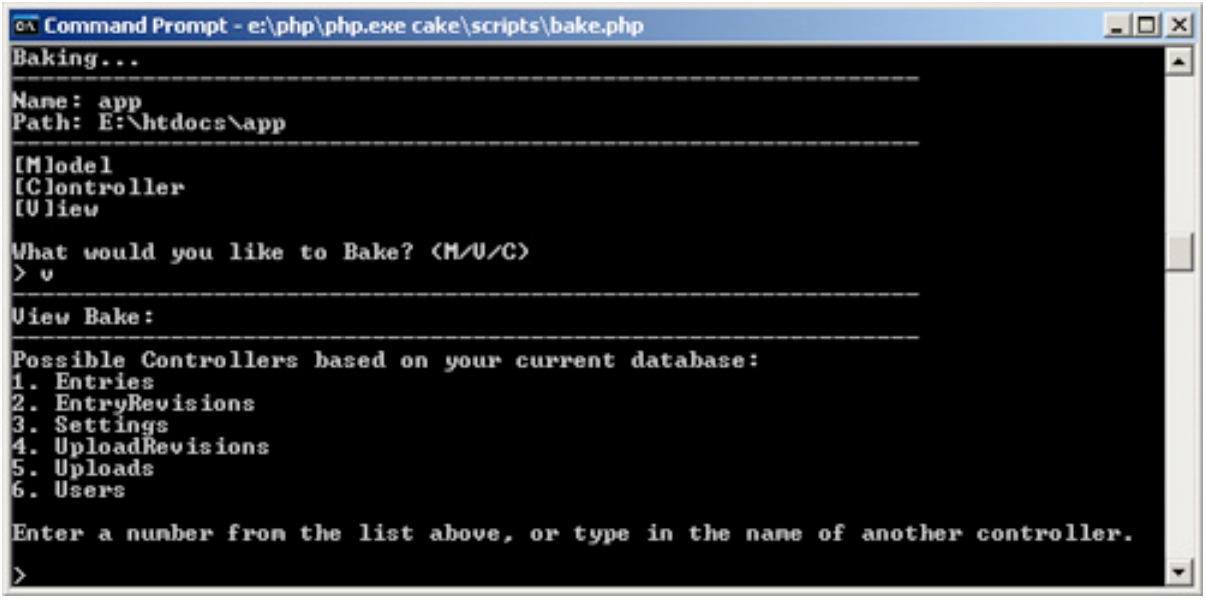


Repeat these steps for each of the controllers.

Baking the views

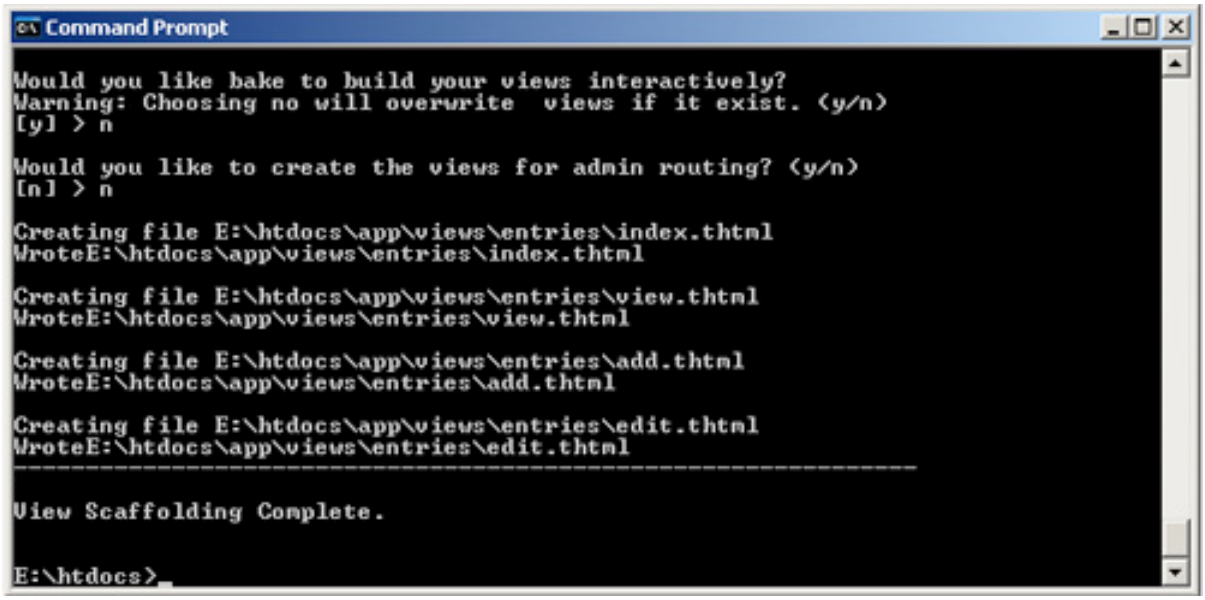
Baking the views is almost exactly like baking the controllers. When you select views from the Bake menu, you are again presented with a list of views that you can create.

Figure 5. Select views from the Bake menu



Start again from the top and work your way down. Bake will ask if you want to build your views interactively, and if you want to create the views for admin routing. Say **no** to both and just let Bake build up the basic views.

Figure 6. Letting Bake build up the basic views



Repeat the above steps for each of the views. Once you're done, access your application and take a look around. http://localhost/users should take you to the users list, where you can add, edit and view users. Likewise, http://localhost/entries, http://localhost/entry_revisions, http://localhost/settings, http://localhost/uploads, and http://localhost/upload_revisions should take you to the various scaffolds for those controllers, as well. Feel free to test out putting some data in. In Part 2, start customizing the views and controllers.

Section 6. Summary

You've got your database together, and you've cobbled together some basic views. But you're not going to want to use the default layout for Criki. Spend some time thinking about how you want it to look. For example, where do you want your links? What colors do you want to use?

Create a default layout (app/views/layouts/default.thtml) and experiment with some different looks. Use the default template in cake/lib/views/templates/layouts/default.thtml as a guide when building your new layout. Just make sure you don't change the original.

Happy coding. Be sure to read Part 2, where we focus on the primary wiki development, including defining the markup, tracking changes, and file uploads.

Downloads

Description	Name	Size	Download method
Part 1 source code	os-php-wiki1.source.zip	4KB	HTTP

Information about download methods

Resources

Learn

- It seems natural to start with the Wikipedia entry for wiki.
- Check out WikiWikiWeb for a good discussion about wikis.
- Visit the official home of CakePHP.
- Check out the "Cook up Web sites fast with CakePHP" tutorial series for a good place to get started.
- 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.
- Find out more about how PHP handles sessions.
- Check out the official PHP documentation.
- Start with the "Considering Ajax" series to learn what you need to know before using Ajax techniques when creating a Web site.
- Read the five-part "Mastering Ajax" series on developerWorks for a comprehensive overview of Ajax.
- CakePHP Data Validation uses PHP Perl-compatible regular expressions.
- See a tutorial on "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.
- Source material for creating users.
- Check out the Model-View-Controller on Wikipedia.
- 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.
- PHP.net is the 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.

- 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.
- Visit Safari Books Online for a wealth of resources for open source technologies.

Get products and technologies

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

Discuss

- Participate in developerWorks blogs and get involved in the developerWorks community.
- Participate in the developerWorks PHP Developer Forum.

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.

Create an interactive production wiki using PHP, Part 2: Developing the basic wiki code

Fueling Criki with user registration, entry storage, and custom markup rendering

Skill Level: Intermediate

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

06 Mar 2007

This "Create an interactive production wiki using PHP" tutorial series creates a wiki from scratch using PHP, with value-added features useful for tracking production. Wikis are widely used as tools to help speed development, increase productivity, and educate others. Each part of the series develops integral parts of the wiki until it is complete and ready for primetime, with features including file uploading, a calendaring "milestone" system, and an open blog. The wiki will also contain projects whose permissions are customizable to certain users.

Section 1. Before you start

This "Create an interactive production wiki using PHP" series is designed for PHP application developers who want to to take a run at making their own custom wikis. You'll define everything about the application, from the database all the way up to the wiki markup you want to use. In the final product, you will be able to configure much of the application at a granular level, from who can edit pages to how open the blog really is.

At the end of this tutorial, Part 2 of a five-part series, you will have the basics of your wiki up and running, including user registration, page creation and editing, history tracking, and file uploads. It sounds like a lot, but if you've completed Part 1, you're well over halfway there.

About this series

Part 1 of this series draws the big picture. You determine how you want the application to look, flow, work, and behave. You'll design the database and rough-out some scaffolding. Part 2 focuses on the primary wiki development, including defining the markup, tracking changes, and file uploads. In Part 3, you define some users and groups, as well as a way to control access to certain aspects of individual wiki pages and uploaded files. Part 4 deals with a Calendaring and Milestones feature to track tasks, to-dos, and progress against set goals. And in Part 5, you put together an open blog to allow discussion of production topics and concerns.

About this tutorial

This tutorial focuses on writing the core code for the wiki engine. With the database in place, your next task is getting the wiki engine up and running, including user creation, signing in, rendering the markup, page creation, file uploads, and more. With these tasks done, your application ("Criki") will take a definite shape. Covered topics include:

- User registration
- Page creation
- Rendering markup
- File uploads

Prerequisites

It is assumed that you have some experience working with PHP and MySQL. We won't be doing a lot of deep database tuning, so as long as you know the basic ins and outs, you should be fine. You may find it helpful to download and install phpMyAdmin, a browser-based administration console for your MySQL database.

System requirements

Before you begin, you need to have an environment in which you can work. The general requirements are reasonably minimal:

- An HTTP server that supports sessions (and preferably mod_rewrite). This tutorial was written using Apache V1.3 with mod_rewrite enabled.
- PHP V4.3.2 or later (including PHP V5). This was written using PHP V5.0.4

- Any version of MySQL from the last few years will do. This was written using MySQL V4.1.15.

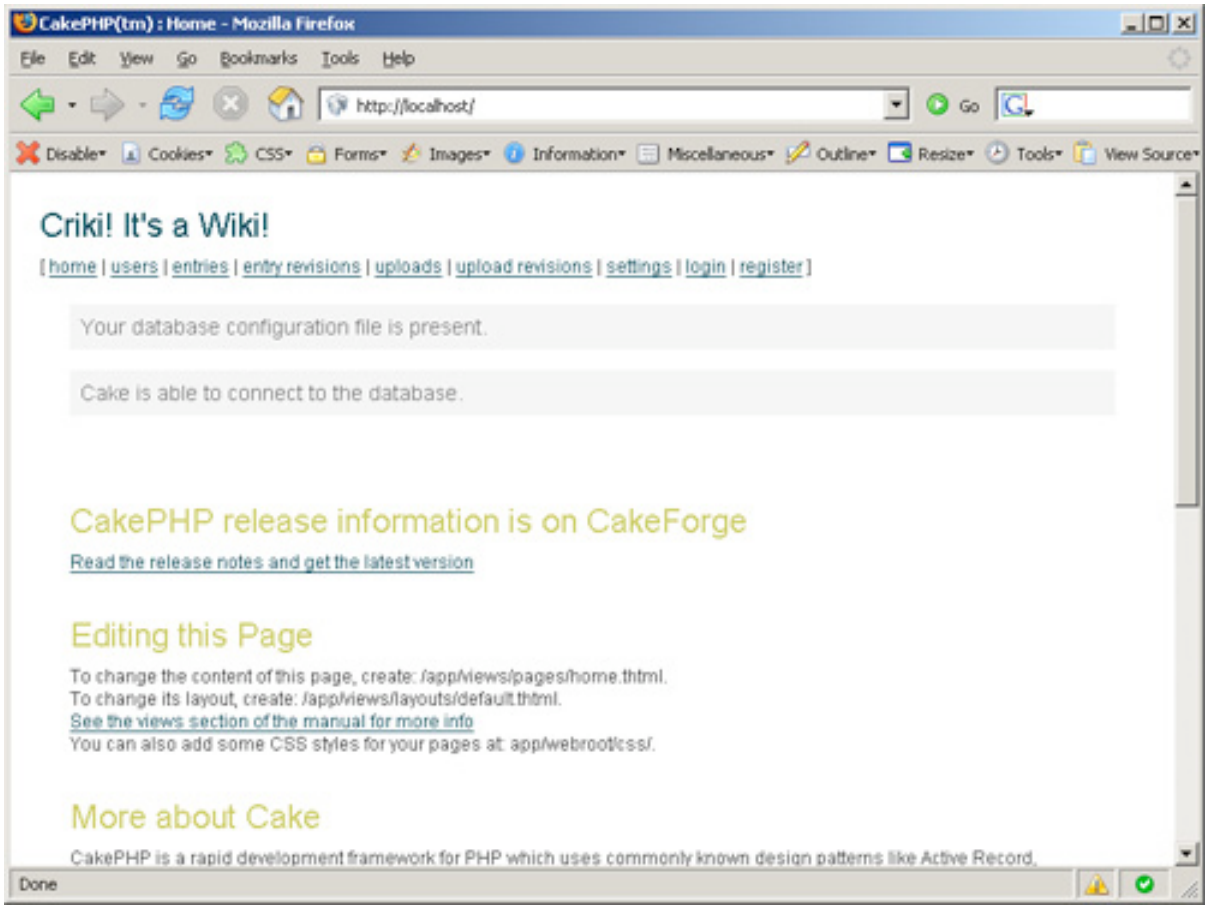
You'll also need a database and database user ready for your application to use. The tutorial will provide syntax for creating any necessary tables in MySQL.

Additionally, to save time, we will be developing Criki using a PHP framework called CakePHP. Download CakePHP by visiting CakeForge.org and downloading the latest stable version. This tutorial was written using V1.1.13. For information about installing and configuring CakePHP, check out the tutorial series titled "Cook up Web sites fast with CakePHP" (see Resources).

Criki so far

At the end of Part 1, you were given the opportunity to redesign the default layout into something that better suited your own tastes. How did you do? It's OK if you didn't get a chance to work on this particular piece. The source code for this tutorial contains a basic layout that includes links to the various controllers. You will be editing this as the series goes on. Copy the file app/views/layouts/default.thtml from the code archive into your own app/views/layouts/ directory. It should look like Figure 1.

Figure 1. Criki so far



If you did get a chance to work on the layout and didn't link to the various controllers yourself, you can work the following code into your layout wherever you want. The code in Listing 1 will give you a horizontal menu. Feel free to rework it as you see fit to suit your own design.

Listing 1. The horizontal menu

```
[
<?php echo $html->link('home','/') ?>
|
<?php echo $html->link('users','/users') ?>
|
<?php echo $html->link('entries', '/entries') ?>
|
<?php echo $html->link('entry revisions', '/entry_revisions') ?>
|
<?php echo $html->link('uploads', '/uploads') ?>
|
<?php echo $html->link('upload revisions', '/upload_revisions') ?>
|
<?php echo $html->link('settings', '/settings') ?>
]
```

Now that you have the layouts taken care of, you can get to work on the core code for Criki. A logical place to start would be with the user registration code.

Section 2. Writing the core code

Now that the basic structure has been baked and you have a layout more suited to your tastes, it's time to dive into writing the core code for Criki.

The core code is broken into three basic sections: user code, page code, and history code. The user-related code covers the basic needs for simple user registration and login. Don't worry at this point about types of users and permissions -- that's covered in Part 3. The page code covers creating, editing, and reading wiki entries, including rendering the markup. And the history code will keep track of page revisions.

User registration

The first piece of Criki you need to work on is the user code. Users need to be able to register for accounts, log in, and log out. There will be a configuration setting later to control if a user has to register before he can edit, but for now, just build out the basic user registration code.

Regardless of the configuration settings, basic user registration will look pretty much the same. You will always need to verify the following:

1. That the username is available

2. That the e-mail address has not been used to register an account already

There's a lot of additional validation you could do at user registration: minimum/maximum username and password values, e-mail validation, etc. But this should be enough for the basic user registration.

Register view

For CakePHP, you'll need to make a register view in the app/views/users directory and a register action in the users controller (app/controllers/users_controller.php). The register view will look like Listing 2.

Listing 2. The register view

```
<?php echo $html->formTag('/users/register') ?>
<p>Please fill out the form below to register an account.</p>
<label>Username:</label>
<?php echo $html->inputTag('User/username') ?>
<?php echo $html->tagErrorMsg('User/username', $username_error) ?>

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

<label>Email Address:</label>
<?php echo $html->inputTag('User/email') ?>
<?php echo $html->tagErrorMsg('User/email', $email_error) ?>

<?php echo $html->submitTag('register') ?>
</form>
```

This is a fairly simple registration form. It includes the basic form elements, and some CakePHP error message placeholders for invalid registration errors.

Register action

The register action in the users controller will look like Listing 3.

Listing 3. The register action

```
function register() {
  $this->set('username_error', 'username is required');
  $this->set('password_error', 'password is required');
  $this->set('email_error', 'email is required');
  if (!empty($this->data) && $this->User->validates($this->data)) {
    if ($this->User->findByUsername($this->data['User']['username'])) {
      $this->User->invalidate('username');
      $this->set('username_error', 'username already in use');
    } else if ($this->User->findByEmail($this->data['User']['email'])) {
      $this->User->invalidate('email');
      $this->set('email_error', 'email address already in use');
    } else {
      $this->data['User']['password'] = md5($this->data['User']['password']);
      $this->User->save($this->data);
      $this->Session->write('User',
$this->User->findByUsername($this->data['User']['username']));
```

```
      $this->Session->setFlash('Thank you for registering.');
      $this->redirect('/');
    }
  } else {
    $this->validateErrors($this->User);
  }
}
```

This action sets the default error messages. Check to make sure that the username and e-mail address are not in use, and if the user is acceptable, the user data is saved into the Users table (after changing the password to a hash value). The user's data is read back out of the database and set into session. This will serve to determine if a user is logged in. The reason the information is read back out of the database, rather than just using the data from the form submission, is because the database contains the default access level of the newly created user. This will be used later when dealing with permissions. Finally, the user is forwarded to the home page on successful registration.

Section 3. Login/logout

OK -- you've got user registration tackled. Criki has its first piece of nonscript-generated code. But you're just getting warmed up. It's great that your users can register, but they also need to be able to use their accounts. This starts with login and logout. In this section, you will create the login and logout actions as well as a login view.

Login view

The login view need not be complicated. All you need is the e-mail address and the password. Given the simple requirements, the login view will look like Listing 4.

Listing 4. Login view

```
<?php echo $html->formTag('/users/login') ?>
<p>Please log in.</p>

<label>Email Address:</label>
<?php echo $html->inputTag('User/email') ?>

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

<?php echo $html->submitTag('login') ?>

<?php echo $html->tagErrorMsg('User/email', $login_error) ?>

</form>
```

The form is very straightforward. Now you need a corresponding login action.

Login action

On login, you will need to verify the user's password and save the user data into the session if it's valid. The login action will look like Listing 5.

Listing 5. Login action

```
function login() {
  $this->set('login_error', );
  if ($this->data) {
    $results = $this->User->findByEmail($this->data['User']['email']);
    if ($results && $results['User']['password'] == md5($this->data['User']['password'])) {
      $this->Session->write('User', $results['User']);
      $results['User']['login'] = date("Y-m-d H:i:s");
      $this->User->save($results);
      $this->redirect('/');
    } else {
      $this->User->invalidate('email');
      $this->User->invalidate('password');
      $this->set('login_error', 'invalid login');
    }
  }
}
```

The action also updates the login field with the current time and date. That's all there is to it.

Logout action

The logout action is much simpler. All you need to do is delete the user information from the session and redirect to something. The action will look like Listing 6.

Listing 6. Logout action

```
function logout() {
  $this->Session->delete('User');
  $this->redirect('/');
}
```

You will note that many of the redirects point back to / or the root directory. Right now, that page just looks like the default CakePHP installation. Later, you will edit this page to be a better landing page for Criki.

Section 4. User cleanup

Your users can now come to Criki, they can create their own accounts, and they can log in and log out of the application. Now that you have some basic user functionality in place, you should clean up the user's controller by deleting unneeded actions and

locking down the edit action, and you should change the default layout to give access to the login/logout/register functionality.

User's controller cleanup

A couple other things should be done to the user's controller while you are here. The delete action won't be needed for now, so go ahead and delete it. Additionally, the edit action should be changed so users can only edit their own information. When you're done, the new edit controller will look something like Listing 7.

Listing 7. The new edit controller

```
function edit($id = null) {
  if ($this->Session->check('User')) {
    $user = $this->Session->read('User');
    if(empty($this->data)) {
      if(!$id) {
        $this->Session->setFlash('Invalid id for User');
        $this->redirect('/user/index');
      }
      $this->data = $this->User->read(null, $id);
    } else {
      if ($id == $user['id']) {
        $this->cleanUpFields();
        if($this->User->save($this->data)) {
          $this->Session->setFlash('The User has been saved');
          $this->redirect('/user/index');
        } else {
          $this->Session->setFlash('Please correct errors below.');
        }
      }
    }
  }
}
```

The edit action will not check to ensure that the user is logged in or that the ID of the user being edited matches the ID of the user performing the edit.

Default layout change

You can access the `$session` variable in the default layout and display logout links only to users that are logged in, while displaying login/register links only to users that are logged out.

Edit app/views/layouts/default.thtml and add the following code to your link menu.

Listing 8. Changing the default layout

```
<?php if ($session->check('User')) {
  echo $html->link('logout', '/users/logout');
} else {
  echo $html->link('login', '/users/login');
  echo ' | ';
  echo $html->link('register', '/users/register');
} ?>
```

That should be it for now as far as the users go. Now you can get to work on the entries controller.

Section 5. Creating pages

You now have the basics of what you need out of the users code. It's time to move on to the meat and potatoes of the wiki: the page code. Your users will need to be able to create entries for Criki, and it would probably be helpful if those entries could be read, as well.

As discussed in Part 1, creating pages is not substantially different from editing pages. Since an edit view and an edit action have already been baked, you can leverage these to get page creation up quickly. But first, you will need to make a quick change to the entries model.

Relating the entries to the users

You want to be able to access user information (specifically, the name of the user who last modified the entry) from the Entries model. In CakePHP, you can do this by establishing a `belongsTo` relationship between the models. You've already laid the groundwork for this in the way the tables were created. Edit app/models/entry.php and replace it with the following from Listing 9.

Listing 9. Establishing a belongsTo relationship between the models

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

Now entry related queries will return the user information for each entry.

Modifying the edit view

The edit view, as it was baked, contains a lot of fields that need to be removed. Really, the field you need when editing a page is content. It can be helpful to include the ID of the entry being edited as a hidden field as well, and you will set the title of the page as a variable and display it in the header (as well as setting it as a hidden

form element). For good measure, you should add a cancel link that points back to the view for the entry. The redacted edit view should look like Listing 10.

