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.

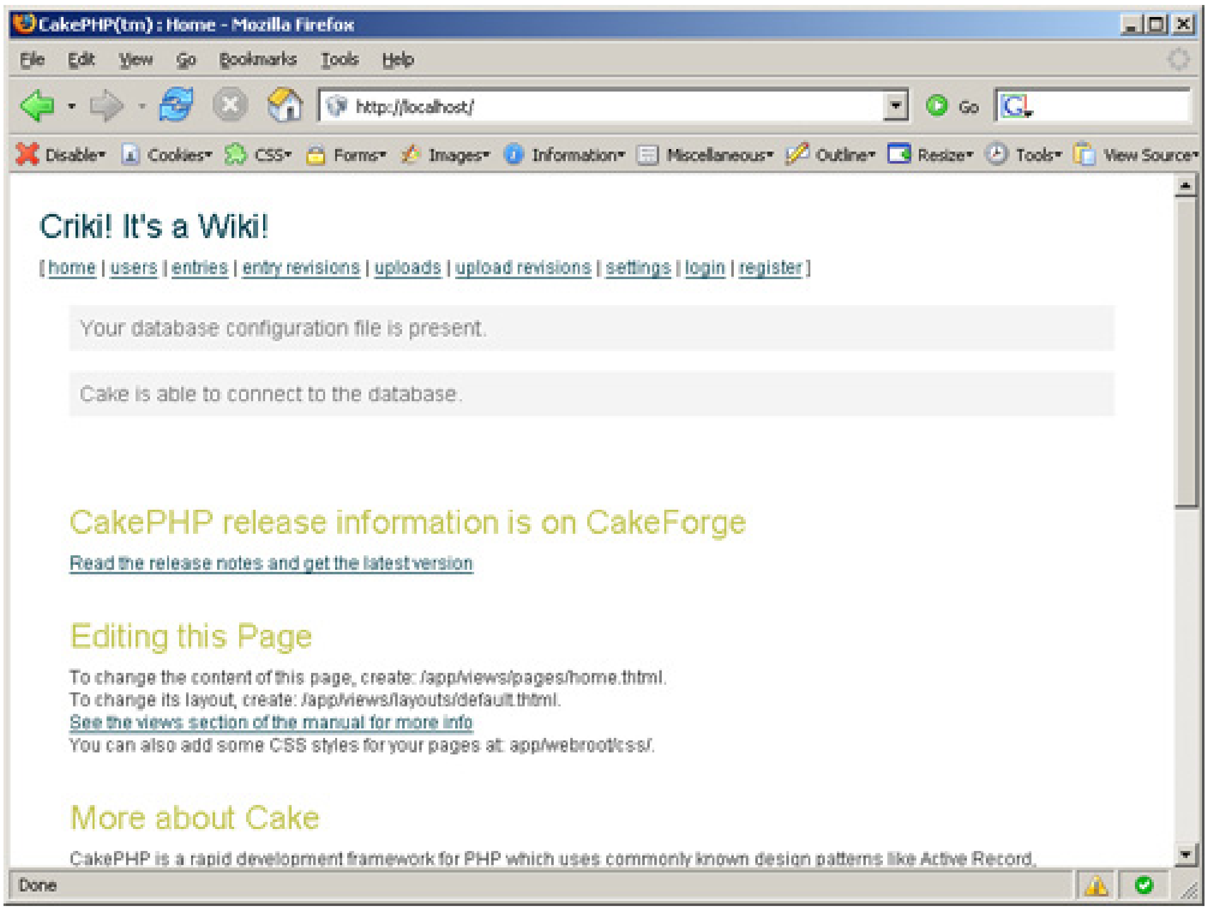
You'll also need a database and database user ready for your application to use. The tutorial will provide syntax for creating any necessary tables in MySQL.

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


CakePHP(tm) : Home - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost/
Go
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tools View Source
Done

Criki! It's a Wiki!

[home | users | entries | entry revisions | uploads | upload revisions | settings | login | register]

Your database configuration file is present.

Cake is able to connect to the database.

CakePHP release information is on CakeForge

Read the release notes and get the latest version

Editing this Page

To change the content of this page, create: /app/views/pages/home.thtml.
To change its layout, create: /app/views/layouts/default.thtml.
See the views section of the manual for more info
You can also add some CSS styles for your pages at: app/webroot/css/.