Listing 10. Redacted edit view

```
<h2>Edit Entry :   <?php echo $entry_title?></h2>
<form action="<?php echo $html->url('/entries/edit/'.$html->tagValue('Entry/id')); ?>"
method="post">
<div class="optional">
  <?php echo $form->labelTag( 'Entry/content', 'Content' );?>
  <?php echo $html->textarea('Entry/content', array('cols' => '60', 'rows' => '10'));?>
  <?php echo $html->tagErrorMsg('Entry/content', 'Please enter the Content.');?>
</div>
<?php echo $html->hidden('Entry/id')?>
<?php echo $html->hidden('Entry/title')?>
<div class="submit">
  <?php echo $html->submit('Save');?>
  <?php echo $html->link('Cancel', '/entries/view/' . $entry_title);?>
</div>
</form>
```

That should be all you need for the edit view. The real work here will be done in the edit action of the entries controller.

Modifying the edit action

Take a look at the edit action as it was baked in Listing 11.

Listing 11. Baking the edit action

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

Translated into English, the action would read, "If there is no form submission, display the information for the ID that was passed, if any. Otherwise, clean up the submitted date fields and save the data."

In English, what you need this action to do is something like this: "If there is no form submission, display the information for the title that was passed, if any. Otherwise, append appropriate information to the data and save it." In this case, you will be appending certain data to the form submission, such as the ID of the user that is performing the edit and the IP address. The new edit action will look like Listing 12.

Listing 12. The new edit action

```
function edit($title = null) {
  if(empty($this->data)) {
    if(!$title) {
      $this->Session->setFlash('Invalid Entry');
      $this->redirect('/entries/index');
    }
    $this->data = $this->Entry->findByTitle($title);
    if ($this->data) {
      $this->set('entry_title', $this->data['Entry']['title']);
    } else {
      $this->data['Entry']['title'] = $title;
      $this->set('entry_title', $title);
    }
  } else {
    $user_id = 0;
    if ($this->Session->check('User')) {
      $user = $this->Session->read('User');
      $user_id = $user['id'];
    }
    $this->data['Entry']['user_id'] = $user_id;
    $this->data['Entry']['ip'] = $_SERVER['REMOTE_ADDR'];
    if($this->Entry->save($this->data)) {
      $this->Session->setFlash('The Entry has been saved');
      $this->redirect('/entries/view/'.$this->data['Entry']['title']);
    } else {
      $this->Session->setFlash('Please correct errors below.');
    }
  }
}
```

Don't worry about revision number right now. You'll be adding that in once we get to the revisions section.

Section 6. Entries cleanup

Now that you have add/edit working the way you want it to, you'll need to do some cleanup on the other views and actions related to the entries.

Entries controller cleanup

Go ahead and delete the add and delete actions in the entries controller. The add action is essentially replaced by the edit action, and the delete action you don't need for now. You'll add another one later -- once you have user permissions in place. Additionally, you need to change the view action. Instead of redirecting back to the index action for invalid entry titles, the user should be directed to the edit action.

```
$this->redirect('/entries/edit/' . \
preg_replace("/[^a-z]/", ,
strtolower($title)));
```

Note that all titles are converted to lowercase and stripped of nonalphabetic

characters.

Entries views cleanup

For the entries views, you can go ahead and delete the add view, as it won't be in use. For the other views, you want to keep in mind that the wiki will be driven by passing in the title of the wiki page, not the ID.

Index view

The index view should be edited down to just show the title, modified date, and user. Most of the other information isn't necessary to display at this level. More importantly, the edit and view links need to be changed to pass in the title, not the ID. When you're done, the view will look something like Listing 13.

Listing 13. Index view

```
<div class="entries">
<h2>List Entries</h2>
<table cellpadding="0" cellspacing="0">
<tr>
  <th>Title</th>
  <th>Modified Date</th>
  <th>Modified By</th>
  <th>Actions</th>
</tr>
<?php foreach ($entries as $entry): ?>
<tr>
  <td><?php echo $entry['Entry']['title']; ?></td>
  <td><?php echo $entry['Entry']['modified']; ?></td>
  <td><?php echo $entry['User']['username']; ?></td>
  <td class="actions">
    <?php echo $html->link('View','/entries/view/' . $entry['Entry']['title'])?>
    <?php echo $html->link('Edit','/entries/edit/' . $entry['Entry']['title'])?>
   </td>
</tr>
<?php endforeach; ?>
</table>
</div>
```

This gives you the basic information that's worth displaying in the index. Now you need to modify the view used when displaying an entry.

The view of the view

The existing view of the view contains far more information than needs to be displayed at this time. Replace app/views/entries/view.thtml with Listing 14.

Listing 14. The view of the view

```
<div class="entry">
<h2><?php echo $entry['Entry']['title']?></h2>
<p>Modified on <?php echo $entry['Entry']['modified']?>
```

```
by <?php echo $html->link($entry['User']['username'], $entry['Entry']['user_id'])?>
 [ <?php echo $html->link('Edit Entry',   '/entries/edit/' . $entry['Entry']['title']) ?>
]</p>
<?php echo $entry['Entry']['content']?>
</div>
```

This new view contains all the base information you need: the title of the page, the content, who modified it last, and when, and a link to edit the page. Simple enough.

That gets your information into the wiki. Now for the hard part: getting it back out the way you want it.

Section 7. Rendering the markup

Your users can create and edit entries in Criki. Now all you need to do is display them in a way that makes them readable. That means rendering the markup.

You may have noticed that the wiki markup wasn't rendered before the data was saved. This was by design. For one thing, if the wiki markup was rendered into HTML before writing the content to the database, the markup would have to be de-rendered whenever a user wanted to edit an entry. And de-rendering the entry content is more trouble than it's worth. De-rendering probably isn't even a word.

A better way to approach the problem, would be to write in the content as the user has submitted it and render the wiki markup when the entry is viewed.

Markup refresher

You'll need to look back at Part 1 for a list of wiki markup that Criki will use. This list failed to include proper handling of newlines. Specifically in Criki, newlines will behave as follows:

- A newline on a line by itself will be rendered as [missing text]
- A newline at the end of any list element (lines that begin with * or #) will signify the end of that list element.
- A newline during any open list or paragraph will close the list or paragraph

This will make more sense as you begin to work on the wiki markup code, itself. Before you start, you will find it helpful to make an entry full of markup for testing purposes.

Setting up a test entry

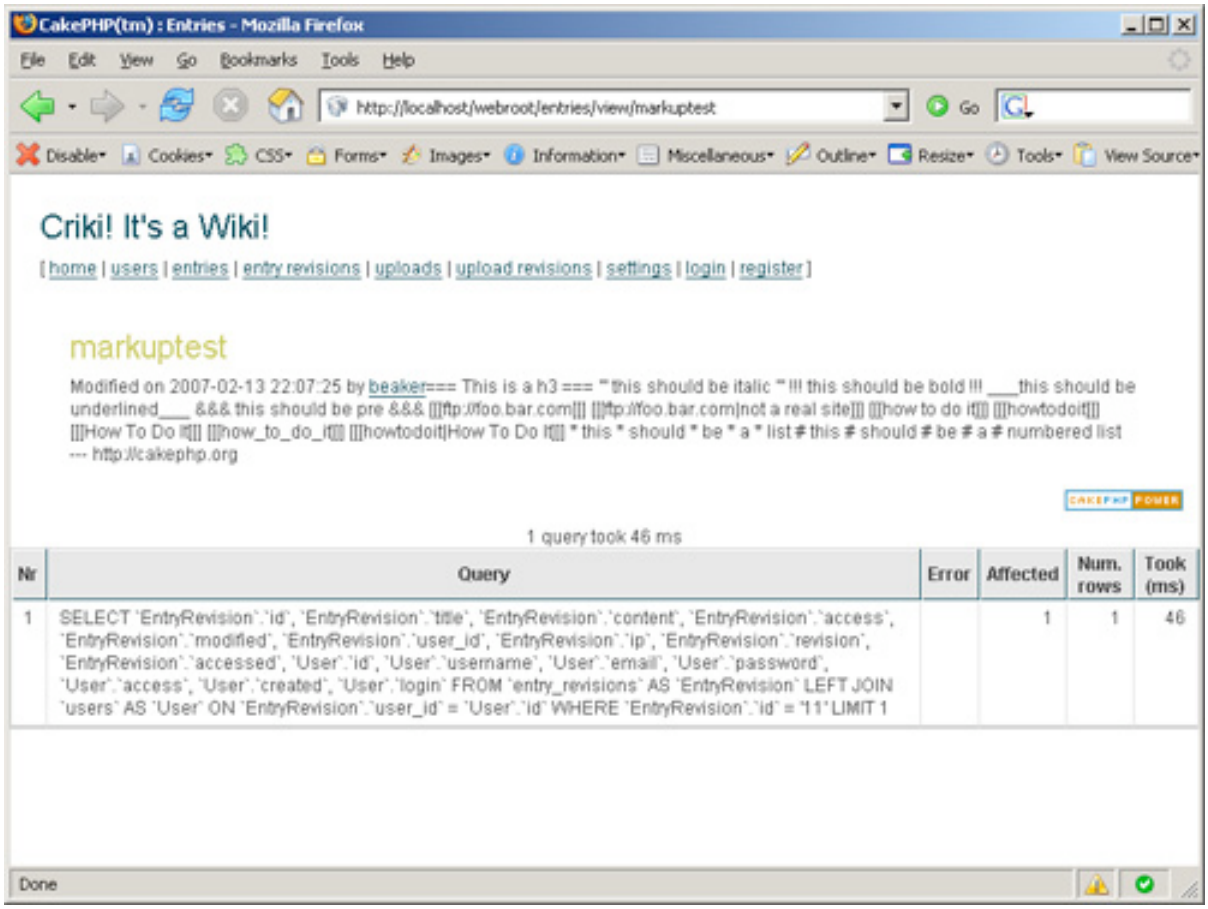
When testing the markup rendering, it will be helpful to have a test entry you can view, which contains an example of each markup. Use the following text for this purpose.

Listing 15. Testing the markup

```
=== This is a h3 ===
''' this should be italic '''
!!! this should be bold !!!
___this should be underlined___
&&& this should be pre &&&
[[[ftp://foo.bar.com]]]
[[[ftp://foo.bar.com|not a real site]]]
[[[how to do it]]] [[[howtodoit]]] [[[How To Do It]]] \
[[[how_to_do_it]]] [[[howtodoit|How To Do It]]]
* this
* should
* be
* a
* list

# this
# should
# be
# a
# numbered list

---

http://cakephp.org
```

If you don't have an entry you can edit for this purpose, remember that editing and adding pages is essentially the same. Go to http://localhost/entries/edit/markuptest and paste in the test markup. Save and view the entry, and you'll see what you have to work with.

Figure 2. Unrendered text



This should cover all the basic positive tests -- to make sure it does what it is supposed to -- for the wiki markup. As you develop the code for rendering the markup, it will be important to cover negative testing -- making sure it doesn't do what it's not supposed to -- as well.

Rendering the markup

Time to get down to the nitty-gritty: rendering the markup. Since this was touched on in Part 1, you should have already been thinking about this particular task -- specifically, the problems that can come from dealing with nested tags.

The markup Criki will be using can be broken into three categories. The first contains all markup that opens and closes with the same set of characters, such as ===, `'''` or `!!!`. The second category contains all the markup that opens, but never explicitly closes, such as lines that start with # or * (newline implies closing the tag), as well as rendering links directly from text starting with http:// (space or newline implies closing the tag). The third category contains self-closing markup, such as ---, and the handling of newlines.

At this point in the process, don't get caught up in trying to handle everything the user might do wrong while entering wiki markup. After you get the base markup rendering working, you can enhance the code to be as forgiving or as strict as you like with the users.

For now, rendering the wiki markup into HTML should look generally like this:

- Process the entry until a valid markup is found. Open the tag and remember that it's open.
- Continue processing the entry until the next markup is found. Open the tag if it's new, close the tag if it's open.

The exception to this rule is the `&&&` markup. Since this markup wraps text in `<pre>` tags, markup rendering should be turned off while processing `&&&` markup, and any open markup tags should be closed.

Doing it the hard way

OK -- so that may be a little deceptive. What might be more accurate is saying, "Writing it out longhand."

What follows is an example of how you can render the markup as described in such a way as to pass the positive test post created above. You can reference the entries_controller.php and follow along, while the code for the view action is broken into parts.

Initializing variables

For the markup rendering, you will use two variables. The first says that if Criki is processing markup at all, use `$processMarkup = true;`.

The second will be used to keep track of what tags are currently being processed.

Listing 16. Keeping track of tags are being processed

```
$processing = array(
  "&&&" => false,
  "===" => false,
  "" => false,
  "!!!" => false,
  "___" => false,
  "[[[" => false,
  "]]]" => false,
  "*" => false,
  "#" => false,
);
```

The array keys are the markup tags themselves. The value for the keys initialized to false.

Now that you have your initial variables, you can move on to processing lines.

Line processing

To process the content lines, start by exploding the content on newlines: `$lines = explode("\n", $content);`.

Then for each of those lines, take a look at the first character. This is so you can process list entries first because you know that list entries will always start at the beginning of a line.

Listing 17. Checking the first character of exploded content

```
switch (substr($line, 0, 1)) {
  case "*" :
    if ($processMarkup) {
      if (!$processing["*"]) {
        $processing["*"] = true;
        $line = " <ul><li> " . substr($line, 1) . " </li> ";
      } else {
        $line = " <li> " . substr($line, 1) . " </li> ";
      }
    }
    break;
```

This checks to see if `$processMarkup` is true and, if so, it continues processing. If the markup encountered (in this case, unordered list) is not currently being processed, the required HTML entity is opened, and the processing flag is set. The line is then wrapped in `<li> tags`, excluding the initial character (the markup itself). This basic approach will be used for rendering the rest of the wiki markup.

The default case on the `switch` statement is used to close any open markup, reset the processing flag and append a `<br />` tag.

Once the line has been processed, you can continue on to processing the words.

Word processing

Continuing to process the text, explode the line being processed on a blank space, and process each *word* individually.

Listing 18. Processing each word individually

```
$words = explode(" ", $line);
foreach ($words as $word) {
  $word = trim($word);
```

It's important to understand that what you are dealing with is now a word in any traditional sense. It is a block of characters that had a space before and after it. That block of text could be a URL, a math formula, or a series of bad punctuation decisions. It will be referred to as a *word* for the sake of ease.

First-round processing

Each word will need to be looked at in two ways. The beginning of the word (the first three characters) will need to be examined to see if it contains opening markup, such as `===This`. Whenever opening markup is encountered, if `$processMarkup` is true and the markup is not being processed, the markup should be opened. If `$processMarkup` is true and the markup is being processed, the markup should be

closed. Listing 19 provides an example.

Listing 19. Checking the first three characters

```
case "===" :
  if ($processing["==="] && $processMarkup) {
    $processing["==="] = false;
    $word = '</h3>' . substr($word,3);
  } else {
    $processing["==="] = true;
    $word = '<h3>' . substr($word,3);
  }
  break;
```

You have seen that check against `$processMarkup` a few times now. It comes from processing the `&&&` markup.

Listing 20. Processing the &&& markup

```
case '&&&' :
  if ($processing["&&&"]) {
    $processing["&&&"] = false;
    $processMarkup = true;
    $word = '</pre>' . substr($word,3);
  } else {
    $processing["&&&"] = true;
    $processMarkup = false;
    $word = '<pre>' . substr($word,3);
  }
  break;
```

The difference in processing the `&&&` markup is that when the markup begins processing, `$processMarkup` is set to false, and it is not turned off until the `&&&` markup processing has completed.

The other two exceptions to the first round of word processing are for the `---` markup (this is self-closing, which means there is no need to track if it is processing), and the case `"htt"` (for processing URLs). The code for these cases should be self-explanatory.

The default case to the first round of processing won't make much sense until you look at the second round of word processing. But to sum it up, it says, "If I am processing a link, and I'm not the last word, push the word onto the link, not onto the content stack."

Second-round processing

Now that you have looked at the beginning of the word, you will need to look at the end of the same word, in particular for cases where a single word is involved in the markup, `===LikeThis===`. Basically, this second round will look much like the first round of processing, with the exception of the markup `]]]`, which indicates the end of a link. Walk through the code.

Listing 21. Looking at the end of the word

```
case "]]]" :
  if ($processing["]]]"] && $processMarkup) {
    if (!$processing["[[["]) {
      $processing["]]]"] .= ' ' . substr($word,0,-3);
    } else {
      $processing["]]]"] = substr($processing["]]]"],0,-3);
    }
```

The `[[[ / ]]]` is the trickiest of the bunch because it spans words but not lines and because the output may change depending on what the next word is. Back when the `[[[` tag was encountered, two processing flags were set. The `processing['[[[']` flag was set to true, while the `processing[']]]']` tag was set to the word encountered. Because of the default case in the first round of processing, each subsequent word is appended to the `processing[']]]']` flag. However, if the markup surrounds a single word -- `[[[likethis]]]` -- the stack should not be appended to.

Listing 22. Checking the link for a | character

```
    if (strpos($processing["]]]"], "|")) {
      list($alink, $atitle) = explode('|', $processing["]]]"]);
    } else {
      $alink = $processing["]]]"];
      $atitle = false;
    }
```

Check the link for a | character (indicating a link with a title). If there is a title, extract it, otherwise set the title to false.

Listing 23. Checking to see if the link is an external site

```
    if (strpos($alink, "://")) {
      $word = "<a href='" . $alink . "'>";
    } else {
      $word = "<a href='/entries/view/" .
      strtolower($alink)) ."'>";
    }
```

If the link appears to be to an external site, link to it. Otherwise, treat the link like an entry title.

Listing 24. Using the link as a title

```
    if ($atitle) {
      $word .= $atitle;
    } else {
      $word .= $alink;
    }
```

If there is no title, use the link as the title.

Listing 25. Closing and finalizing the link

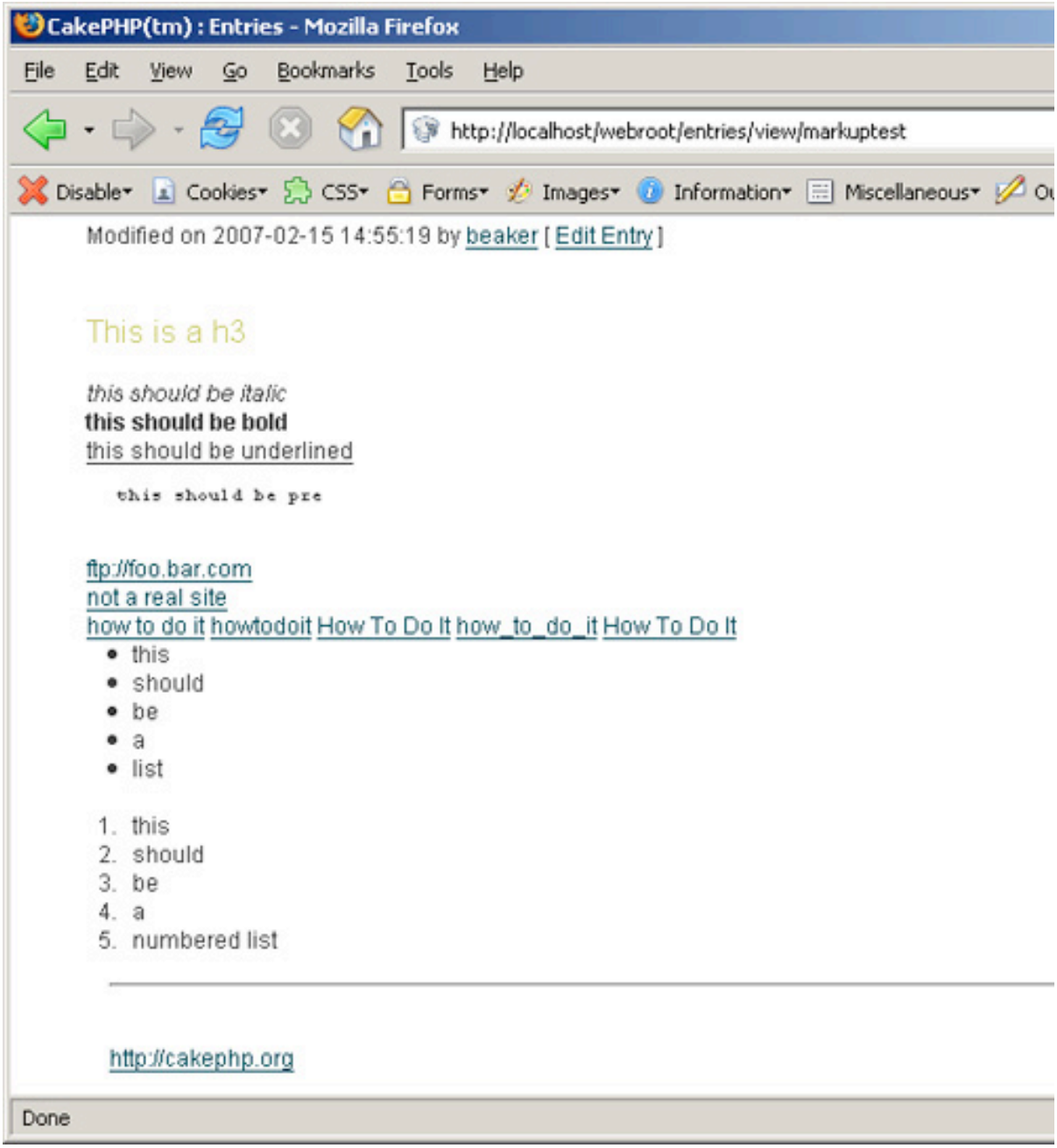
```
      $word .= "</a>";
      $processing["[[["] = false;
      $processing["]]]"] = false;
    }
    break;
```

Finally, close the link and clear out any related processing tags.

The link processing is easily the most tricky of the lot. The rest of the second-round processing will look much like the first-round processing.

Once you feel like you understand the code, copy the controller into place and view that test entry you created. It should look something like Figure 3.

Figure 3. Rendered text



Start thinking about those negative test cases because they're coming up later.

Section 8. Entry histories

Criki is really taking shape. With user registration and entries being created and read, it's beginning to turn into a real application.

Now that you can edit and view pages properly, you can look at adding some history retention for the pages in Criki. Saving the histories will be fairly simple. When a

page edit is submitted, the old page is retrieved. If the versions are different, save the old page into the histories table, then save the newly edited page.

Amending the entries controller

You will recall that the entry_revisions table was identical to the entries table, except that the title field did not have to be unique. This was intentional, so that minimal work would have to be done to get the old entry into the revisions table.

For starters, in entries_controller.php, you need to specify that more than just the Entry controller is being used by adding the following class variable: `var $uses = array('Entry','EntryRevision');`.

Note: If you define `$uses`, you must include ALL models you want to use (not just additional ones). So, the top of entries_controller.php should now look like Listing 26.

Listing 26. Top of the entries_controller.php

```
<?php
class EntriesController extends AppController {

var $name = 'Entries';
var $helpers = array('Html', 'Form' );
var $uses = array('Entry','EntryRevision');
...
```

This will allow you to access the `EntryRevisions` controller from within the `Entries` controller, making it easy for you to save the revision.

Now that this is in place, you need to add just a few lines to the controller. In the edit action, before you update the `user_id` and `ip`, add the following lines shown in Listing 27.

Listing 27. Saving new revisions

```
$entry = $this->Entry->findByTitle($this->data['Entry']['title']);
if ($entry) {
  if ($entry['Entry']['content'] == $this->data['Entry']['content']) {
    $this->Session->setFlash('No changes were made.');
    $this->redirect('/entries/view/'.$this->data['Entry']['title']);
    exit;
  } else {
    $revision['EntryRevision'] = $entry['Entry'];
    unset($revision['EntryRevision']['id']);
    $this->EntryRevision->save($revision);
  }
  $this->data['Entry']['revision'] = $entry['Entry']['revision']+1;
}
```

Walking through the code in English, it says, "Get the existing entry. If the content has not changed, don't do anything. If it has, dump the ID and save the data with the `EntryRevision` controller, and increment the revision number."

That's all there is. Try editing an entry and visiting http://localhost/entry_revisions to see a list of revisions. You should see the old version of your entry in the revisions list.

Updating the EntryRevision model

As with the `Entry` model, you will need to specify that the `EntryRevision` model has a relationship to users so that user data associated with an entry can be retrieved with the entry. This association will look exactly as it did for the `Entry` model.

Listing 28. Specifying a relationship with a user to an EntryRevision model

```
var $belongsTo = array('User' => array (
    'className' => 'User',
    'conditions' => ,
    'order' => ,
    'foreignKey' => 'user_id'
  )
);
```

Now that your model is taken care of, you need to update your controller.

Updating the EntryRevisions controller

The `EntryRevision` controller will only need to serve two purposes. The first is to display a list of revisions for a particular entry. This will be done by modifying the index action. The second purpose is to display an individual revision. This will be the view action. The add, edit, and delete actions should be deleted.

Displaying a revision list

Take a look at the index action.

Listing 29. Index action

```
function index() {
  $this->EntryRevision->recursive = 0;
  $this->set('entryRevisions', $this->EntryRevision->findAll());
}
```

With a slight modification, this can be used to get all revisions for a specific title.

Listing 30. Getting all the revisions for a specific title

```
function index($title = null) {
  $this->EntryRevision->recursive = 0;
  if ($title) {
    $revisions = $this->EntryRevision->findAllByTitle($title);
    if ($revisions) {
      $this->set('entryRevisions', $revisions);
```

```
    } else {
      $this->Session->setFlash('No Revision History For This Article');
      $this->redirect('/entries/view/' . $title);
    }
  } else {
    $this->set('entryRevisions', $this->EntryRevision->findAll());
  }
}
```

Now, by visiting http://localhost/entry_revisions/index/TITLE, you can view the revisions for the individual entries by title -- once you have updated the views.

Updating the EntryRevisions views

Rather than re-editing the view and index views for `EntryRevisions`, you can simply copy those from app/views/entries/ to /app/views/entry_revisions -- that's copy, not move. These views will serve well as a template for the EntryRevision views. You should also delete the add and delete views from app/views/entries and app/views/entry_revisions because they won't be necessary.

You need to change the following things about the view and index views:

- Remove the edit links -- no one may edit a revision.
- Change instances of `['Entry']` to `['EntryRevision']`.
- Change instances of `entries'` to `'entryRevisions`.
- Change the view links for revisions to pull the view by ID -- remember, titles are not unique in the revision history.
- Change the view of the view to display the revision number.

You get the basic idea. Consult the source code if the changes are unclear.

Your new views are in place. Spend some time creating revisions and see how it all flows.

Section 9. File uploads

Allowing files to be uploaded to a Web application is a task that needs to be handled with great care. Allowing anyone to upload any kind of file, and having the file be accessible via a direct Web request, is an incredibly dangerous proposition, regardless of the precautions you take. At the very least, it can open you up to malicious code execution on clients visiting your application. At the very worst, if can lead to arbitrary code execution on your server.

The approach described in this tutorial is not perfect. You are highly encouraged to

consider the problem between now and Part 3.

What's the big deal?

Consider a basic approach to the problem: You want the user to be able to upload files into Criki in such a way that anyone can use them. Without thinking, one might simply make a directory in app/webroot, such as uploads, and dump files submitted for upload in this directory, which would make them accessible via http://localhost/uploads. This is a fast and easy solution. That should be the first warning sign that it can go horribly wrong.

For one thing, consider that user uploads a file containing malicious JavaScript of some new and untold variety -- perhaps it steals session or cookie data or worse: changes the user's coffee to decaf. Regardless, the script now resides on your Web server.

Now suppose the same user uploads an HTML file that invokes the JavaScript and makes it do those horrible things it does. Now any user who views the page will reveal his cookies, and may find himself sluggish and unable to focus in the early afternoon for unknown reasons. This is not the behavior you want from Criki.

Getting into the more serious, consider the repercussions of a user uploading a file called info.php containing valid code to execute `phpinfo()`: `<?php phpinfo(); ?>`.

It seems like an awfully simple thing -- not inherently malicious. But now anyone can visit http://localhost/uploads/info.php and execute the script. Now imagine what a *genuinely* malicious user could write and upload and execute, directly on your server, and the damage it can do.

What should you do instead?

That's an excellent question. You should think about it between now and Part 3, where a suggestion solution will be provided. Here are the parameters:

1. Files a user uploads should be accessible by other users.
2. At some point, you may want to allow users to add images to entries via new wiki markup.
3. Under no circumstances do you want a remote user to be able to somehow execute a file uploaded on your server.
4. You want to involve a mechanism for controlling what file types can be uploaded.
5. Any solution you come up with should balance the need for security

against performance.

There are lots of ways to approach this. None of them are without flaw. See what you can come up with.

Filling in the gaps

You've gotten a lot done. The basic structure code of your wiki is in place. But there's a lot that can be done to improve it. Specifically, try to address the following:

1. If an entry is edited by an user who is not logged in, an offset error occurs on the view page. This is because the username is unknown. Fix this so that, if the username is not known, the IP address is displayed instead.
2. The code to translate the wiki markup could use some work:
   - Try entering raw HTML into a content box. What happens? How would you fix it? What other negative test cases can you identify?
   - Experiment with nested markup and see if you can get it to break.
   - Clean up the wiki markup translation code. It could easily be reduced in a number of ways.
   - The view action of the `EntryRevisions` controller should look just like the view action for the entry controller. But rather than copy all of the ugly wiki markup translation code over, wait until you have gotten it more compact, then copy it over.
3. Don't forget to ponder The Problem of File Uploads.

This should give you more than enough to keep you coding until Part 3, where you work Users and Permissions into Criki. Until then, happy coding.

Section 10. Summary

You've gotten quite a lot done. The core code for Criki is up and running, including user registration, entry storage, and markup rendering. You've also started to take a look at file uploads and the problems they present. In Part 3, you address the file uploads problem. Once that's done, you define user types, and write code to define and apply permissions to entries and uploaded files.

Downloads

Description	Name	Size	Download method
Part 2 source code	os-php-wiki2.source.zip	9KB	HTTP

Information about download methods

Resources

Learn

- Read Part 1 and Part 3 of this "Create an interactive production wiki using PHP" series.
- Check out the Wikipedia entry for wiki.
- Check out WikiWikiWeb for a good discussion about wikis.
- Visit the official home of CakePHP.
- Check out the "Cook up Web sites fast with CakePHP" tutorial series for a good place to get started.
- 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.
- Find out more about how PHP handles sessions.
- Check out the official PHP documentation.
- Read the five-part "Mastering Ajax" series on developerWorks for a comprehensive overview of Ajax.
- Check out the "Considering Ajax" series to learn what developers need to know before using Ajax techniques when creating a Web site.
- CakePHP Data Validation uses PHP Perl-compatible regular expressions.
- See a tutorial on "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 the Model-View-Controller on Wikipedia.
- Here is more useful background on the Model-View-Controller.
- Here's a whole list of different types of software design patterns.
- Read more about Design Patterns.
- PHP.net is the 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.

- 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.
- Visit Safari Books Online for a wealth of resources for open source technologies.

Get products and technologies

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

Discuss

- Participate in developerWorks blogs and get involved in the developerWorks community.
- Participate in the developerWorks PHP Developer Forum.

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.

Create an interactive production wiki using PHP, Part 3: Users and permissions

Taking control of Criki

Skill Level: Intermediate

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

20 Mar 2007

This "Create an interactive production wiki using PHP" tutorial series creates a wiki from scratch using PHP, with value-added features useful for tracking production. Wikis are widely used as tools to help speed development, increase productivity and educate others. Each part of the series develops integral parts of the wiki until it is complete and ready for prime time, with features including file uploading, a calendaring "milestone" system, and an open blog. The wiki will also contain projects whose permissions are customizable to certain users. In Part 2, you got the basic wiki working. Now it's time to add some control over who can do what when accessing Criki.

Section 1. Before you start

This "Create an interactive production wiki using PHP" series is designed for PHP application developers who want to to take a run at making their own custom wikis. You'll define everything about the application, from the database all the way up to the wiki markup you want to use. In the final product, you will be able to configure much of the application at a granular level, from who can edit pages to how open the blog really is.

At the end of this tutorial, Part 2 of a five-part series, you will have the basics of your wiki up and running, including user registration, page creation and editing, history tracking, and file uploads. It sounds like a lot, but if you've completed Part 1, you're well over halfway there.

About this series

Part 1 of this series draws the big picture. You determine how you want the application to look, flow, work, and behave. You'll design the database and rough-out some scaffolding. Part 2 focuses on the primary wiki development, including defining the markup, tracking changes, and file uploads. Here in Part 3, you define some users and groups, as well as a way to control access to certain aspects of individual wiki pages and uploaded files. Part 4 deals with a Calendaring and Milestones feature to track tasks, to-dos, and progress against set goals. And in Part 5, you put together an open blog to allow discussion of production topics and concerns.

About this tutorial

This tutorial, Part 3 of a five-part series, focuses on users and permissions primarily. Criki (your new wiki engine) has already taken a lot of shape as it allows you to edit, view, and track the history of various entries. Once you get users and permissions sorted out, you have a good foundation on which you can start to add those production related features in the next tutorials.

Covered topics include:

- File uploads
- User types
- User permissions

Prerequisites

It is assumed that you have some experience working with PHP and MySQL. We won't be doing a lot of deep database tuning, so as long as you know the basic ins and outs, you should be fine. You may find it helpful to download and install phpMyAdmin, a browser-based administration console for your MySQL database.

System requirements

Before you begin, you need to have an environment in which you can work. The general requirements are reasonably minimal:

- An HTTP server that supports sessions (and preferably mod_rewrite). This tutorial was written using Apache V1.3 with mod_rewrite enabled.
- PHP V4.3.2 or later (including PHP V5). This was written using PHP V5.0.4

- Any version of MySQL from the last few years will do. This was written using MySQL V4.1.15.

You'll also need a database and database user ready for your application to use. The tutorial will provide syntax for creating any necessary tables in MySQL.

Additionally, to save time, we will be developing Criki using a PHP framework called CakePHP. Download CakePHP by visiting CakeForge.org and downloading the latest stable version. This tutorial was written using V1.1.13. For information about installing and configuring CakePHP, check out the tutorial series titled "Cook up Web sites fast with CakePHP" (see Resources).

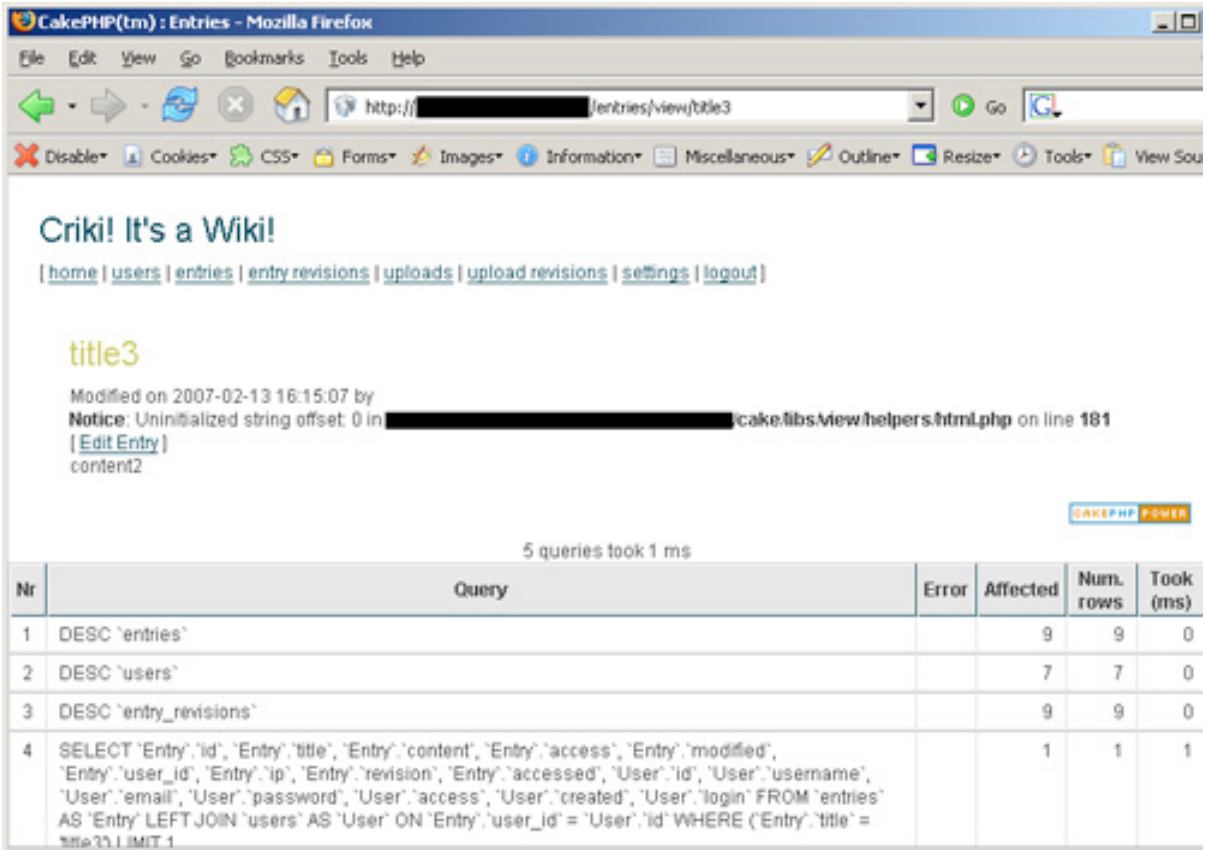
Section 2. Criki so far

At the end of Part 2, you were given a few things to work on: fixing the error that happens when an article is edited by a user who is not logged in, further enhancing the wiki markup translation code, and you were to ponder the problems of file uploads. How did you do?

Logged-out user editing

When a logged-out user edited an entry, if the entry was viewed, you would have seen the following error:

Figure 1. Error



This error originates in the model, though it's not the model doing anything wrong. You will recall that you established a relationship between the entry model and the user model, such that when an entry was retrieved, associated user information was also retrieved. In the case of a logged-out user, the `user_id` for the entry will be empty because there is no user date to retrieve.

The error doesn't bubble up to the surface until you try to output the user data in the view in line 4 of app/view/entries/view.thtml.

```
by <?php echo $html->link($entry['User']['username'], $entry['Entry']['user_id'])?>
```

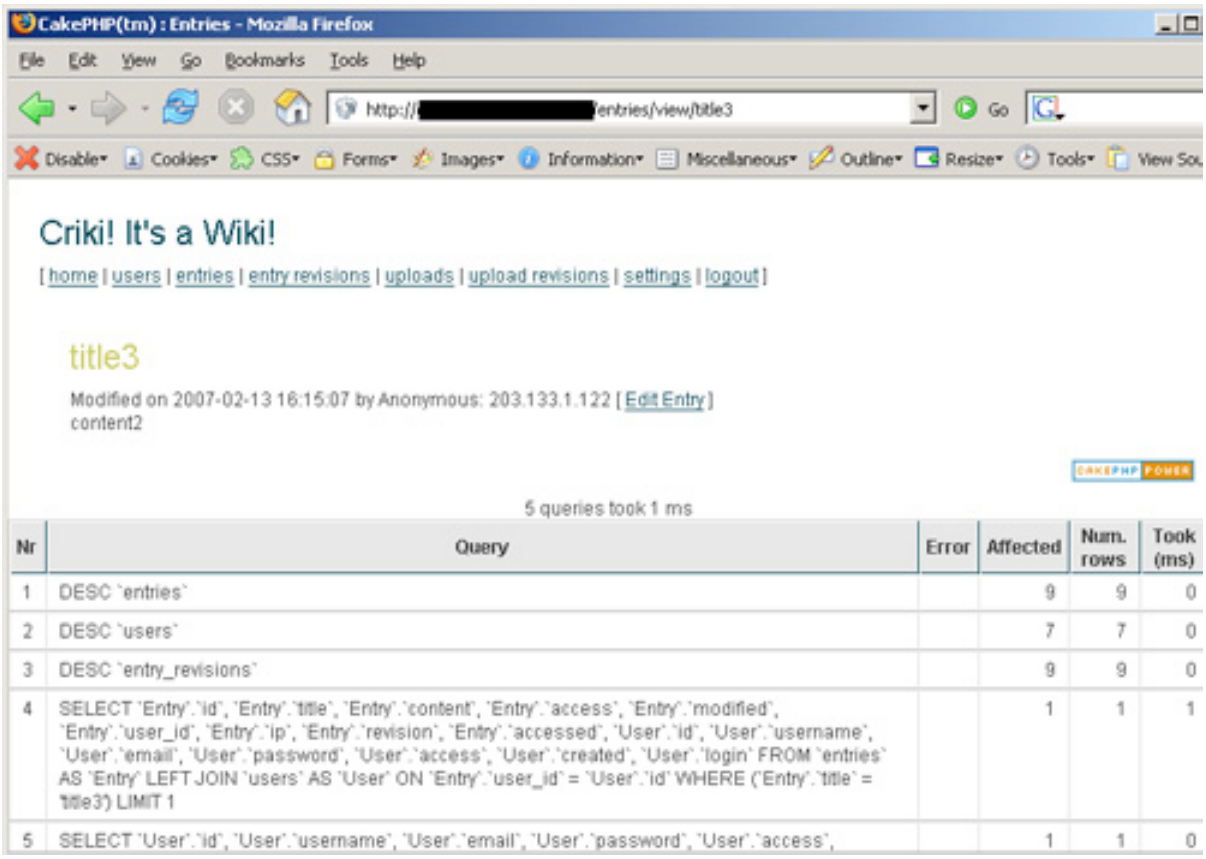
The best way to address this is to verify that the username is set and display the IP address if it's not.

Listing 1. Fixed user link in entries view

```
by
<?php
if (isset($entry['User']['username'])) {
        echo $html->link($entry['User']['username'], $entry['Entry']['user_id']);
} else {
        echo 'Anonymous: ' . $entry['Entry']['ip'];
}
?>
```

Now when you view the entry that was edited by a logged-out user, it should look more like Figure 2.

Figure 2. Fixed error



Nr	Query	Error	Affected	Num. rows	Took (ms)
1	DESC `entries`		9	9	0
2	DESC `users`		7	7	0
3	DESC `entry_revisions`		9	9	0
4	SELECT `Entry`.`id`, `Entry`.`title`, `Entry`.`content`, `Entry`.`access`, `Entry`.`modified`, `Entry`.`user_id`, `Entry`.`ip`, `Entry`.`revision`, `Entry`.`accessed`, `User`.`id`, `User`.`username`, `User`.`email`, `User`.`password`, `User`.`access`, `User`.`created`, `User`.`login` FROM `entries` AS `Entry` LEFT JOIN `users` AS `User` ON (`Entry`.`user_id` = `User`.`id`) WHERE (`Entry`.`title` = 'title3') LIMIT 1		1	1	1
5	SELECT `User`.`id`, `User`.`username`, `User`.`email`, `User`.`password`, `User`.`access`,		1	1	0

That was fairly simple, with what was hopefully a simple solution for you.

Wiki markup revisions

You had four tasks to resolve regarding the wiki markup:

1. Identify and fix negative test cases, such as rendering raw HTML.
2. Resolve nested markup problems.
3. Clean up the Wiki markup translation code.
4. Set up the View action of the EntryRevisions controller.

Each task should have presented a unique set of problems, but they can be covered in a couple pieces.

Negative test cases and nested markup

You tested to make sure that Criki was doing what you wanted it to. But it's far more important to test that it doesn't do what it's not supposed to do. Consider the following.

Listing 2. Negative markup test

```
===I don't want to fight
*unless I have to===

===I don't want to fight===
*unless I have to
*And I don't want to
===If you have to fight===
*Fight Dirty
*Win

=== if you ___have___ to '''fight''' ===

<h4>this is a h4 tag</h4>

<script>alert('boo!')</script>

[[[thingy" onclick="alert('boo!')]]]
```

You should recognize those first two lines from the initial markup discussion. Paste this entry into your version of Criki. What happens?

The most important thing to fix in the code shown in Listing 2 is the HTML rendering. It leads to embedding malicious JavaScript, cross-site scripting, and a tectonic plate shift. You have two choices when it comes to dealing with the HTML: strip it out completely or convert it to HTML entities.

To strip it out completely, you can use PHP's `strip_tags` function. It's a draconian approach, but it gets the job done. However, if a user wants to paste in an HTML code sample, he would be out of luck.

If you want to convert the HTML special characters (using PHP's `htmlspecialchars` function), you run into a different problem. You can't do the conversion before saving the data to the database because `&` and `'` characters would be converted (thus breaking the wiki markup). You could do the conversion on display, but if the conversion fails for some reason, the embedded HTML would be rendered normally.

Probably the safest way to deal with this problem is to use a `preg_replace` to replace problem characters. The idea here is to change as little of the original text as possible, while still keeping Criki secure. So, in the edit action, before you save the entry, you would do something like that shown below.

```
$patterns = array ('/</', '/>/');
$replacements = array ('<', '>');
$this->data['Entry'] = preg_replace($patterns, $replacements, $this->data['Entry']);
```

You need to edit the problem entry before the HTML in the entry ceases rendering, and the revisions will still show the HTML. If you feel adventurous, you can work on stripping out HTML after wiki markup rendering, as well. But be forewarned: It can get tricky. Speaking of wiki markup rendering, that also needed some cleaning up.

Clean up the markup; add an EntryRevisions view action

Were it not for the <pre> markup, most of the wiki markup rendering could be done with a couple regular expressions. However, the code provided in Part 2 to render the wiki markup can still be simplified significantly. Without reproducing large blocks of the code here, you will remember that the wiki markup rendering code consisted of some switch statements that looked something like Listing 3.

Listing 3. Old markup switch case

```
case "!!!" :
  if ($processing["!!!"] && $processMarkup) {
    $processing["!!!"] = false;
    $word = substr($word,0,-3) . '</b>';
  } else {
    $processing["!!!"] = true;
    $word = substr($word,0,-3) . '<b>';
  }
  break;
```

Most of the cases looked exactly the same, which meant that the code could be streamlined by building an array to hold some markup information, calling out deviant cases specifically, and handling normal cases with code similar to the following.

Listing 4. New markup processing

```
if ($markup[$key]['processing']) {
  $markup[$key]['processing'] = false;
  $word = $markup[$key]['close'] .  substr($word,3);
} else {
  $markup[$key]['processing'] = true;
  $word = $markup[$key]['open'] . substr($word,3);
}
```

In the source code for this tutorial, the entries_controller.php contains the old view action (renamed to review), as well as a new view action. Compare the two. The review action is more immediately readable, but is less efficient overall and requires more work if markup is added. The view action is less immediately readable, but more efficient overall and can handle most new markup by simply adding the markup to the markup array. Do you see room for improvement? You should, there's plenty!

Once you have the action working the way you want, remember to copy it to the EntryRevisions controller. There wasn't a view action for that controller at the end of Part 2.

That about covers the "Filling in the Gaps" section from Part 2. Now you need to finish addressing the problem of file uploads.

Section 3. File uploads, continued

To review, at the end of Part 2, you were given some parameters to consider when solving how to upload files securely:

1. Files a user uploads should be accessible by other users.
2. At some point, you may want to allow users to add images to their entries via new wiki markup.
3. Under no circumstances do you want a remote user to be able to somehow execute a file uploaded on your server.
4. You want to involve a mechanism for controlling what file types can be uploaded.
5. Any solution you come up with should balance the need for security against performance.

Examining these points more closely, you should be able to rank them in terms of importance. Least important would be the ability to add images via markup. When designing your solutions, you should consider the "maybe" requirements, but not write your code to them.

Most important would be -- in order of importance -- security, sharing, and access control. Balancing the security against performance shouldn't be ignored, but it's more of an overall guiding principle than a requirement.

Why not store them in a database?

Given the purpose of Criki (a wiki to track production tasks), users will probably be more likely to upload files like word documents, PDFs, flat files, etc. In other words, large files. Writing and retrieving large files from the database is not a task well suited for the configuration under which Criki is likely to be installed (single server, no database tuning).

Storing the files on the file system

Rather than trying to store large files in the database, you'll store them on the file system. The basic approach:

- The files will be stored in a directory not directly Web-accessible.
- The files will be served to the user for download by way of an action in the

files controller.

- File information and versioning will be stored in the database.

Following this approach, you should be able to meet the base requirements: The files cannot be accessed directly using a Web browser or executed directly on the system; other users can get access to the files; and you have the ability to check access control in the action that serves the files.

Now that you know what you're going to do with uploaded files, you can start putting it all together. You'll need a directory to store the files, a view to present the upload form, and an action in the uploads controller to handle the incoming file.

Creating the directory

Uploaded files are initially stored in the server's default tmp directory unless you change this in the php.ini file. If you want to keep them around, you'll want to move them from the tmp directory to someplace more stable.

Start by making a directory to hold the uploaded files. This directory should be outside the web root directory you used for CakePHP. For example, if your web root is /var/htdocs, you would want to create something like /var/uploads as a directory to hold uploaded files. This directory will need to be readable and writable by the same user your Apache server uses.

Additionally, you want to define a constant somewhere to hold the location of this directory. The following line should be added to app/config/bootstrap.php: `define ('UPLOADS_DIRECTORY', '/var/uploads/');`, where /var/uploads/ is the directory you created. Note the trailing slash.

Creating the view

Now that you have a place to keep the files, you need a form to be used to upload. Create the basic view app/views/uploads/files.thtml shown below.

Listing 5. Upload file view

```
 <h2>Upload File</h2>
 <form enctype="multipart/form-data" action="<?php echo $html->url('/uploads/file'); ?>"
method="post">
 <div class="optional">
        <?php echo $form->labelTag('Upload/file', 'Filename');?>
        <?php echo $html->file('Upload/file');?>
        <?php echo $html->tagErrorMsg('Upload/file', 'Please enter a file.');?>
 </div>
 <div class="submit">
        <?php echo $html->submit('Upload');?>
 </div>
 </form>
 <ul class="actions">
 <li><?php echo $html->link('List Uploads', '/uploads/index')?></li>
 </ul>
```

Listing 5 is short, straightforward, and to the point. Most of the rest of the upload information you don't need from the user; it will either be in session or derived from the file itself.

Creating the file action

You have a place to keep files and a view to allow the user to upload files. Now you need to do something with the uploaded files.

For starters, just create the following file action in app/controllers/uploads_controller.php.

Listing 6. Debug uploads file action

```
function file() {
  if(empty($this->data)) {
    $this->render();
  } else {
    $this->cleanUpFields();
    debug($this->data);
  }
}
```

This is a simple action that will take the uploaded file and output to the screen the contents `$this->data`.

Save the controller and try uploading a file. You should get output that looks roughly like the following (format notwithstanding).

Listing 7. Debug output of $this->data

```
Array
(
  [Upload] => Array
  (
    [file] => Array
    (
      [name] => testupload.txt
      [type] => text/plain
      [tmp_name] => /tmp/php4LVxoe
      [error] => 0
      [size] => 19
    )
  )
)
```

Does that file information look familiar? It's the same information you would access through the `$_FILES` variable: original filename, MIME type according to the browser, temporary filename, errors if any, and size in bytes. CakePHP pulls the information into `$this->data` to make your life that much easier (you're always going to the same place for your data).

The basic steps (for now) you'll want to take are: verify the file, copy the file to your storage directory, and create an entry in the uploads database for the file. (Later,

you'll cover keeping track of revisions). The resulting action for accomplishing these steps looks like Listing 8.

Listing 8. Upload file action

```
function file() {
  if(empty($this->data)) {
    $this->render();
  } else {
    if ($this->data['Upload']['file']['error'] == 0) {
      if (move_uploaded_file($this->data['Upload']
          ['file']['tmp_name'], UPLOADS_DIRECTORY .
$this->data['Upload']['file']['name'])) {
        $this->Session->setFlash('The file has been saved.');
        $this->data['Upload']['filename'] = $this->data['Upload']
['file']['name'];
        $this->data['Upload']['location'] = UPLOADS_DIRECTORY;
        $user_id = 0;
        if ($this->Session->check('User')) {
          $user = $this->Session->read('User');
          $user_id = $user['id'];
        }
        $this->data['Upload']['user_id'] = $user_id;
        $this->data['Upload']['ip'] = $_SERVER['REMOTE_ADDR'];
        if($this->Upload->save($this->data)) {
          $this->redirect('/uploads/view/'.$this->Upload->id);
        } else {
          $this->Session->setFlash('An error occurred saving the
 upload information.');
        }
      } else {
        $this->Session->setFlash('The file successfully uploaded
 but an error occurred.');
      }
    } else {
      $this->Session->setFlash('There was an error uploading the file.');
    }
  }
}
```

Take the new action for a spin. You should be able to upload a file from http://localhost/uploads/file and be directed to the uploads view. You should also be able to see the uploaded file in the uploads directory you created earlier.

You did it. You're able to upload files. Now, you need to be able to get them back.

Retrieving files

Sending the files back to the user is actually the easy part. You know where the files are, and you have saved that information into the uploads table. All you need now is an action to get the file and send it to the browser for download. This will require a new action: `fetch`.

It will look something like Listing 9.

Listing 9. Upload fetch action

```
function fetch($id = null) {
  if(!$id) {
```

```
    $this-<Session-<setFlash('Cannot file the file indicated.');
    $this-<redirect('/uploads/index');
  }
  $upload = $this-<Upload-<read(null, $id);
  if ($upload) {
    header('Content-Type: application/octet-stream');
    header('Content-Disposition: attachment;
        filename="' . $upload['Upload']['filename'] . ' "');
    header('Content-Length: ' filesize($upload['Upload']
        ['location'].$upload['Upload']['filename']));
    readfile($upload['Upload']['location'].$upload['Upload']['filename']);
    exit;
  } else {
    $this-<Session-<setFlash('Cannot file the file indicated.');
    $this-<redirect('/uploads/index');
  }
}
```

The most important parts of this action are the header calls and the readfile. The header calls say, in order, "Browser, prepare to receive a stream for download. Here is the filename. Here is the size." The readfile opens the file to be served and outputs it to the stream.

To test it, you need to add a link to the `fetch` action. You can add this to app/views/uploads/index.thtml so the actions available are as follows.

```
<?php echo $html->link('View','/uploads/view/' . $upload['Upload']['id'])?>
<?php echo $html->link('Fetch','/uploads/fetch/' . $upload['Upload']['id'])?>
```

Go ahead and try it out on a file you uploaded. Go to http://localhost/uploads/ and click **Fetch**. You should be prompted to download the file. Once you have downloaded the file, open it and verify that the contents are valid.

File revisions

You're able to upload files and retrieve them from Criki. As with the entries, it will be helpful to keep some history for uploaded files. Keeping track of the file history will look much like tracking entry history. You will start by querying the uploads table to see if a version of the file already exists. If a file with the same name already exists, you'll save that the information for that file into the upload_revisions table before you save the new upload data (with an updated revision number). However, there is additional work to be done: You need to keep the previous version of the file on the file system and update the filename (and potentially the file location) in the upload_revisions table.

The scheme for keeping prior versions of a file will be very straightforward. When a file is backed up, the file will be renamed to FILENAME.REVISION. So, if the file is test.txt and the revision is 3, the file will be renamed "test.txt.3" when the revision is backed up. The file name will be corrected when the revision is fetched.

To accomplish this, you need to update the uploads controller, UploadRevision Model, and the UploadRevisions controller.

Updating the uploads controller

As you did when setting up the entry revisions tracking, you need to specify that an additional model is being used by the controller. This means declaring the `$uses` class variable in uploads_controller.php: `var $uses = array('Upload','UploadRevision');`.

Now the uploads controller has access to the UploadRevision model for the purpose of saving revisions. But you still need to make some changes to the uploads controller. Specifically, the file action needs to be changed to save revision data and file backups, as discussed. After verifying that the file was uploaded successfully, the code to save the revision should look like Listing 10.

Listing 10. Additional uploads file action code

```
if ($upload) {
  $revision['UploadRevision'] = $upload['Upload'];
  unset($revision['UploadRevision']['id']);
  $revision['UploadRevision']['location'] = UPLOADS_DIRECTORY;
  rename($upload['Upload']['location'].$upload['Upload']['filename'], UPLOADS_DIRECTORY .
$upload['Upload']['filename'] . '.' . $upload['Upload']['revision']);
  $this->UploadRevision->save($revision);
  $this->data['Upload']['revision'] = $upload['Upload']['revision']+1;
  $this->data['Upload']['id'] = $upload['Upload']['id'];
}
```

Of particular interest is setting the upload revision location to the value of `UPLOADS_DIRECTORY`. This is done so that if the value of the constant is ever changed, files will still be copied to and fetched from the correct location.

Updating the UploadRevision model

So that you can display and access user data associated with an upload, you need to make the same kind of model association in the upload and UploadRevision models. This association will look identical to the association you did for the entry and EntryRevisions models.

Listing 11. UploadRevision model association

```
 var $belongsTo = array('User' => array (
     'className' => 'User',
     'conditions' => '',
     'order' => '',
     'foreignKey' => 'user_id'
   )
 );
```

Make sure to add this association to app/models/upload.php and app/models/upload_revision.php so both can access the necessary models.

Updating the UploadRevision controller

The UploadRevision controller will serve purposes similar to the EntryRevisions controller: Displaying a list of revisions for a file and fetching individual revisions. These tasks will be accomplished by the `index` and `fetch` actions, respectively. All other actions should be deleted.

Displaying a revision list

Similar to what you did with the `index` action in the EntryRevisions controller, you will make slight modifications to the `index` action to get all revisions for a specific filename.

Listing 12. Modified UploadRevisions index action

```
function index($filename = null) {
  $this->UploadRevision->recursive = 0;
  if ($filename) {
    $revisions = $this->UploadRevision->findAllByFilename($filename);
    if ($revisions) {
      $this->set('uploadRevisions', $revisions);
    } else {
      $this->Session->setFlash('No Revision History For This File');
      $this->redirect('/uploads/view/' . $filename);
    }
  } else {
    $this->set('uploadRevisions', $this->UploadRevision->findAll());
  }
}
```

Now, by visiting http://localhost/upload_revisions/index/FILENAME, you can view the revisions for the individual uploaded files by title -- once you have updated the views.

Fetching a previous revision

The `fetch` action for the UploadRevisions controller will look much like the `fetch` action for the uploads controller, except that the file being retrieved for the fetch has had the revision number appended to the end of the filename.

Listing 13. UploadRevisions fetch action

```
function fetch($id = null) {
  if(!$id) {
    $this->Session->setFlash('Cannot file the file indicated.');
    $this->redirect('/upload_revisions/index');
  }
  $upload = $this->UploadRevision->read(null, $id);
  if ($upload) {
    header('Content-Type: application/octet-stream');
    header('Content-Disposition: attachment;
        filename="' . $upload['UploadRevision']['filename'] . '
"');
    header('Content-Length: ' .
filesize($upload['UploadRevision']['location'].$upload['UploadRevision']['filename']));
    readfile($upload['UploadRevision']
        ['location'].$upload['UploadRevision']['filename'] .
'.'.$upload['UploadRevision']['revision']);
    exit;
```

```
    } else {
      $this->Session->setFlash('Cannot file the file indicated.');
      $this->redirect('/upload_revisions/index');
    }
  }
```

To see it in action, make a quick modification to the app/views/upload_revisions/index.thtml file, removing unnecessary actions and adding a link to the fetch.

```
<?php echo $html->link('Fetch','/upload_revisions/fetch/' .
$uploadRevision['UploadRevision']['id'])?>
```

You should now be able to go to http://localhost/upload_revisions and fetch a previous revision of a file. Try it out.

You've gotten a lot done so far. You can upload files, keep track of revisions, and get files back from Criki. You are to the point now that user types and permissions begin to become important.

Section 4. User types

As you will recall from Part 1, three types of basic users were identified: contributors, editors, and administrators. Thus far, nothing has been done to distinguish one from the other, save that there is a general sense that a contributor is a base user, an editor is a kind of super-contributor with some rights over other contributors, and an administrator is a kind of super-editor, with power over editors and contributors.

By defining the user groups in this sort of hierarchy, you have simplified the task of defining and assigning user permissions. You'll learn more about that later. For now, you can focus on the task of user promotion/demotion.

Note: CakePHP comes with an excellent access control system using Access Control Lists, Access Request Objects, and Access Control Objects. The system would be well suited for solving this particular problem, since it is specific to CakePHP. A more general approach has been used here to allow you to apply the same principles directly to non-CakePHP projects.

How users will be promoted

When you created the users table, you included a field called access. This field was declared as `int(1)` meaning that it would hold integer values, one-digit maximum. You might correctly assume from this that user types will, therefore, be represented with numbers.

So that you have room to grow and expand the different types of users, you will use the following system:

- Access of 0 will represent a contributor
- Access of 4 will represent an editor
- Access of 8 will represent an administrator

For now, user types and user permissions will follow these rules:

- No user may demote another user of higher access (Editor cannot demote Administrator, but can demote editor).
- No user may promote another user to an access above his own (Editor can promote Contributor to Editor, but not Editor to Administrator).
- No user may demote content of a higher access then his own (Editor cannot demote Administrator-level content).
- No user may promote content to an access higher than his own (Editor cannot promote content past Editor-level access).

That may sound a little confusing, but remember: It's all just numbers. It will make more sense as you write the code. As for the actual mechanics of user promotion/demotion, it can all be done with a link.

Creating the promote and demote user actions

User promotion and demotion will require a pair of actions in the users controller: promote and demote. They will look very similar. A user ID will be queried, the access levels will be verified, and if the action is permitted, it will proceed. Both actions will require that the user be logged in. The promote action looks like Listing 14.

Listing 14. Users promote action

```
function promote($id = null) {
  if ($this->Session->check('User')) {
    $user = $this->Session->read('User');
    if(!$id) {
      $this->Session->setFlash('Invalid id for User');
      $this->redirect('/users/index');
      exit;
    }
    $user = $this->User->read(null, $user['id']);
    if ($user['User']['access'] == 0) {
      $this->Session->setFlash('Contributors cannot promote.');
      $this->redirect('/users/view/'.$id);
      exit;
    }
    if ($user['User']['id'] == $id) {
      $this->Session->setFlash('You cannot promote yourself.');
      $this->redirect('/users/view/'.$id);
      exit;
    }
```

```
    $subject = $this->User->read(null, $id);
    if ($user['User']['access'] > $subject['User']['access']) {
      $subject['User']['access'] += 4;
      $this->User->save($subject);
      $this->Session->setFlash('The User has been promoted');
      $this->redirect('/users/view/'.$id);
    } else {
      $this->Session->setFlash('You cannot promote a User of equal or higher clearance');
      $this->redirect('/users/view/'.$id);
    }
  } else {
    $this->Session->setFlash('You must be logged in to perform this action');
    $this->redirect('/users/login');
  }
}
```

Pay attention to the kinds of checks you are doing here. Is the user logged in? Was an ID passed? Is the user a contributor? (Since they can't promote anyone, you can throw out the request immediately.) Is the user trying to promote himself? (This would fail anyway when the access levels are checked, but this way, the user knows you are on to him.) All of these checks take place before the subject of the promotion is even looked at. As for the actual promotion method, since the access levels are evenly spaced, you can simple add a fixed number to the subject's access, resulting in promotion. It's a little simplistic, but it's sufficient for demonstration purposes.

The demote action is going to look very similar.

Listing 15. Users demote action

```
function demote($id = null) {
  if ($this->Session->check('User')) {
    $user = $this->Session->read('User');
    if(!$id) {
      $this->Session->setFlash('Invalid id for User');
      $this->redirect('/users/index');
      exit;
    }
    $user = $this->User->read(null, $user['id']);
    if ($user['User']['access'] == 0) {
      $this->Session->setFlash('Contributors cannot demote.');
      $this->redirect('/users/view/'.$id);
      exit;
    }
    $subject = $this->User->read(null, $id);
    if ($subject['User']['access'] == 0) {
      $this->Session->setFlash('Contributors cannot be demoted.');
      $this->redirect('/users/view/'.$id);
      exit;
    }
    if ($user['User']['access'] >= $subject['User']['access']) {
      $subject['User']['access'] -= 4;
      $this->User->save($subject);
      $this->Session->setFlash('The User has been demoted');
      $this->redirect('/users/view/'.$id);
    } else {
      $this->Session->setFlash('You cannot demote a User of higher clearance');
      $this->redirect('/users/view/'.$id);
    }
  } else {
    $this->Session->setFlash('You must be logged in to perform this action');
    $this->redirect('/users/login');
  }
}
```

You are performing the same basic kinds of checks before proceeding with demotion, with one exception: If the user wants to demote himself, go ahead and let him. Additionally, you need to check to make sure the subject is not already a contributor -- there is nothing lower to demote him to.

Now that you have the actions in place, a couple quick modifications to the user views will make user promotion/demotion a cinch.

Showing the right links

You want to modify the index view and the "view" view, so that promotion and demotion links are only shown when the actions can be performed by the user. For this tutorial, the changes will only be made to the index view. You will need to apply the same kind of changes to the "view" view.

Amending the index view

The users index view still contains some links that should be removed -- namely, the links to the edit, delete, and add actions. You should pull those links out while you are here, but mainly, you need to add code to pull the logged-in user's data and conditionally display the promote and demote links. When you are done, the actions table cell should look like Listing 16.

Listing 16. Users index view update

```
<?php $user_data = $session->read('User');
      if ($user_data['access'] > $user['User']['access']
          && $user_data['id'] != $user['User']['id']) {
        echo $html->link('Promote','/users/promote/' . $user['User']['id']);
        echo " ";
      }
      if ($user_data['access'] >= $user['User']['access']
          && $user['User']['access'] != 0) {
        echo $html->link('Demote','/users/demote/' . $user['User']['id']);
      }
?>
```

Save the view and pop over to the database and set your user's access level to 8, so you can do some promotion and demotion. (You'll need to log in and log out for the change to take effect.) Then go to http://localhost/users and try out your new powers. You may have to register a few additional users so you have test cases.

This is very basic user-type management. You can promote and demote users, and you're doing some checks around the user permissions before doing so. The next step will be to promote and demote the access levels of your content (entries and uploads).

Section 5. Content access levels

Before you can control access to the various types of content in Criki, you need to define what the access levels are for the content, and provide a mechanism for promoting and demoting the content itself. This will mean promote/demote actions for entries and uploads.

Access levels

The access levels for files and entries will look almost exactly like the access levels for the users:

- Content with access level 0 can be accessed by anyone.
- Content with access level 4 can only be access by editors and administrators.
- Content with access level 8 can only be accessed by administrators.

In these rules, "accessed" means "viewed and/or modified." You could get very granular down the line defining permissions, but these access levels will suffice as broad examples.

The rest of the tutorial will deal primarily with entries: promoting and demoting access levels, verifying access before taking actions, etc. You'll need to make the same kinds of changes for the uploads later.

Creating promote and demote actions for the entries

The promote action for the entries controller will look similar to the one you created for users.

Listing 17. Entries promote action

```
function promote($title = null) {
  if ($this->Session->check('User')) {
    $user = $this->Session->read('User');
    if(!$title) {
      $this->Session->setFlash('Invalid title for Entry');
      $this->redirect('/entries/index');
      exit;
    }
    $user = $this->Entry->User->read(null, $user['id']);
    if ($user['User']['access'] == 0) {
      $this->Session->setFlash('Contributors cannot promote.');
      $this->redirect('/entries/view/'.$title);
      exit;
    }
    $subject = $this->Entry->findByTitle($title);
    if ($user['User']['access'] > $subject['Entry']['access']) {
```

```
      $subject['Entry']['access'] += 4;
      $this->Entry->save($subject);
      $this->Session->setFlash('The Entry has been promoted');
      $this->redirect('/entries/view/'.$title);
    } else {
      $this->Session->setFlash('You cannot promote an
          Entry of equal or higher clearance');
      $this->redirect('/entries/view/'.$title);
    }
  } else {
    $this->Session->setFlash('You must be logged in to perform this action');
    $this->redirect('/entries/login');
  }
}
```

The primary differences are that entries are driven by title, not by ID, and you don't have to verify that the entry is trying to promote itself. Everything else will look basically the same. This is also true of the demote action.

Listing 18. Entries demote action

```
function demote($title = null) {
  if ($this->Session->check('User')) {
    $user = $this->Session->read('User');
    if(!$title) {
      $this->Session->setFlash('Invalid title for Entry');
      $this->redirect('/entries/index');
      exit;
    }
    $user = $this->Entry->User->read(null, $user['id']);
    if ($user['User']['access'] == 0) {
      $this->Session->setFlash('Contributors cannot demote.');
      $this->redirect('/entries/view/'.$title);
      exit;
    }
    $subject = $this->Entry->findByTitle($title);
    if ($subject['Entry']['access'] == 0) {
      $this->Session->setFlash('This Entry cannot be demoted any further.');
      $this->redirect('/entries/view/'.$title);
      exit;
    }
    if ($user['User']['access'] >= $subject['Entry']['access']) {
      $subject['Entry']['access'] -= 4;
      $this->Entry->save($subject);
      $this->Session->setFlash('The Entry has been demoted');
      $this->redirect('/entries/view/'.$title);
    } else {
      $this->Session->setFlash('You cannot demote a Entry of higher clearance');
      $this->redirect('/entries/view/'.$title);
    }
  } else {
    $this->Session->setFlash('You must be logged in to perform this action');
    $this->redirect('/users/login');
  }
}
```

Again, everything in this action is driven by title, not ID, and you don't have to verify that the entry is trying to demote itself.

With the promote and demote actions completed, you can move on to modifying the entries index view to show the correct links.

Showing the right links

Displaying the correct links on the index view looks similar to the code you used to show/hide the promote/demote links on the users index view.

Listing 19. Entries index view update

```
<?php $user_data = $session->read('User');
  if ($user_data['access'] > $entry['Entry']['access']) {
    echo $html->link('Promote','/entries/promote/' . $entry['Entry']['title']);
    echo " ";
  }
  if ($user_data['access'] >= $entry['Entry']['access']
      && $entry['Entry']['access'] != 0) {
    echo $html->link('Demote','/entries/demote/' . $entry['Entry']['title']);
  }
?>
```

Make the change to the index view, save it, and go to http://localhost/entries and try out your new promote/demote buttons. You should find that you cannot promote content so high that you can't read it, and you can't demote content below 0.

Now that you have user types defined and content access levels in place, you can apply access control to the content. After all the groundwork you have laid, you will find this remarkably easy.

Applying access controls

You've set the stage. Your users have access levels defined. So do your entries. Now it's time to put the two together and apply access controls to your content. Again, this will be applied only to the entries in this tutorial. You will need to apply the same principles to the uploads later.

Checking the access

In the entries controller, you need to check access rights for any action related to a specific entry. This has already been done for the promote and demote actions, so you should only need to add the access control to the edit and view actions.

It's as simple as adding the following lines after the entry has been retrieved.

Listing 20. Code to control access

```
$user = $this->Session->read('User');
$user = $this->Entry->User->read(null, $user['id']);
if ($user['User']['access'] < $entry['Entry']['access']) {
  $this->Session->setFlash('Access Denied.');
  $this->redirect('/entries/index');
}
```

Walking through the code in English, you're pulling fresh user information from the database. (If the user has been promoted or demoted while logged in, the access level in session will be inaccurate.) If the user has an access level below that of the

entry, he is refused access.

Go ahead and add the code to the view and edit actions for the entries controller. You can use the code in the archive for reference if need be. When you're done, try viewing or editing entries above your access level, and you will be met with an Access Denied error.

Filling in the gaps

You've got all kinds of room for improvement in Criki. But there are some specific tasks you should complete between now and starting Part 4.

1. Using the principles demonstrated in promoting, demoting, and protecting entries, add the code necessary to promote, demote, and protect uploads.
2. Go through all the controllers and remove any actions not currently in use. This includes the review action from the entries controller.
3. Similarly, go through the views and remove links to actions that are no longer valid.
4. Just as you performed an access check to determine if you should show or hide the promote/demote buttons, you could use the same access check to show the view/edit links, or to hide content completely to which the user has no rights. Spend some time experimenting with this and see what you find. You could also take the opportunity to streamline the menu bar in the default layout and link revisions to specific articles or uploads.
5. The access control system, as it has been designed, presents two problems. How would you address the following?
    1. User access levels changes require a login/login to take full effect.
    2. Revisions retain access levels from the past, meaning that promoting or demoting an entry or upload does not change the access levels of any related revisions.
6. Think about the wiki markup for linking to an uploaded file.

That's plenty to keep you going, for certain. Happy coding.

Section 6. Summary

You now have file uploads working. You're tracking revisions for the uploaded files

and sending them to the user. You have a system in place for promoting and demoting users and content, and you're able to control access to the content. Criki continues to grow, as do your skills. Why don't you put some of them to use before you start Part 4?

Downloads

Description	Name	Size	Download method
Part 3 source code	os-php-wiki3.source.zip	22KB	HTTP

Information about download methods

Resources

Learn

- Read Part 1 and Part 2 of this "Create an interactive production wiki using PHP" series.
- Check out the Wikipedia entry for wiki.
- Check out WikiWikiWeb for a good discussion about wikis.
- Visit the official home of CakePHP.
- Check out the "Cook up Web sites fast with CakePHP" tutorial series for a good place to get started.
- 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.
- Find out more about how PHP handles sessions.
- Check out the official PHP documentation.
- Read the five-part "Mastering Ajax" series on developerWorks for a comprehensive overview of Ajax.
- Check out the "Considering Ajax" series to learn what developers need to know before using Ajax techniques when creating a Web site.
- CakePHP Data Validation uses PHP Perl-compatible regular expressions.
- See a tutorial on "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 the Model-View-Controller on Wikipedia.
- Here is more useful background on the Model-View-Controller.
- Here's a whole list of different types of software design patterns.
- Read more about Design Patterns.
- PHP.net is the 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.

- 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.
- Visit Safari Books Online for a wealth of resources for open source technologies.

Get products and technologies

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

Discuss

- Participate in developerWorks blogs and get involved in the developerWorks community.
- Participate in the developerWorks PHP Developer Forum.

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.

Create an interactive production wiki using PHP, Part 4: Task management

Customizing the controller and modifying the views

Skill Level: Intermediate

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

03 Apr 2007

This "Create an interactive production wiki using PHP" tutorial series creates a wiki from scratch using PHP, with value-added features useful for tracking production. Wikis are widely used as tools to help speed development, increase productivity and educate others. Each part of the series develops integral parts of the wiki until it is complete and ready for prime time, with features including file uploading, a calendaring "milestone" system, and an open blog. The wiki will also contain projects whose permissions are customizable to certain users and will contain projects whose permissions are customizable to certain users. In Part 3, we added some control over who can do what. Now it's time to add some task management.

Section 1. Before you start

This "Create an interactive production wiki using PHP" series is designed for PHP application developers who want to to take a run at making their own custom wikis. You'll define everything about the application, from the database all the way up to the wiki markup you want to use. In the final product, you will be able to configure much of the application at a granular level, from who can edit pages to how open the blog really is.

At the end of this tutorial, Part 4 of a five-part series, you will have basic task management functionality working in your wiki, including assigning tasks, viewing tasks, and marking task progress.

About this series

Part 1 of this series draws the big picture. You determine how you want the application to look, flow, work, and behave. You'll design the database and rough-out some scaffolding. Part 2 focuses on the primary wiki development, including defining the markup, tracking changes, and file uploads. In Part 3, you define some users and groups, as well as a way to control access to certain aspects of individual wiki pages and uploaded files. Part 4 deals with a Calendaring and Milestones feature to track tasks, to-dos, and progress against set goals. And in Part 5, you put together an open blog to allow discussion of production topics and concerns.

About this tutorial

This tutorial deals mainly with task management. Criki (your new wiki engine) has all the basic wiki features you need, but it still lacks those features that will make it useful as a tool to assist in production. When it comes to production, task management stands at the top of the needed features list.

Covered topics include:

- Tasks workflow design
- Building out the tasks database table
- Basic task management features

Prerequisites

It is assumed you have completed Part 1, Part 2, and Part 3 of this "Create an interactive production wiki using PHP" series. And it is assumed that you have some experience working with the PHP programming language and MySQL. We won't be doing a lot of deep database tuning, so as long as you know the basic ins and outs, you should be fine.

System requirements

Before you begin, you need to have an environment in which you can work. The general requirements are reasonably minimal:

- An HTTP server that supports sessions (and preferably mod_rewrite). This tutorial was written using Apache V1.3 with mod_rewrite enabled.
- PHP V4.3.2 or later (including PHP V5). This was written using PHP V5.0.4

- Any version of MySQL from the last few years will do. This was written using MySQL V4.1.15.

You'll also need a database and database user ready for your application to use. The tutorial will provide syntax for creating any necessary tables in MySQL.

Additionally, to save time, we will be developing Criki using a PHP framework called CakePHP. Download CakePHP by visiting CakeForge.org and downloading the latest stable version. This tutorial was written using V1.1.13. For information about installing and configuring CakePHP, check out the tutorial series titled "Cook up Web sites fast with CakePHP" (see Resources).

In addition, you may find it helpful to download and install phpMyAdmin, a browser-based administration console for your MySQL Database.

Section 2. Criki so far

At the end of Part 3, you were given a few things to work on. You needed to add `accesscControl` to uploaded files. There was some cleanup work to be done in the controllers and the views. You should have experimented with using access checks to display or hide links and content. There were a couple problems with the access control system to be worked out. And you never defined any wiki markup for linking up uploaded files. How did you do?

Uploaded file access control

Defining permissions and access controls for file uploads should look much like it did for entries. In the uploads controller, you want to add code to the fetch action to verify the user's access level before serving the file.

Listing 1. Access control in the uploads fetch action

```
...
$upload = $this->Upload->read(null, $id);
if ($upload) {
  $user = $this->Session->read('User');
  $user = $this->Upload->User->read(null, $user['id']);
  if ($user['User']['access'] < $upload['Upload']['access']) {
    $this->Session->setFlash('Access Denied.');
    $this->redirect('/uploads/index');
    exit;
  }
  header('Content-Type: application/octet-stream');

...
```

You also need some promote/demote code added to the uploads controller to allow the user to protect the files. The promote and demote actions look almost identical to

those you created for the entries controller. Rather than reproduce both here, just the promote action has been shown. Both actions are included in the code archive below.

Listing 2. Uploads controller promote action

```
function promote($id = null) {
  if ($this->Session->check('User')) {
    $user = $this->Session->read('User');
    if(!$id) {
      $this->Session->setFlash('Invalid id for Upload');
      $this->redirect('/uploads/index');
      exit;
    }
    $user = $this->Upload->User->read(null, $user['id']);
    if ($user['User']['access'] == 0) {
      $this->Session->setFlash('Contributors cannot promote.');
      $this->redirect('/uploads/view/'.$id);
      exit;
    }
    $subject = $this->Upload->findById($id);
    if ($user['User']['access'] > $subject['Upload']['access']) {
      $subject['Upload']['access'] += 4;
      $this->Upload->save($subject);
      $this->Session->setFlash('The Upload has been promoted');
      $this->redirect('/uploads/view/'.$id);
    } else {
$this->Session->setFlash('You cannot promote an Upload of equal or higher clearance');
      $this->redirect('/uploads/view/'.$id);
    }
  } else {
    $this->Session->setFlash('You must be logged in to perform this action');
    $this->redirect('/users/login');
  }
}
```

Finally, you'll need to modify the uploads views (index and view to show the appropriate promote/demote links. This will look much like it did for entries, with the exception that you are passing the upload id, not the title. For example, in the index view, the code to show the promote/demote links might look like Listing 3.

Listing 3. Promote/demote links in uploads index view

```
  $user_data = $session->read('User');
  if ($user_data['access'] > $upload['Upload']['access']) {
    echo $html->link('Promote','/uploads/promote/' . $upload['Upload']['id']);
    echo " ";
  }
  if ($user_data['access'] >= $upload['Upload']['access'] && $upload['Upload']['access']
!= 0) {
    echo $html->link('Demote','/uploads/demote/' . $upload['Upload']['id']);
  }
```

Remember: There's an outstanding task related to access control on revisions. We will address that shortly.

Controller and view cleanup (including enhanced access checks)

There were many unused actions and links across the controllers and views as a result of the initial baking you did and the shape Criki took. The changes are too numerous to spell out here. The updated code in the source code for this tutorial contains cleaned-up versions of all the controllers and views. If you have deviated heavily from the code as provided, you should make sure to go through the code archive to see what changes have been made.

Mainly, you just want to be sure that unused actions (like the delete action for the uploads controller) and links to the unused actions (like the link to the delete action in the uploads *view* view) have been removed. See the source code for details.

Included in the view updates are the enhanced access checks to show promote/demote links for entries and uploads in useful places, such as the related *view* views. The code for showing these links looks like it did when you wrote it for the index views. See the source code for details.

Access control issues

There were two main access control issues you needed to address: user access-level changes requiring a user login/logout to take effect (for proper view rendering), and access control across revisions.

Updating user access levels without login/logout

The solution to this particular problem is simple. Recall that when a user logs in, the user data is set into the session.

```
$this->Session->write('User',
$this->User->findByUsername($this->data['User']['username']));
```

The simplest way to avoid forcing a login/logout for updated user permissions is to take advantage of the various access checks you perform in the controllers. Anytime you retrieve the user's information to verify access, write the information back into to session. For example, in the view action of the entries controller, the user information is retrieved to verify access levels.

Listing 4. Retrieving user data in the entries controller

```
...
if ($entry) {
  $user = $this->Session->read('User');
  $user = $this->Entry->User->read(null, $user['id']);
  if ($user['User']['access'] < $entry['Entry']['access']) {
    $this->Session->setFlash('Access Denied.');
    $this->redirect('/entries/index');
  }
...
```

This is a good opportunity to write the information back into session.

Listing 5. Amended entries user retrieval code

```
...
if ($entry) {
  $user = $this->Session->read('User');
  $user = $this->Entry->User->read(null, $user['id']);
  $this->Session->write('User', $user);
  if ($user['User']['access'] < $entry['Entry']['access']) {
    $this->Session->setFlash('Access Denied.');
    $this->redirect('/entries/index');
  }
...
```

If you really wanted to, you could just pull the user's information on every page load, but that adds an additional query to each page load, which isn't efficient.

Access control across revisions

When you promote or demote an entry or an upload, any associated revisions are not promoted or demoted. It may not be necessary to promote or demote revisions. But if you find it necessary to do so, you can do it in one of two ways: by taking advantage of the relationship to the revisions table and updating the revisions with the new access level (which is heavy-handed, but would probably be a little tighter in terms of security), or you can check the access level of the master record when viewing a revision and allow or deny access based on the access check. Neither method seems especially useful to Criki, as the revisions will retain the access level they held before revision. Editing an entry with an access level of 4 will create a revision with an access level of 4. This should be sufficient, provided you add basic access controls as outlined in the source code.

Wiki markup for uploaded files

There are many ways to deal with wiki markup for uploaded files. The cheapest, laziest, and easiest way is to make your users just provide links to uploaded files, like any other link.

```
[[[http://localhost/uploads/view/1|Uploaded File]]]
```

For that matter, if the user pasted the URL to the uploaded file without using wiki markup, it would still be rendered as a link.

That's a pretty cheap way to go about it, but it demonstrates an important concept: It's not always necessary to reinvent the wheel.

A couple other ways to address the problem would be to either use new markup to indicate that a file is being referenced (wrapping the filename in +++, for example) or adding handling to the link-managing markup allowing the user to specify that the link is to a file, not an entry title -- such as [[[file:FileName.txt]]]. Neither concept is written into Criki at this time. If you've got a great solution to the problem, feel free to

incorporate it into the code from the source code.

You've filled in the gaps. It's time to sink your teeth into the next feature: task management.

Section 3. Defining user tasks

Since Criki is intended to be a wiki used to track production, the next logical step in developing Criki would be to add the ability to assign tasks to users and for users to be able to keep an eye on their tasks. This will represent a whole new workflow that hasn't yet been thought through. Before you jump into getting a table together writing some code, spend some time thinking about how the tasks workflow will look.

Thinking through the tasks workflow

Before you can write a line of code, you need to get a clear image in your head for how the tasks workflow will look. Unless you know where you're going, you'll find it hard to get there, or even know when you've gotten there.

There are several questions that need to be addressed when thinking through the tasks workflow. What information will need to be tracked for a task? Who can assign tasks to whom? How will a user mark a task completed? How will a user see what tasks are assigned to him? Considering these questions will help the tasks workflow to take on a definite shape.

What information will need to be tracked for a task?

Before you can start thinking about how to assign or view tasks, you have a more important question to answer: What is a *task*? You might say, "A thing that needs to be done." And that's a good place to start. But you need to know what the *thing* looks like. You can do this by defining what information you're tracking.

For each individual task, there will be specific pieces of information that need to be tracked. You should strive to keep this information minimal, but make sure to cover the basics. Tasks should be somewhat loose and conceptual, not tight and restrictive. For example, while you might track such things as last modified, last viewed, number of views, etc., most of that information isn't especially useful to the user. The following basic pieces of information represent the basic necessary information for a task:

1. Who has the task been assigned to?
2. Who assigned the task?

3. When is the task to be completed?
4. What is the task?
5. How much of the task has been completed?

It can also be helpful to include a title for the task, so that the task can be briefly described in a list. You might be able to argue that other information should be tracked, but this basic list should do for now.

Who can assign tasks to whom?

Now you know what a task is, and you probably are starting to think about what the a record in the tasks table will look like. You can start thinking about tasks and who can assign them.

In keeping with the open spirit of Criki, the tasks workflow will not be tightly controlled. The rules about who can assign tasks to whom are very simple:

- A user must be logged in to assign a task.
- A user cannot assign a task to someone with an access level higher than his own. For example, an editor cannot assign a task to an administrator, but an editor can assign a task to another editor.

The idea here, once again, is to leverage the basic trust at work within the structure of the wiki. Derive strength from the openness of the wiki. In some cases, this structure might not work for you. You may want tighter control over how tasks are assigned. After you get the basic tasks structure built, you can tweak it to suit your own needs.

How will a user's tasks be viewed?

You know what a task looks like, and you know who can assign them. As a user, now you need to be able to see your tasks.

When it comes to viewing tasks, there are several ways to display task information to the user. A list of tasks across users will be helpful when trying to determine which users have less to do. Viewing a list of tasks for a specific user will be helpful for the user and for anyone interested in the user's current workload. Of course, individual tasks should be viewable. And displaying the information in a calendar-style format will make it easy for a user to visualize what tasks are coming up when.

The task structure as it is being defined doesn't include things like setting task permissions or controlling access to individual tasks. It's not necessary to include it at this time, but it would be a good exercise for you to try adding permissions and access controls to tasks, since you learned all about that in Part 3.

How will a user mark a task completed?

You can now see your task. You've finished whatever the task called for. Now you need to be able to mark your task complete. A user should be able to update tasks, indicating how far along he has gotten in completing a task.

A user shouldn't be able to change anything else about a task -- not the description, the title, or the due date. As for completion, this will be tracked as a basic percentage, broken into quarters: 0, 25, 50, 75, 100 percent. Getting more granular than this in tracking the percentage of task completion probably isn't necessary. The more choices you give the user, the more time he will spend trying to determine which is most accurate. In most cases, it's not really important if a task is 20 or 25 percent done because both numbers translate into "Not yet half."

It might be easier to think of the percentage breakdowns as phrases: "Not started," "Just started," "Half done," "Mostly done," and "Completed." Breaking the percentages into coarse pieces will also pay off when it comes time to display the tasks in a calendar.

You have a pretty good idea at this point for how the tasks workflow will be put together. Now you can jump in and start putting it together.

Defining the tasks database

At this point, you should have a pretty good idea for what your tasks database is going to look like. You'll want an ID field set to `auto_increment`, as with other tables. You'll need a field to hold the ID of the user to whom the task has been assigned, as well as a field to hold the ID of the user who assigned the task. You also need something to hold the due date, a title, description and percentage complete. The SQL to create the table is shown below.

Listing 6. Tasks table SQL

```
CREATE TABLE 'tasks' (
 'id' int(10) NOT NULL auto_increment,
 'user_id' int(10) NOT NULL default '0',
 'assigned_id' int(10) NOT NULL default '0',
 'duedate' datetime NOT NULL default '0000-00-00 00:00:00',
 'title' varchar(255) NOT NULL default ,
 'description' text NOT NULL,
 'percent' enum('0','25','50','75','100') NOT NULL default '0',
 PRIMARY KEY  ('id')
) ENGINE=MyISAM DEFAULT CHARSET=latin1 ;
```

It's all pretty straightforward, with the possible exception of the `user_id` and `assigned_id` fields, which represent the user to whom the task has been assigned and the assigning user, respectively. You may have questions as to how to establish the necessary model relationships to get the data back the way you want it. That's good. It means you're paying attention.

Creating the task model

Now that you have the tasks table created, you need to write the model to get data from the table. But the task model won't look like your other models. All the models you have defined so far have either had no model associations or very simple ones. For example, consider the entry model below.

Listing 7. Entry model

```
<?php
class Entry extends AppModel {

        var $name = 'Entry';

        var $belongsTo = array('User' => array (
                'className' => 'User',
                'conditions' => ,
                'order' => ,
                'foreignKey' => 'user_id'
                )
        );
}
?>
```

This model has a simple `belongsTo` relationship defined to indicate that the `user_id` field in the entries table references a row in the users table. If you have debugging turned on in CakePHP, you might see a SQL statement like this when viewing an entry (formatted to easier reading), as shown below.

Listing 8. SQL statement to retrieve an entry

```
SELECT
  'Entry'.'id',
  'Entry'.'title',
  'Entry'.'content',
  'Entry'.'access',
  'Entry'.'modified',
  'Entry'.'user_id',
  'Entry'.'ip',
  'Entry'.'revision',
  'Entry'.'accessed',
  'User'.'id',
  'User'.'username',
  'User'.'email',
  'User'.'password',
  'User'.'access',
  'User'.'created',
  'User'.'login'
FROM
   'entries' AS 'Entry'
LEFT JOIN
   'users' AS 'User'
ON
   'Entry'.'user_id' = 'User'.'id'
WHERE
   ('Entry'.'title' = 'foo')
LIMIT 1
```

If you've worked with SQL much at all, you should be able to parse this statement fairly easily. It's a query on the entries table, which is joined to users by the

`user_id` field.

But what you want for a task is to join tasks to the users table on two different fields: `user_id` (the user to whom the task has been assigned) and `assigned_id` (the user who assigned the task). You can probably write out the SQL statement just fine.

Listing 9. SQL statement to retrieve a task

```
SELECT
  'Task'.'id',
  'Task'.'user_id',
  'Task'.'assigned_id',
  'Task'.'title',
  'Task'.'description',
  'Task'.'duedate',
  'Task'.'percent',
  'User'.'id',
  'User'.'username',
  'User'.'email',
  'User'.'password',
  'User'.'access',
  'User'.'created',
  'User'.'login',
  'Assigned'.'id',
  'Assigned'.'username',
  'Assigned'.'email',
  'Assigned'.'password',
  'Assigned'.'access',
  'Assigned'.'created',
  'Assigned'.'login'
FROM
   'tasks' AS 'Task'
LEFT JOIN
   'users' AS 'User'
ON
   'Task'.'user_id' = 'User'.'id'
LEFT JOIN
   'users' AS 'Assigned'
ON
   'Task'.'assigned_id' = 'Assigned'.'id'
WHERE
   ('Task'.'id' = 1)
LIMIT 1
```

It's a long query, but it's not terribly complicated. And it may seem difficult to get the model to retrieve the data in this way. But it's actually very simple. When you define the `belongsTo` for the task model, you will define two tables, both of them users, but with different aliases. It should look like Listing 10.

Listing 10. Task model with belongsTo aliases

```
<?php
class Task extends AppModel {

       var $name = 'Task';

       var $belongsTo = array(
               'User' => array (
                       'className' => 'User',
                       'conditions' => ,
                       'order' => ,
                       'foreignKey' => 'user_id'
               ),
               'Assigned'=>array(
```

```
                        'className'=>'User',
                        'conditions' => ,
                        'order' => ,
                        'foreignKey' => 'assigned_id'

                )
        );

}
?>
```

You are basically just aliasing the user as assigned, and specifying the foreign key to be used. Easy as pie.

But you won't be able to see this in action until you get your controllers and views into place. You can short-circuit the process by using Bake, as you did before.

Baking the tasks controller and views

I hope you are somewhat accustomed to using Bake at this point. You used it in Part 1, and you've probably played with it on your own or followed the "Cook Up Web sites Fast with CakePHP tutorial series" (see Resources), which also walks you through using Bake. However, as a reminder, you need to make sure the PHP executable is in your `PATH` and that you have changed into the directory where you installed Cake.

To run bake, use `php cake\scripts\bake.php`. Walk through the menus specifying that you want to bake first a controller for the tasks table. When you have baked a controller, bake the views for tasks. If you need a refresher on the bake menus, consult Part 1 of this series.

Once you have the controller and views baked, turn the debugging up to at least 2 if you haven't already. This will show you the SQL involved in page rendering. In app/config/core.php, use `define('DEBUG', 2);`.

That should be it. Create a couple basic tasks and check out the SQL statement that's generated when you view them. It should look exactly like the one you wrote out above to join tasks to users and assigned (the users alias).

That gives you the basic controller and views. Now it's time to tweak them to better suit your needs.

Section 4. Customizing the controller

Hang onto your hats. It's going to get bumpy.

The controller you baked should have had some basic actions in it to `index`, `add`,

`view`, `edit`, and `delete` tasks. None of the actions are suitable for Criki as written, and some will be deleted altogether. Looking back at the workflow you've designed for tasks, you should be able to make a list of things to be done with the tasks controller:

- The `index` action needs to accept filtering parameters. In this case, `user_id` and `duedate`.
- The `view` action needs to set a variable to show or hide the edit link.
- The `add` action needs heavy reworking to perform access checks and tweak or verify data before insertion.
- The `edit` action needs to be changed to set the values of the percent-select list.
- The `delete` action will go away completely.
- A new action will be added called `buildCalendar` and will be used to generate an array with dates and associated tasks, for use in displaying a `Calendar` element.

Rather than go over this line by line, the highlights will be covered, leaving you on your own to dig into the source code if you want to get a closer look at all the changes. Even taking this approach of focusing on the highlights, there's a lot to cover.

Modifying the index action

The `index` action is baked to pull all tasks and format them for display. You need to change this action so you can use it to pull all the tasks for a specific user, or all the tasks for a specific user on a specific day.

For starters, you'll want to specify that the index action receives two parameters: `$user_id` and `$duedate`.

```
function index($user_id = null, $duedate = null) {
```

Now when you access http://localhost/tasks/USERID/DUEDATE, you pass in the value of `USERID` as `$user_id` and the value of `DUEDATE` as `$duedate`.

But passing the `DUEDATE` will be much easier to pass as a UNIX® timestamp rather than a date-formatted string. This will mean additional work, as the database will be expecting a date-formatted string. You will translate back and forth between the two.

The rest of the action is pretty straightforward. Since you want the index action to pull all tasks, all tasks for a user, or all tasks for a user on a date, you need to check to see what parameters have been passed, if any, and tailor your data retrieval accordingly. For details, consult the source code.

Modifying the view action

The `view` action doesn't need much modification. You just need to set a variable the view can use to show or hide an edit link for the task. The rule here is that a task can only be edited by the user to whom it has been assigned.

Listing 11. Setting the showedit variable

```
$showedit = false;
if ($this->Session->check('User')) {
  $user = $this->Session->read('User');
  if ($task['Task']['user_id'] == $user['id']) {
    $showedit = true;
  }
}
$this->set('showedit', $showedit);
```

The rest of the action is pretty much the same, though the view will be heavily edited.

Modifying the add action

The `add` action needs the most modification. You have more checking to do at this step, and some additional data massaging is also involved.

For starters, you'll need to make sure that the task is not being assigned in the past.

Listing 12. Deny adding past tasks

```
if ($this->data['Task']['duedate'] < strtotime('today')
                && $duedate < strtotime('today')) {
  $this->Session->setFlash('you cannot assign tasks in the past');
  $this->redirect('/tasks/index');
  exit;
}
```

It's important to call `exit` after the redirect, as redirect does not imply exit and the action could continue to execute.

Next, you need to make sure the target user exists and that the target user's access level is not higher than the assigning user.

Listing 13. Verify target user exists and access levels

```
if (!$target) {
  $this->Session->setFlash('User not found');
  $this->redirect('/tasks/index');
  exit;
} else if ($target['User']['access'] > $user['access']) {
  $this->Session->
                setFlash('You cannot assign tasks to users with higher access than your
own.');
  $this->redirect('/users/login');
```

```
  exit;
}
```

The last main point is to make sure you remember to set the `'assigned_id'` value to be the ID of the user performing the action. By the time you get to that stage of the action, the logged-in user's information is in the variable `$user`. While you are setting that value, you should set the default percentage of completion to 0, and format the `duedate`.

```
$this->data['Task']['percent'] = '0';
$this->data['Task']['duedate'] = date('Y-m-d H:i:s', $this->data['Task']['duedate']);
$this->data['Task']['assigned_id'] = $user['id'];
```

The rest of the action should be fairly straightforward. Consult the source code for details.

Modifying the edit action

The `edit` action doesn't need much work (or does it?). Mainly, you need to set the `$percents` variable for the view to use when rendering the percent-select list.

```
$this->set('percents', array ('0' => '0%', '25' => '25%', '50' => '50%',
        '75' => '75%', '100' => '100%'));
```

You'll be modifying the view later to only display the percent field for modification.

Adding the buildCalendar action

If you've been looking at the actions as defined in the tasks controller in the source code, you'll have seen references to a new action: `buildCalendar`. They look something like this:

```
$this->set('month', $this->buildCalendar($task['Task']['user_id'],
        $task['Task']['duedate']));
```

Setting the output of the `buildCalendar` action to the `month` variable allows you to create something new: a `Calendar` element. This will allow you to put the `Calendar` wherever you like, without having to reproduce lots of code. It's a great time-saver.

This new action will be used to generate an array of dates to be displayed in a standard `Calendar` format. The `buildCalendar` action will need to accept two parameters: `$user_id` (the user for whom the `Calendar` is being displayed) and `$date` (the base date to be used when building the `Calendar` data).

This action begins by looking for tasks for the provided `$user_id`, building out an

array of dates if there are tasks, and including the status (percentage complete) of the task.

Listing 14. Building a dates/tasks array

```
$dates = array();
if ($user_id) {
  $tasks = $this->Task->findAllByUserId($user_id);
  foreach ($tasks as $task) {
    $dates[strtotime($task['Task']['duedate'])][] = $task['Task']['percent'];
  }
}
```

Once this array is built, the base date needs to be evaluated. Since you are building a list of date information for use in displaying a `Calendar`, you want the first date to be a Sunday. Therefore, check to see what day the base date is and set `base_date` to the previous Sunday if the date is not a Sunday.

Listing 15. Tweaking the base date

```
$base_date = date('Y-m-d', strtotime($date));
$dow = date('w', strtotime($date));
if ($dow != 0) {
  $base_date = date('Y-m-d', strtotime($base_date . '- ' . $dow . ' days'));
}
```

The number of weeks to be shown has been hardcoded to three. This will allow you to include the `Calendar` on any page without taking up too much real estate, while still allowing a multiweek look at what needs to be done.

Listing 16. Initialize the $month variable

```
$month = array(
  '1' => array(),
  '2' => array(),
  '3' => array(),
);
```

Finally, you'll need to iterate through the days following the `base_date`, checking to see if there are tasks for that day. If there are tasks, update the status of that day to the least-completed task. You will use this information to color-code the `Calendar`.

Listing 17. Walking the dates

```
$n=0;
for ($i = 0; $i < 21; $i++) {
  $status = null;
  if ($i % 7 == 0) {
    $n++;
  }
  $date = strtotime($base_date . ' +' . $i . ' days');
  if ($dates && array_key_exists($date, $dates)) {
    foreach ($dates[$date] as $task) {
      if ($status == null) {
        $status = $task;
      } else {
```

```
        $status = $task < $status ? $task : $status;
      }
    }
  }
  $month[$n][] = array('date' => $date, 'status' => $status);
}
```

Last but not least, slide in the user data for the `Calendar` (so you don't have to call it up later) and return the data.

```
$month['user_data'] = $this->Task->User->findById($user_id);
return $month;
```

Phew! That's a lot of groundwork. But now that you have built the `buildCalendar` action and set its output to the month variable, you can create an element to take advantage of it.

Creating the Calendar element

In CakePHP, an element is simply a small block of presentation code you want to display in multiple places. A good example would be a list of menu items. Like other views, an element is just a PHP file that specifically outputs formatted data in HTML.

In Criki, you've laid the groundwork to use a `Calendar` element. In the controller, you made an action to create an array of data that the `Calendar` element can use. You've called the action a few times and assigned the results to the `month` variable. Any subsequent view or action should be able to then access the information via the `$month` variable.

The Calendar element template

Take a look at app/views/elements/calendar.thtml in the source code. The full text is too long to reproduce here, but most of it is HTML anyway. The important parts will be covered. For starters, look at the top of the element below.

Listing 18. Calendar element variable inits

```
<?php
  $caluser = $month['user_data'];
  unset($month['user_data']);
  $caluser_id = $caluser['User']['id'];
  $calusername = $caluser['User']['username'];
  $begin = current(current($month));
  $end = end(end($month));
  $base_month = date('M', $begin['date']);
  $base_day = date('d', $begin['date']);
  $base_year = date('o', $begin['date']);
  $last_month = date('M', $end['date']);
  $last_day = date('d', $end['date']);
  $last_year = date('o', $end['date']);
  $base_color = '#eeeeee';
  $colors = array (
    '0' => '#dd0000',
```

```
    '25' => '#ff9966',
    '50' => '#ffff00',
    '75' => '#ccff33',
    '100' => '#00cc66',
  );
?>
```

The file starts by pulling the user data out of the `$month` variable, setting some variables for use later. Then, the beginning and ending dates from the `$month` array are identified, so that the `Calendar` element can be labeled intelligently. Finally, some colors are defined, using the same coarse percentages you defined earlier as array keys. This will allow you to set color-coded backgrounds in the `Calendar` element, ranging from Red (0-percent complete) to Yellow (50-percent complete) to Green (100-percent complete).

The only other potentially sticky bit is setting the background color for a specific day. It's helpful to change the background colors slightly if the `Calendar` element spans multiple months, but mainly you want to show the background color that corresponds to the completion status of the tasks for the given day.

Listing 19. Setting Calendar day background colors

```
<?php
  foreach ($week as $day):
    if ($base_month != date('F', $day['date'])) :
      $base_month = date('F', $day['date']);
      $base_year = date('o', $day['date']);
      $base_color = $base_color == '#eeeeee' ? '#dddddd' : '#eeeeee';
    endif;
    if ($day['status'] != null) {
      $color = $colors[$day['status']];
    } else {
      $color = $base_color;
    }
    $bg_color = $color;
?>
```

The rest of the `Calendar` element should look like a regular view. The days themselves link to the tasks view for the user in question.

Now that you've got the `Calendar` element defined, you need to put it someplace where it can be used.

Modifying the default layout

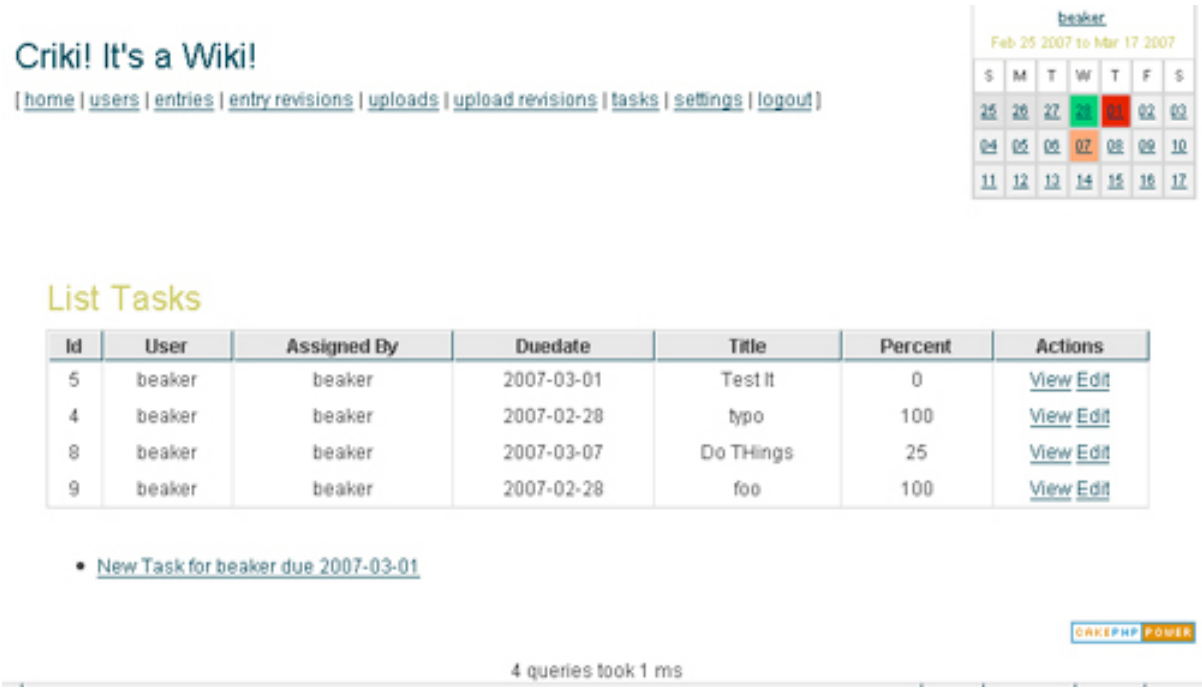
The easiest way to include the `Calendar` element would be to conditionally include it in the default layout. You will recall that you modified app/views/layouts/default.thtml to customize the look and feel of Criki. By modifying this file further, you can include the `Calendar` element. You simply check to see if the `$month` variable has been defined. If it has, include the element as shown below.

Listing 20. Including the Calendar element

```
<?php if (isset($month)) : ?>
<div id="calendar">
<?php echo $this->renderElement('calendar', array("month" => $month)); ?>
</div><?php endif; ?>
```

If you have some tasks defined, and if you are using the code from the source code for this tutorial, you should be able to view the tasks for a user and see the `Calendar` element in action.

Figure 1. Calendar element in action



In order to get the element to fit into the layout, some modification was done to the default CSS file (app/webroot/css/cake.generic.css). If your layout doesn't look much like the screenshot, make sure you replaced your version of the default CSS file with the one from the source code.

Setting the Calendar data from the users controller

It would be especially helpful to see the `Calendar` element for a given user when viewing that user's profile. To do this, you will need to use modify the view action in the users controller to set the month variable using the `requestAction` method.

Listing 21. Modifying the view action in the users controller

```
function view($id = null) {
  if(!$id) {
    $this->Session->setFlash('Invalid id for User.');
    $this->redirect('/user/index');
  }
  $this->set('user', $this->User->read(null, $id));
  $this->set('month',
          $this->requestAction('/tasks/buildCalendar/'.$id.'/'.date('Y-m-d')));
  $this->set('user_id', $id);
```

```
}
```

The `requestAction` method is used by a controller to call an action on another controller. You'll remember that the `buildCalendar` action took in two parameters: the `$user_id` and the `$date` to be used as the `base_date`. Calling the action with `requestAction` should look familiar to you -- it's exactly like calling it via a URL.

Once you've modified the view action in the users controller, view any user, preferably one with tasks assigned to him. It should look like Figure 2.

Figure 2. User view with Calendar element

You've got the `Calendar` element working, and you're putting it to good use. Now it's time to clean up those task views you baked earlier.

Section 5. Customizing the views

You're almost there. Your controller should be in good shape. You've got a fancy `Calendar` element to show what tasks are coming up at a glance. Now you need to make some modifications to those views you baked earlier. Each view needs modified in some way. As with the controllers, rather than fully reproduce the code for each view, the primary changes will be highlighted.

Modifying the index view

The index view modifications are very straightforward. You'll want to show the user

name for the user to whom the task has been assigned rather than the user's ID. The same goes for the user who assigned the task. You can omit displaying the full description of the task -- that's why you included the shorter title field.

Of particular interest may be the following:

Listing 22. Showing the edit link

```
<?php
  if ($task['Task']['user_id'] == $user['id']) :
    echo $html->link('Edit','/tasks/edit/' . $task['Task']['id']);
  endif;
?>
```

Recall that the standing rule for editing a task is that only the user to whom the task has been assigned can edit the task, and even then, he can only change the percent-complete field. By checking in the view to see if the `user_id` for the task is the same is the ID of the logged-in user, you can determine if the edit link should be shown.

We see the whole view in the source code at app/views/tasks/index.thtml.

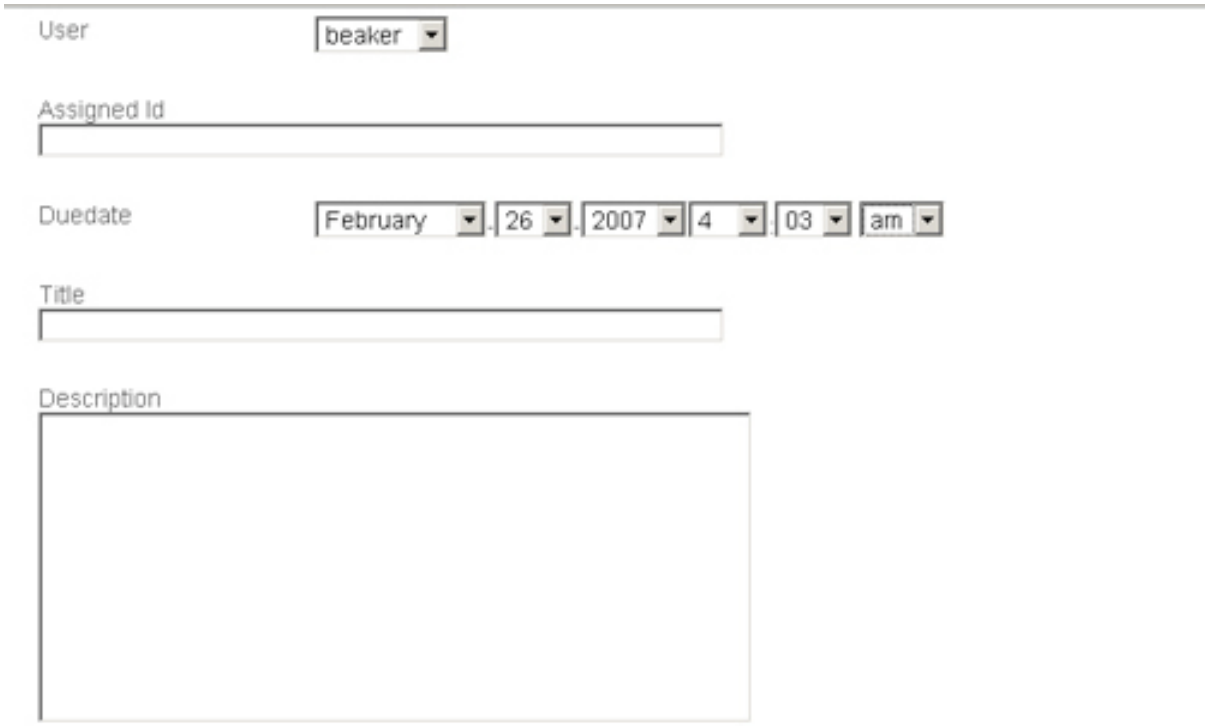
Modifying the add view

The add view doesn't need much modification at all. We want to trim back the fields the user can fill in, as fields like `assigned_id` should be set by the controller. As for the `duedate` field, the approach that has been taken is that it is passed to the initial request to load the add view, and the value is passed as a hidden field. In the view, the net result is a need to format the `duedate`.

```
<?php echo $form->labelTag('Task/duedate', 'Duedate');?>
<b><?php echo date('F d, Y', $task['Task']['duedate']) ?></b>
<?php echo $html->hidden('Task/duedate');?>
```

The idea behind this approach is to simplify the user experience by reducing the number of fields to be filled out. If you looked at the way the `duedate` field was rendered when baking the views, you'd have seen something like Figure 3.

Figure 3. Many fields in the add view



Rather than having to modify all those fields, it should be much simpler to click on a date in a user's calendar and create a task for him. If your users dislike the approach, you can experiment with different approaches and find what works.

Modifying the edit view

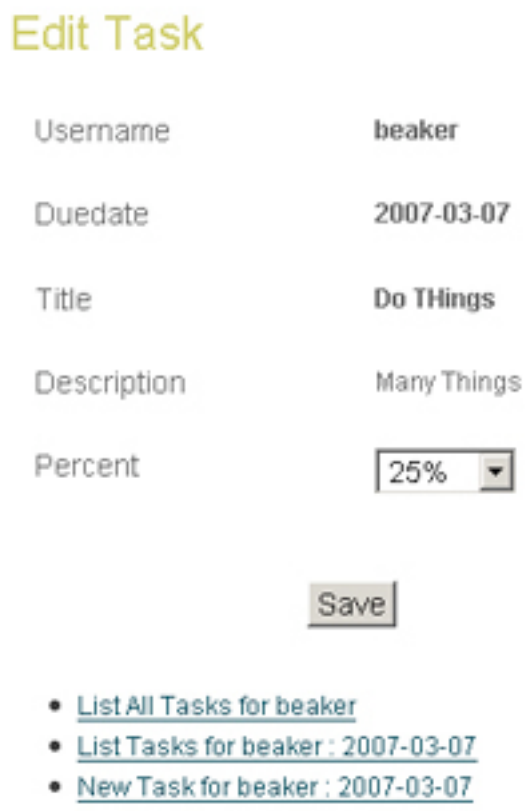
The most important thing about modifying the edit view is to remove all editable fields except for the percent field. You shouldn't remove all the information. You can display the user name and the due date. Just don't provide the ability for the user to modify the fields. The only field a user should need to modify is percent.

Listing 23. Edit view percent field

```
<?php echo $form->labelTag('Task/percent', 'Percent');?>
<?php echo $html->selectTag('Task/percent', $percents,
        $html->tagValue('Task/percent'), array(), array(), true);?>
<?php echo $html->tagErrorMsg('Task/percent', 'Please select the Percent.') ?>
```

Also of note in the edit view is some modification to the list of actions a user can perform. You have access to a lot of information in the edit and *view* views. You should use the information to provide convenient links (see Figure 4).

Figure 4. Edit view actions

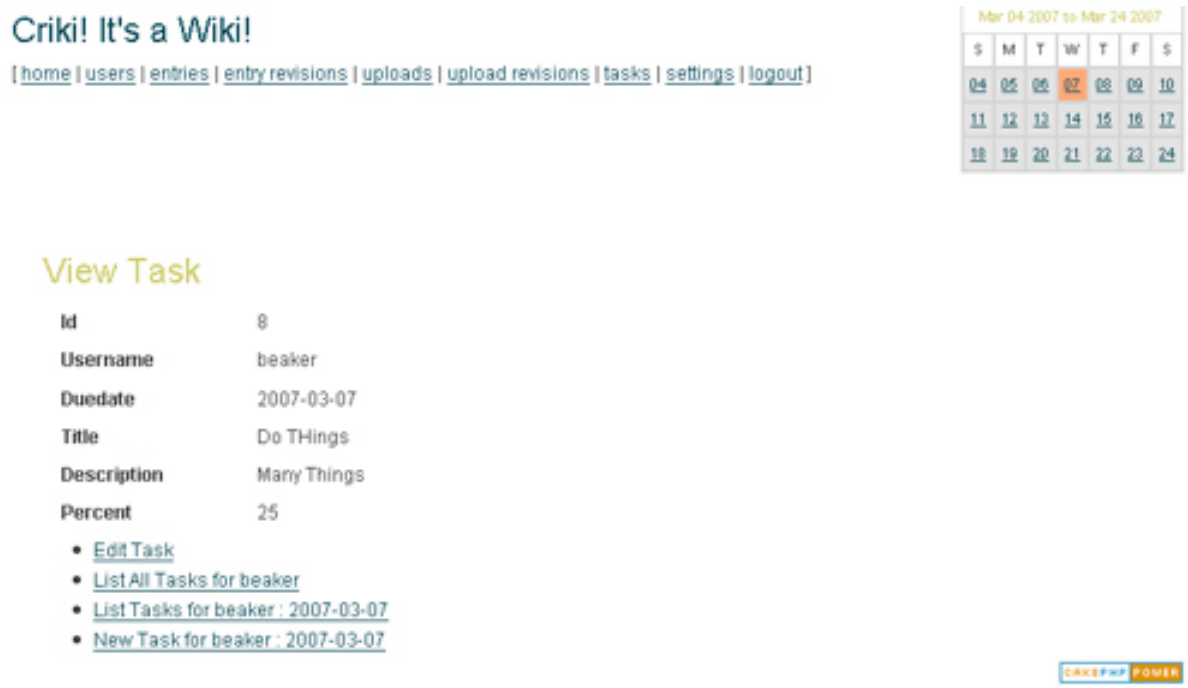


These same actions will be of particular use in the *view* view.

Modifying the view view

The *view* view shouldn't need much modification. Again, you'll want to show users' names, rather than their IDs. This is the only place that the full task description can be seen. Remember to leave it in. Mainly, you just want to modify the actions links to point to actions the user can perform. The links will look much like they did on the edit page.

Figure 5. View view actions



That should wrap it up! The controller is doing its job. Your `Calendar` element is working overtime. You have the views looking like they should. Spend some time playing with task management. See if you can find ways to improve it.

Filling in the gaps

As usual, there are many ways in which Criki can still be improved. See what you can do with these tasks:

- There's at least one glaring security hole in the task management edit workflow. Find it and fix it, as well as any others you recognize.
- Wouldn't it be great to be able to use the wiki markup when writing task descriptions? Come up with a way to render wiki markup in task descriptions *without* rewriting the wiki markup code yet again. (Hint: Take a look at how the `buildCalendar` action is put together.)
- In the tasks views from the source code, where user names are displayed as text, they could be displayed as links to the users controller to allow for easy viewing of a user's profile. What would that look like?

You are encouraged to get your fingers deep into the code. Find something you don't like? Think it should work differently? Think it's broken? Awesome. Fix it. It's the best way to learn.

Section 6. Summary

Good task management should now be easy for everyone. Assigning a task to a user should be intuitive. As a user, task management should help the user keep track of priorities, rather than hindering by burdening with overly complicated workflows. The task management system you have put in place for Criki should meet these basic standards. Be sure to read our final tutorial in this "Create an interactive production wiki using PHP" series (Part 5), where we add an open blog to Criki to allow discussion of production topics and concerns.

Downloads

Description	Name	Size	Download method
Part 4 source code	os-php-wiki4.source.zip	26KB	HTTP

Information about download methods

Resources

Learn

- Read Part 1, Part 2, and Part 3 of this "Create an interactive production wiki using PHP" series.
- Check out the Wikipedia entry for wiki.
- Check out WikiWikiWeb for a good discussion about wikis.
- Visit the official home of CakePHP.
- Check out the "Cook up Web sites fast with CakePHP" tutorial series for a good place to get started.
- 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.
- Find out more about how PHP handles sessions.
- Check out the official PHP documentation.
- Read the five-part "Mastering Ajax" series on developerWorks for a comprehensive overview of Ajax.
- Check out the "Considering Ajax" series to learn what developers need to know before using Ajax techniques when creating a Web site.
- CakePHP Data Validation uses PHP Perl-compatible regular expressions.
- See a tutorial on "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 the Model-View-Controller on Wikipedia.
- 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.
- Read more about Design Patterns.
- PHP.net is the 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.
- 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.

Get products and technologies

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

Discuss

- Participate in developerWorks blogs and get involved in the developerWorks community.
- Participate in the developerWorks PHP Developer Forum.

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.

Create an interactive production wiki using PHP, Part 5: The open blog

An environment for open discussion

Skill Level: Intermediate

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

10 Apr 2007

This "Create an interactive production wiki using PHP" tutorial series creates a wiki from scratch using PHP, with value-added features useful for tracking production. Wikis are widely used as tools to help speed development, increase productivity and educate others. Each part of the series develops integral parts of the wiki until it is complete and ready for prime time, with features including file uploading, a calendaring "milestone" system, and an open blog. The wiki will also contain projects whose permissions are customizable to certain users and will contain projects whose permissions are customizable to certain users. In Part 4 we added some task management. Now you will create an open blog, which will allow users a place to hold public discussions.

Section 1. Before you start

This "Create an interactive production wiki using PHP" series is designed for PHP application developers who want to to take a run at making their own custom wikis. You'll define everything about the application, from the database all the way up to the wiki markup you want to use. In the final product, you will be able to configure much of the application at a granular level, from who can edit pages to how open the blog really is.

At the end of this tutorial, Part 5 of a five-part series, you will have an open blog working in your wiki. You should not continue until you have completed the first four tutorials.

About this series

Part 1 of this series draws the big picture. You determine how you want the application to look, flow, work, and behave. You'll design the database and rough-out some scaffolding. Part 2 focuses on the primary wiki development, including defining the markup, tracking changes, and file uploads. In Part 3, you define some users and groups, as well as a way to control access to certain aspects of individual wiki pages and uploaded files. Part 4 deals with a Calendaring and Milestones feature to track tasks, to-dos, and progress against set goals. Here in Part 5, you put together an open blog to allow discussion of production topics and concerns.

About this tutorial

This tutorial deals with creating an open blog for Criki. You have built the basic wiki features, and you have added task management for your users. An open blog is another valuable feature that will allow your users a place to hold public discussions. Topics include:

- What is meant by "open blog"
- Blog workflow design
- Building out the blog database table
- Basic blog features

Prerequisites

It is assumed you have completed Part 1, Part 2, Part 3, and Part 4 of this "Create an interactive production wiki using PHP" series. And it is assumed that you have some experience working with the PHP programming language and MySQL. We won't be doing a lot of deep database tuning, so as long as you know the basic ins and outs, you should be fine.

System requirements

Before you begin, you need to have an environment in which you can work. The general requirements are reasonably minimal:

- An HTTP server that supports sessions (and preferably mod_rewrite). This tutorial was written using Apache V1.3 with mod_rewrite enabled.
- PHP V4.3.2 or later (including PHP V5). This was written using PHP V5.0.4

- Any version of MySQL from the last few years will do. This was written using MySQL V4.1.15.

You'll also need a database and database user ready for your application to use. The tutorial will provide syntax for creating any necessary tables in MySQL.

Additionally, to save time, we will be developing Criki using a PHP framework called CakePHP. Download CakePHP by visiting CakeForge.org and downloading the latest stable version. This tutorial was written using V1.1.13. For information about installing and configuring CakePHP, check out the tutorial series titled "Cook up Web sites fast with CakePHP" (see Resources).

Section 2. Criki so far

At the end of Part 4, you were given several items to complete. There was at least one glaring security hole in the task management edit workflow. You were tasked with finding a way to use wiki markup when writing task descriptions, without reproducing the wiki markup rendering code. And the tasks views contained user names as text, which could have been made into links to the users' profiles. How did you do?

Securing the task management edit workflow

There are two specific problems in the task management edit workflow to address. Alert code monkeys will have noticed them already. The first should be fairly obvious. Consider this line from the tasks edit view in Listing 1.

Listing 1. Tasks edit view excerpt

```
<?php
  if ($task['Task']['user_id'] == $user['id']) :
    echo $html->link('Edit','/tasks/edit/' . $task['Task']['id']);
  endif;
?>
```

This displays the edit link for only the user to whom the task has been assigned.

Now consider the task edit action in Listing 2.

Listing 2. Task edit action

```
function edit($id = null) {
  if(empty($this->data)) {
    if(!$id) {
      $this->Session->setFlash('Invalid id for Task');
      $this->redirect('/task/index');
```

```
    }
    $task = $this->Task->read(null, $id);
    $this->data = $task;
    $this->set('task', $task);
    $this->set('percents', array ('0' => '0%', '25' => '25%', '50' => '50%', '75' =>
'75%', '100' =>
'100%'));
    $this->set('month', $this->buildCalendar($task['Task']['user_id'],
$task['Task']['duedate']));
  } else {
    $this->cleanUpFields();
    if($this->Task->save($this->data)) {
      $this->Session->setFlash('The Task has been saved');
      $this->redirect('/tasks/view/' . $this->data['Task']['id']);
    } else {
      $this->Session->setFlash('Please correct errors below.');
    }
  }
}
```

At no time in this action do you verify that the user who has submitted the data for edit is the user to whom the task has been assigned. It is not enough to simply keep from showing the link to the user. Once the HTML has been sent to the user, it can easily be modified.

While this seems an unlikely case, it is easily reproduced. In your favorite tabbed browser, Firefox, for example, open two tabs to Criki. In one tab, log in as User A and navigate to the edit task page for any task. Then in the second window, log out of Criki and log in again as User B. Return to the tab with the edit task page loaded, and you will find you can successfully edit the task, even though the task belongs to User A, and you are now logged in as User B.

Always verify your data. Always. Twice if you're not certain.

The second half of the Edit action might look more like Listing 3.

Listing 3. Verify that the user is assigned to the task

```
$user = $this->Session->read('User');
$task = $this->Task->read(null, $this->data['Task']['id']);
if ($user['id'] == $task['Task']['user_id']) {
  $this->cleanUpFields();
  if($this->Task->save($this->data)) {
    $this->Session->setFlash('The Task has been saved');
    $this->redirect('/tasks/view/' . $this->data['Task']['id']);
  } else {
    $this->Session->setFlash('Please correct errors below.');
  }
} else {
  $this->Session->setFlash('You cannot modify this task.');
  $this->redirect('/tasks/view/' . $this->data['Task']['id']);
}
```

The second security problem with the task edit action may have been a little harder to spot. You will recall that the primary directive for editing a task was that the user should not be able to edit anything but the percentage complete. Only the percentage field was presented to the user via the task edit view.

But, just as you verified the user, you should verify that the fields the user have

submitted match the fields you are expecting. Or, if you prefer, make sure you only use the data from the fields that you desire (a *whitelisting* approach). Achieving this is fairly simple. You can build a new array of data to be saved, including only those elements from the submitted data you desire. Just remember to format the array correctly.

Listing 4. Whitelisting form fields

```
$user = $this->Session->read('User');
$task = $this->Task->read(null, $this->data['Task']['id']);
if ($user['id'] == $task['Task']['user_id']) {
  $this->cleanUpFields();
  $savedata = array (
    'Task' => array (
      'id' => $this->data['Task']['id'],
      'percent' => $this->data['Task']['percent'],
    )
  );
  if($this->Task->save($savedata)) {
    $this->Session->setFlash('The Task has been saved');
    $this->redirect('/tasks/view/' . $this->data['Task']['id']);
  } else {
    $this->Session->setFlash('Please correct errors below.');
  }
} else {
  $this->Session->setFlash('You cannot modify this task.');
  $this->redirect('/tasks/view/' . $this->data['Task']['id']);
}
```

You can test that this works as you expect it to by adding a field to the task edit view that will allow the user to modify the title of a task.

```
<?php echo $html->input('Task/title', array('size' => '60'));?>
```

Save the view and edit the action, making sure to change the percentage and the title. You should find that only the percentage is modified. This approach of whitelisting form fields is a powerful tool that can help protect you from maliciously inserted form data. Always verify your data. Always. Twice if you're not certain.

Take some time now to look back over the code for Criki. See if you can find other places where you should be verifying your data.

Future versions of CakePHP will automatically apply this form of whitelisting for form fields generated using the form helper.

Reusing the wiki markup rendering code

Using the wiki markup rendering code you already wrote to allow wiki markup in task descriptions could be done a couple different ways.

One way would be to create an action in the entries controller called `renderMarkup`. This would work much like the `buildCalendar` action you wrote for the tasks controller. You could then call it via `requestAction`, passing in the unformatted task description and getting back the task description after the wiki

markup has been rendered.

Another way -- and, perhaps, a better way -- would be to create a custom helper, specifically for rendering wiki markup into HTML, and using this helper in your views. This would be a more versatile and somewhat more elegant approach. To do this, you would create the file app/views/helpers/wiki.php.

Listing 5. Creating the WikiHelper

```
<?php
  class WikiHelper extends Helper {
    function render($content) {
      ...
      ...
      ...
    }
  }
?>
```

The `Render` function is far too long to reproduce here (you need to import it to the EntryRevisions controller), but it is functionally the same, save that `$content` is passed into the function, rather than being pulled from `$this>data`, and a `$return` variable necessary (as the rendered markup is not being set in `$this>data`.)

Once you have created the helper, you need to add the helper to any controller you want to have access to it.

```
var $helpers = array('Html', 'Form', 'Wiki' );
```

Now that the wiki markup rendering code is in a helper, you can significantly reduce the size of the entries view action.

Listing 6. New entries view action

```
function view($title = null) {
  if(!$title) {
    $this->Session->setFlash('Invalid Entry.');
    $this->redirect('/entries/index');
  }
  $entry = $this->Entry->findByTitle($title);
  if ($entry) {
    $user = $this->Session->read('User');
    $user = $this->Entry->User->read(null, $user['id']);
    $this->Session->write('User', $user['User']);
    if ($user['User']['access'] < $entry['Entry']['access']) {
      $this->Session->setFlash('Access Denied.');
      $this->redirect('/entries/index');
    }
    $this->set('entry', $entry);
  } else {
    $this->redirect('/entries/edit/' . preg_replace("/[^a-z]/", , strtolower($title)));
  }
}
```

Now you should be able to use the helper to output data in a view. For example, if

you added *wiki* to the list of helpers for the Entries controller, and if you changed the entries view action to the one above, then in the *entries view* view, instead of using `echo` to output the content, you would use `echo` to output the result of passing the content through the wiki helper.

```
<?php echo $wiki->render($entry['Entry']['content'])?>
```

Save the view and view any entry. You should see no actual change; the entry should be rendered normally, just as before. However, now you have a helper you can use to render wiki markup across the application.

This means you can replace the EntryRevisions view action with one more like the new one you wrote for the Entries controller. All you need to do is include the wiki helper and pass the content through `$wiki render` in the view. The same goes for using wiki markup in the Tasks controller: Include the wiki helper and pass the task description through `$wiki render`. You can even apply the same principle for the open blog you will write in this tutorial.

Adding user profile links

Changing the various task views to output links to user profiles instead of just user names should have been the easiest gap to fill. The code for it has appeared many times in the entry and upload views. It should look something like this:

```
echo $html->link($task['User']['username'], '/users/view/'.$task['Task']['user_id']);
```

The source code for this tutorial has updated task views that contain this code. You should either update your code with the code from the archive or, at the very least, compare the two sets of code to make sure you understand the changes before proceeding with the next task.

Section 3. Creating the open blog

In this section, we will create the open blog, thus allowing any user to post to the Criki blog.

What is an open blog?

To start, don't get too wrapped up in the terminology. It'll help to understand an open blog by specifying what is meant by blog in the first place.

A *blog* is basically a Web site where the content is generated by a user. Generally, the content is displayed with the newest information at the top. Frequently (but not always), readers are able to leave comments about specific blog postings, allowing for discussion.

In an *open blog*, the content is not generated by a user. It's generated by any user. Conceptually, it's much like a cross between a forum -- where many users partake in threaded discussions -- and a blog. By allowing any user to post to the blog, you are further enhancing the open nature of Criki. It should be remembered that this can sometimes be a double-edge sword. Comment spam in blogs is an everyday problem. That problem could become significantly worse when you open up a blog to allow any user to post. If the open blog is external-facing, it will require more regular moderation and babysitting. If the open blog is internal-facing (such as an intranet), this is less of an issue.

Keeping these potential pitfalls in mind, you can jump into the design work for the open blog.

Doing the design

As you did with the other parts of Criki, you should spend some time thinking through how the blog will work before you start writing code or building tables.

For this open blog, some general principles will be adhered to:

- Only registered users will be able to post, though anonymous users will be able to comment.
- Comments and posts will reside in distinct tables.
- Editors can delete posts and comments from contributors. Administrators can delete posts and comments from anyone.
- The information that will be maintained for the blog will be minimal.
- The blog will use the same wiki markup that has already been written.

With these principles in mind, you should have some ideas for how the code will look and behave in the controllers. But don't jump ahead to that yet. You should get the database tables sorted out first.

Building the blog database tables

The open blog will consist of two tables. One table, called posts, will hold blog posts. The other table, called comments, will hold blog comments. One could argue successfully that posts and comments could reside in the same table, as a comment could be described as a post with no parent. But there are merits to keeping posts and comments segregated. For example, you can more easily change post or comment behavior without one affecting the other, you have more granular control

over the way the controllers will behave, and you can more easily apply different sets of security rules.

One of the directives is to keep the information being maintained minimal. Minimal does not imply Spartan. Finding the mix between "The least amount of useful information" and "The most information that can be recorded" is important. The least amount of useful information in a post would be content. Without content, there is no post. But you could go in ever-widening circles trying to catch all the information related to a post.

A minimal posts table would need to record the following:

- Id
- Title
- Content
- Author
- Access
- Date
- Ip

The SQL to create the table might look like this:

Listing 7. Posts table SQL

```
CREATE TABLE 'posts' (
  'id' int(10) NOT NULL auto_increment,
  'title' varchar(255) NOT NULL default ,
  'content' text NOT NULL,
  'user_id' int(10) NOT NULL default '0',
  'access' tinyint(4) NOT NULL default '0',
  'date' datetime NOT NULL default '0000-00-00 00:00:00',
  'ip' varchar(15) NOT NULL default ,
  PRIMARY KEY  ('id')
) ENGINE=MyISAM DEFAULT CHARSET=latin1 AUTO_INCREMENT=1 ;
```

For the comments table, you need a different set of information:

- Id
- Post Id
- Content
- Author
- Access
- Date
- Ip

The SQL to create this table might look like this:

Listing 8. Comments table SQL

```
CREATE TABLE 'comments' (
  'id' int(10) NOT NULL auto_increment,
  'post_id' int(10) NOT NULL default '0',
  'content' text NOT NULL,
  'user_id' int(10) NOT NULL default '0',
  'access' tinyint(4) NOT NULL default '0',
  'date' datetime NOT NULL default '0000-00-00 00:00:00',
  'ip' varchar(15) NOT NULL default ,
  PRIMARY KEY  ('id')
) ENGINE=MyISAM DEFAULT CHARSET=latin1 AUTO_INCREMENT=1 ;
```

In both cases, you will notice the IP address is being recorded. For posts, this can be helpful for tracking problem users, but it's much more important for comments, where you will be allowing anonymous comments. What you may notice is the absence of tracking things like last modified, last read, hits, or comment count. Keep the application simple where you can. The extra data generally won't be necessary.

Go ahead and create the tables, if you haven't already. Next, you write the models, bake the controllers and views, and dive in to building out your open blog.

Section 4. Basic blog functionality

By now, you've written a few models, and you should be pretty familiar with Bake. If you feel ambitious, you can write the controllers out by hand and skip the Bake section below. But you should at least read through the models information provided because the model associations for your blog tables are important.

Writing the models

You will need to define two models -- one for posts and one for comments. Setting up the associations for these models will be important and slightly differently model associations from what you've done before.

The post model

The post model will have two associations. It will have a `belongsTo` association to user, just like the entry and task models. But the post model will also have a `hasMany` association to comment. The code for the model might look like this:

Listing 9. Post model

```
<?php
class Post extends AppModel {
```

```
        var $name = 'Post';

        var $belongsTo = array('User' => array (
          'className' => 'User',
          'conditions' => ,
          'order' => ,
          'foreignKey' => 'user_id'
        )
      );

      var $hasMany = array('Comment' => array (
        'className' => 'Comment',
        'conditions' => ,
        'order' => 'Comment.date DESC',
        'foreignKey' => 'post_id'
      )
    );
}
?>
```

By setting up this `hasMany` association, you are able to get the post and comments all at once. The order key defined above will sort the comments for you.

That takes care of the post model. The comment model looks a little different from what you might expect.

The comment model

It seems like the comment model would have two associations. A comment belongs to a user in the sense that the user was the author, and a comment belongs to a post, as the comment is tied to the post by ID. However, specifying the post association is not necessary in this context, as comments will not be retrieved outside the context of their parent posts, with the exception of editing comments, which does not require the post information. By not specifying the post association, you will save some query overhead.

Listing 10. Comment model

```
<?php
class Comment extends AppModel {

    var $name = 'Comment';

    var $belongsTo = array('User' => array (
      'className' => 'User',
      'conditions' => ,
      'order' => ,
      'foreignKey' => 'user_id'
      )
  );

}
?>
```

Now that you've got your models sorted out, you can start baking.

Baking the basics

Since you used Bake in Parts 1 and 4 of this series, you don't need to be reminded that you need to make sure the PHP executable is in your `PATH` and that you have changed into the directory where you installed Cake.

So, run bake, as shown below.

```
php cake\scripts\bake.php
```

Walk through the menus baking the controllers for post and comment, then baking the views for post and comment. If you need a refresher on the Bake menus, consult Part 1 of this series (see Resources). Once everything is baked, you can jump right in to customizing the code.

Customizing the code

There are several things you need to change about the controller and view code that you just baked. A list of the principal changes will help you to understand the scope:

- Comment and post `add` actions need to set the value of date.
- Comment and post `add` and `edit` actions need to restrict the directly editable fields.
- Comment and post add and edit views need to display only editable fields.
- Comment `view` action and view can be removed.
- Comment `add`, `edit` and `delete` actions should redirect the user to the parent post.
- Post `view` action needs to be modified to pull comments from the model.
- Post *view* view need to be modified to display comments well.
- Access checks should be made for post and comment `edit` and `delete` actions.

The code for each of these changes will be highlighted. Full details can be found in the source code.

Setting date values

Setting the value of the date fields in the posts or comments controller isn't difficult. Before the data is saved, you would need to set the date manually, for example, in the comments controller.

```
$this->data['Comment']['date'] = date('Y-m-d H:i:s');
if($this->Comment->save($this->data)) {
...
```

While you're already working with the fields, you may as well restrict the once that can be edited.

Restricting editable fields

Restricting the fields which the user can edit means more than just removing the fields from the related views -- though that's an excellent place to start. Actually, when you look at a post, the only fields a user should be able to edit are title and content. For a comment, a user should only be able to edit the content field. Everything else will be set by the controllers.

Go through the add and edit views for posts and comments, and remove all but the fields mentioned above. You can also take the opportunity to remove the links to controllers or actions you don't need, such as the add user link in the comments edit view.

But, as pointed out in the Criki so far section of this tutorial, removing the fields from the views is not enough to ensure that the user cannot modify the data. You should build a new array to hold the data to be saved and explicitly set the values for your fields. Building such an array for the posts controller will look like Listing 11.

Listing 11. Building the data array for posts

```
$savedata = array(
  'Post' => array (
    'title' => $this->data['Post']['title'],
    'content' => $this->data['Post']['content'],
    'user_id' => $user['id'],
    'access' => $user['access'],
    'date' => date('Y-m-d H:i:s'),
    'ip' => $_SERVER['REMOTE_ADDR'],
  )
);
```

Then, rather than saving `$this>data`, you would instead save `$savedata`. By following this method, you reduce the potential for a malicious user to force field editing you had not permitted.

Redirecting to the parent post

Within the comments controller, anytime a comment is added, edited or deleted, redirection should send the user back to the view action for the parent post. (You can delete the comments index and view actions, as the will not be used.)

The actual process of redirecting the user to the parent post is simple, once you have the data you need (namely, the `post_id` for the comment). You already have to modify the comment add action, as you will be passing it the ID of the parent post, which means for Add, you will have the data you need. For the edit and delete actions, while you could pass the `post_id` in as a hidden form variable; it will be more secure to query for the comment by ID and read the `post_id` from the database. The user cannot edit `post_id`, so it will never change. For example, the

code to do this in the comment delete action looks like Listing 12.

Listing 12. Comment delete action code snippet

```
...
$comment = $this->Comment->findById($this->data['Comment']['id']);
if ($comment) {
  if($this->Comment->del($id)) {
    $this->Session->setFlash('The Comment deleted: id '.$id.);
    $this->redirect('/posts/view/' . $comment['Comment']['post_id']);
  }
} else {
  $this->Session->setFlash('Invalid id for Comment');
  $this->redirect('/posts/index');
}
...
```

Take a look at all the redirect statements in the comments controller found in the source code for more changes of this nature.

Retrieving comments from within the posts controller

For the posts view action, retrieving the post and the related comments will be two separate model requests. The first will retrieve the post and the user information.

```
$post = $this->Post->read(null, $id);
```

This request, if the model associations are set up properly, will actually get the post data, the user data for the post author, and the comments for the post. But the comments will not include the user data for each comment. Rather than open up a recursive can of worms, it's much simpler to just make an additional request.

The second model request will retrieve the comments and comment user data for the post.

Listing 13. Post view action code to retrieve comments data

```
$post['Comment'] = $this->Post->Comment->findAll(
  'Comment.post_id = ' . $id . ' AND User.id = Comment.user_id',
  array('Comment.*', 'User.*'),
  'Comment.date DESC'
);
```

This `findAll` request is a little more complicated than `findByFIELDNAME`, but you are also exercising a bit more control over things like sort order. Sorting the comments in descending order by date will mean less scrolling for the end user. If you prefer to have the comments displayed in ascending order by date, you could use `findByPostIt` and save users a little typing.

If you look at the posts index action, you will see it also is passing additional data to `findAll` to sort the posts in descending order by date.

Displaying comments

Last, but not least, you will need to modify the*posts* view view. Not only do you need to account for the new data structure but the comments should be reformatted into something more readable. Fully reproducing the code here is impractical, but look at Listing 14 to see how the comment iteration code looks.

Listing 14. Posts view view amendment

```
<?php if(!empty($post['Comment'])):?>
<div class="related">
<h2>Comments</h2>
<?php foreach($post['Comment'] as $comment):?>
<hr />
<h4>
By <?php echo $html->link($comment['User']['username'], '/users/view/'
.$comment['User']['id'])?>
on <?php  echo $comment['Comment']['date']?>
[
<? php echo $html->link('Edit Comment', '/comments/edit/' . $comment['Comment']['id']) ?>
|
<?php echo $html->link('Delete Comment', '/comments/delete/' . $comment['Comment']['id'],
null,  'Are you sure  you want to delete this comment?')
?>
]
</h4>
<br />
<p>
<?php echo $wiki->render($comment['Comment']['content']) ?>
</p>
<?php endforeach; ?>
<hr />
</div>
<?php endif; ?>
```

Check the posts index view in the source code for more.

Access checks

As you have done with the tasks and entries parts of Criki, you will need to implement some access control within blog. Specifically, the following principles will apply:

- A user must be logged in to post.
- A user need not be logged in to comment.
- A user may always edit or delete his own posts or comments.
- Any user with an access level the access level of the post or comment may perform delete actions.

An example will be provided below and in the code of applying these principles to the post edit action. From there, you should be able to apply access checks to the comment edit action, as well as the post and comment delete actions.

The process is should be a familiar one. You have already amended the edit views to only show the fields that can be edited. You will need to amend the posts edit action to perform the access check. You will need to amend the posts index and view actions to set variables used by the views to show/hide links. You will need to amend the posts index and view views to take advantage of these variables.

Amending the posts edit action

You will recall that the user data is already being pulled from session during the posts edit action.

Listing 15. Posts edit action revisited

```
$user = $this->Session->read('User');
  $savedata = array(
    'Post' => array (
    'id' => $this->data['Post']['id'],
    'title' => $this->data['Post']['title'],
    'content' => $this->data['Post']['content'],
    'user_id' => $user['id'],
    'access' => $user['access'],
    'ip' => $_SERVER['REMOTE_ADDR'],
  )
);
```

Performing the access check for the edit action will mean pulling the post information from the database and checking for access. In this case, the user may only edit the post if the user was the author. If you were checking a delete action, you would also pull the user information from the database and check to make sure the user access was greater than the post access.

Listing 16. Post edit action revised

```
$post = $this->Post->findById($this->data['Post']['id']);
if ($post['Post']['user_id'] == $user['id']) {
  $savedata = array(
    'Post' => array (
      'id' => $this->data['Post']['id'],
      'title' => $this->data['Post']['title'],
      'content' => $this->data['Post']['content'],
      'user_id' => $user['id'],
      'access' => $user['access'],
      'ip' => $_SERVER['REMOTE_ADDR'],
    )
  );
  if($this->Post->save($savedata)) {
    $this->Session->setFlash('The Post has been saved');
    $this->redirect('/posts/index');
  } else {
    $this->Session->setFlash('Please correct errors below.');
    $this->set('users', $this->Post->User->generateList());
  }
} else {
  $this->Session->setFlash('You cannot edit this post.');
  $this->redirect('/posts/index');
}
```

This will protect the post from being edited by anyone but the author. Now you

should make sure you don't show edit links to anyone but the author.

Amending the posts index and view actions

To allow the index and view actions to make decisions about showing or not showing the edit post link, you need to tell the posts controller to set a variable that the index and view views can use. The easiest way to do this is to retrieve the user data and set it as a variable. To do this in the index action, see Listing 17.

Listing 17. Index action setting user data

```
function index() {
  $this->Post->recursive = 0;
  $this->set('posts', $this->Post->findAll(
    'Post.user_id = User.id AND Post.id',
    array('Post.*', 'User.*'),
    'Post.date DESC'
  ));
  $this->set('user', $this->Session->read('User'));
}
```

The posts index view can access the variable `$user` and make decisions based on the user data about showing or hiding the edit post links.

Amending the posts index and view views

You've got the user data into your view. Using it to show or hide the edit post link is as easy as, well, Cake. See the posts index view below.

Listing 18. Conditionally showing the edit post link

```
<?php
if ($user['id'] == $post['User']['id']) {
  echo $html->link('Edit','/posts/edit/' . $post['Post']['id']);
}
?>
```

The same code for the *posts view* view will look only marginally different from this, as indicated by the code in the source code.

That's really all there is to it. Take it for a spin. Log in as different users and create some posts, testing out the access controls you've put in place to protect the posts from being edited. After you've done that, apply the same principles and add access controls to the posts and comments controller for delete actions (users with higher clearance or original author), comment edit actions (original author), post add (registered and logged in user), etc. Keep in mind the lessons you've learned, and you should be in good shape.

Section 5. Configuration options

In Part 1, you created a table called settings, which was designed as a sort of Entity-Attribute-Value table to hold specific configuration settings, allowing the administrator granular control over specifics like "Who can create a new Entry" and "Valid file upload types." Now that you have your primary Criki features in place, you can turn your attention to putting some of these settings into place.

Gathering all of these configuration settings into a list will give you an idea for what is in play:

- Editor promotion/demotion rights
- Registration rights
- Invitation rights
- Page creation permissions (who can create a page?)
- Page edit permissions
- Valid file upload types

You will walk through implementing one of the configuration options: page creation permissions. From there, you should be able to set up any individual settings and add new ones.

Tweaking the settings controller

Depending on the version of Cake you used to bake your settings controller, you may need to go through the controller and verify that all of the `redirect` statements point to the settings controller (plural), not the setting controller. The `redirect` statements should look like the code below.

```
$this->redirect('/settings/index');
```

You will want to come back to this controller later to add access controls, so only administrators can change or add settings.

Adding a setting

Adding a new setting is fairly simple. You can later decide to change the views or add data validation to facilitate or streamline the process, but the views as they were baked are sufficient.

The add setting view contains four fields: Name, Value, Default, and Description.

The Name field will be used to reference the setting in the code. It should be a single word with no spaces, such as `entryCreatePermission`.

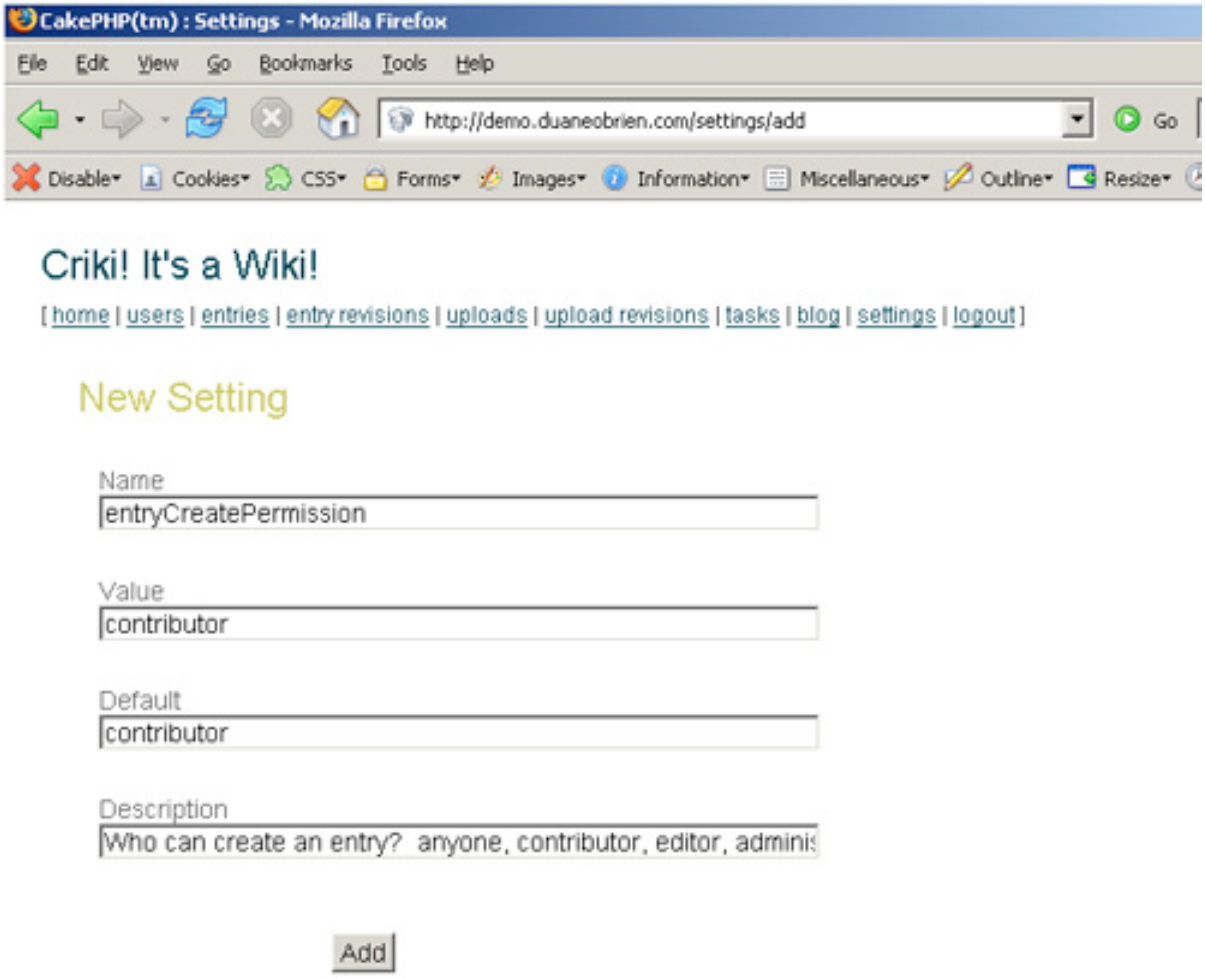
The Value field holds the value for the setting. You determine what is and isn't a valid value, but you will probably find it helpful in the long term to make the values easy to comprehend. In this case, the `entryCreatePermission` setting will be used to identify the minimum user type that can create an entry. Valid values for this field might be anyone -- contributors, editors, administrators.

The Default field is used to maintain the original value for the setting, so an administrator has the capacity to restore the defaults.

The Description field should be used to sum up the purpose of the field and list the valid values for the setting.

Adding the `entryCreatePermission` setting might look like this:

Figure 1. Adding the entryCreatePermission setting



Retrieving and caching the settings

Retrieving the settings is easy. You can simply add the setting model to the `$uses`

class variable much like you did with the EntryRevisions and UploadRevisions controllers. Add the setting model to the entries controller.

Listing 19. Adding the setting model to the entries controller

```
$settings = cache(FILENAME);
if (!$settings) {
  $settings = $this->Entry->Setting->findAll();
  cache(FILENAME, $settings);
}
```

It wouldn't be efficient to keep querying the database to get the settings every time you want to access them. Instead, what you'll do is add a method to the setting model to cache the settings data.

Adding the cacheSettings method to the model

The CakePHP `cache` function serves two purposes. If you pass in only a filename, such as `cache('settings')`, Cake will check the Cake tmp directory for the file, and, if found, the `cache` function will use `file_get_contents` to return the contents of the file.

If you pass a filename and a second variable that holds data, such as `cache('settings', $anArray)`, Cake will use `file_put_contents` to write the data to the filename specified within Cake's tmp directory. The default life of the cache would be one day. You can pass in a third parameter, consisting of any date representation that can be interpreted by `strtotime`.

The method you will add to the setting model will be called `cacheSettings`. Like the `cache` function, this method will return the settings from the cache, or if a `$data` parameter is set, this method will write the data to the cache. You can only pass a string to `put_file_contents`, so you need to flatten or serialize the array first.

When you retrieve the data from the cache, it may be helpful to reformat the settings array into something more shallow, as well.

The cacheSettings method might look something like Listing 20.

Listing 20. cacheSettings method in the settings model

```
function cacheSettings($data = null) {
  if ($data != null) {
    cache('settings', serialize($data));
  } else {
    $data = cache('settings');
    if ($data == null) {
      $data = $this->findAll();
      cache('settings', serialize($data));
    } else {
      $data = unserialize($data);
    }
  }
  foreach ($data as $key => $value) {
    $return[$value['Setting']['name']] = $value['Setting']['value'];
```

```
  }
  return $return;
}
```

Using this method, you can save unnecessary database calls to get the value of an individual setting. Now that you can get the settings, you can put them to work.

Using the page creation setting

Retrieving and using this setting involves calling the `cacheSettings` method and applying some logic to the settings. For example, in the entries controller, if you wanted to verify the `entryCreatePermission` value allowed for anyone to create a new entry, you might do something like Listing 21.

Listing 21. Checking against the entryCreatePermission setting

```
$user_id = 0;
if ($this->Session->check('User')) {
  $user = $this->Session->read('User');
  $user_id = $user['id'];
}
$settings = $this->Setting->cacheSettings();
if ($settings['entryCreatePermission'] != 'anyone' && $user_id == 0) {
  $this->Session->setFlash('You must be a Contributor to create a new entry.');
  $this->redirect('/entries/index');
  exit();
}
```

This is a fairly basic overview of how to get and use your settings. How you decide to implement them is entirely up to you. You have a whole list of settings you can use for practice.

Filling in the gaps

You've gotten a lot done. But there's still tons of room for improvement within the Criki application. For example:

1. The access controls for the open blog in the code archive are incomplete. You still need to add access controls that protect the delete actions on the posts and comments controllers, and an access control is needed to ensure that only logged-in users can post. Build out these access controls.
2. There is a lot of settings and configuration work that you can put to use. Try implementing a few and see how you do.
3. The settings controller is entirely unprotected. You should really do something about that.
4. New settings won't make it into the cache until the existing cache expires.

You should have the settings controller recache the settings whenever one is added, edited, or deleted.

Happy coding!

Section 6. Summary

You did it! You've completed this "Create an interactive production wiki using PHP" tutorial series and built your own wiki engine from scratch. If it feels like you've just scratched the surface, that's because you have loads of room to add new things to Criki.

Or, you may feel like tearing it all back down and building a bigger and better wiki engine than Criki could ever be. Go for it! Tear it all down and build it back up again your own way. The experience will teach you much.

But maybe that's not the case. Maybe you'd rather build on the work you've already done. That's great! Make Criki better. Build out the features of your dreams.

Downloads

Description	Name	Size	Download method
Part 5 source code	os-php-wiki5.source.zip	32KB	HTTP

Information about download methods

Resources

Learn

- Read Part 1, Part 2, Part 3, and Part 4 of this "Create an interactive production wiki using PHP" series.
- Check out the Wikipedia entry for wiki.
- Check out WikiWikiWeb for a good discussion about wikis.
- Visit the official home of CakePHP.
- Check out the "Cook up Web sites fast with CakePHP" tutorial series for a good place to get started.
- 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.
- Find out more about how PHP handles sessions.
- Check out the official PHP documentation.
- Read the five-part "Mastering Ajax" series on developerWorks for a comprehensive overview of Ajax.
- Check out the "Considering Ajax" series to learn what developers need to know before using Ajax techniques when creating a Web site.
- CakePHP Data Validation uses PHP Perl-compatible regular expressions.
- See a tutorial on "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 the Model-View-Controller on Wikipedia.
- 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.
- PHP.net is the 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.

- 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.

Get products and technologies

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

Discuss

- Participate in developerWorks blogs and get involved in the developerWorks community.
- Participate in the developerWorks PHP Developer Forum.

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.