More about Cake

CakePHP is a rapid development framework for PHP which uses commonly known design patterns like Active Record, 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

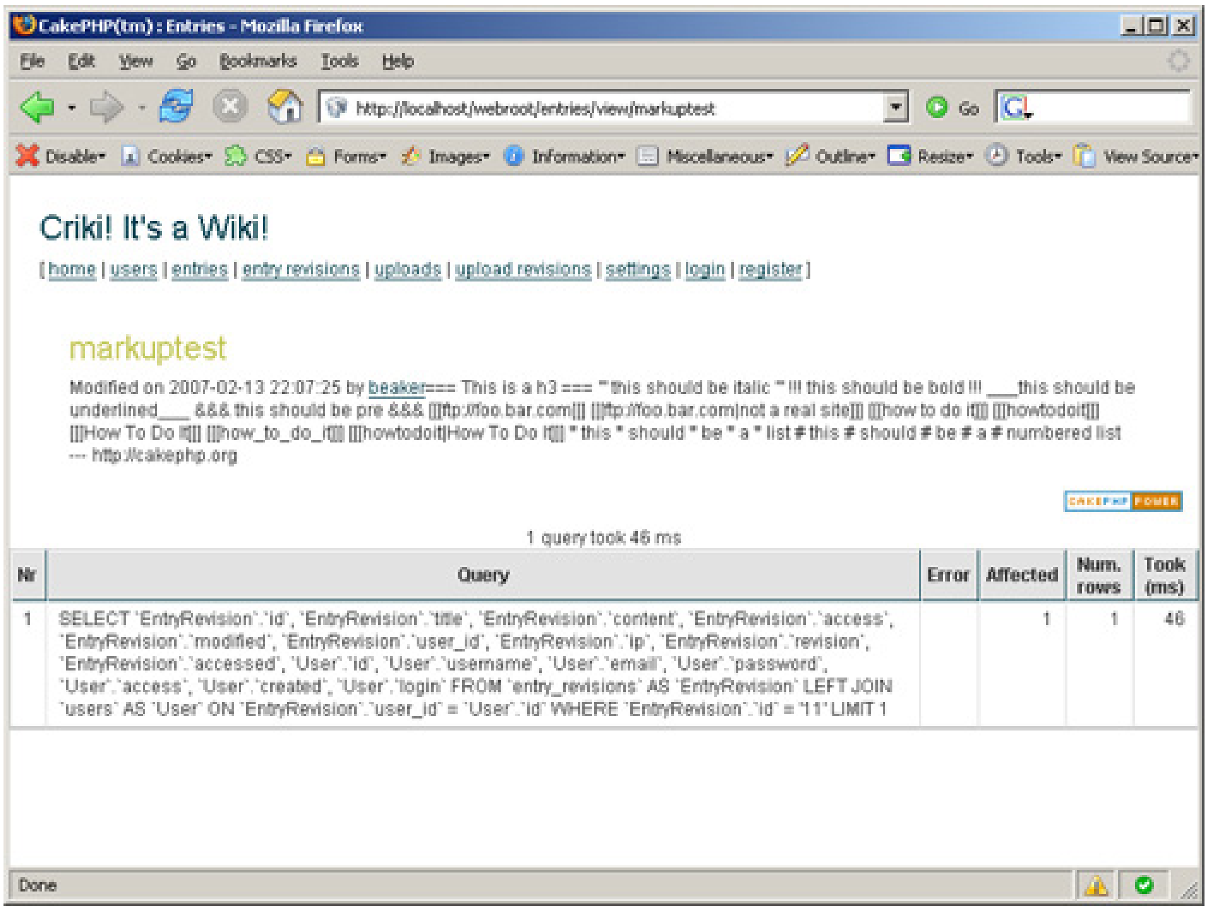
When testing the markup rendering, it will be helpful to have a test entry you can view, which contains an example of each markup. Use the following text for this purpose.

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


CakePHP(tm) : Entries - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost/webroot/entries/view/markuptest
Go
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tools View Source
CAKEPHP POWER
Done

Criki! It's a Wiki!

[home | users | entries | entry revisions | uploads | upload revisions | settings | login | register]

markuptest

Modified on 2007-02-13 22:07:25 by beaker=== 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

1 query took 46 ms

Nr	Query	Error	Affected	Num. rows	Took (ms)
1	SELECT `EntryRevision`.`id`, `EntryRevision`.`title`, `EntryRevision`.`content`, `EntryRevision`.`access`, `EntryRevision`.`modified`, `EntryRevision`.`user_id`, `EntryRevision`.`ip`, `EntryRevision`.`revision`, `EntryRevision`.`accessed`, `User`.`id`, `User`.`username`, `User`.`email`, `User`.`password`, `User`.`access`, `User`.`created`, `User`.`login` FROM `entry_revisions` AS `EntryRevision` LEFT JOIN `users` AS `User` ON `EntryRevision`.`user_id` = `User`.`id` WHERE `EntryRevision`.`id` = '11' LIMIT 1		1	1	46

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

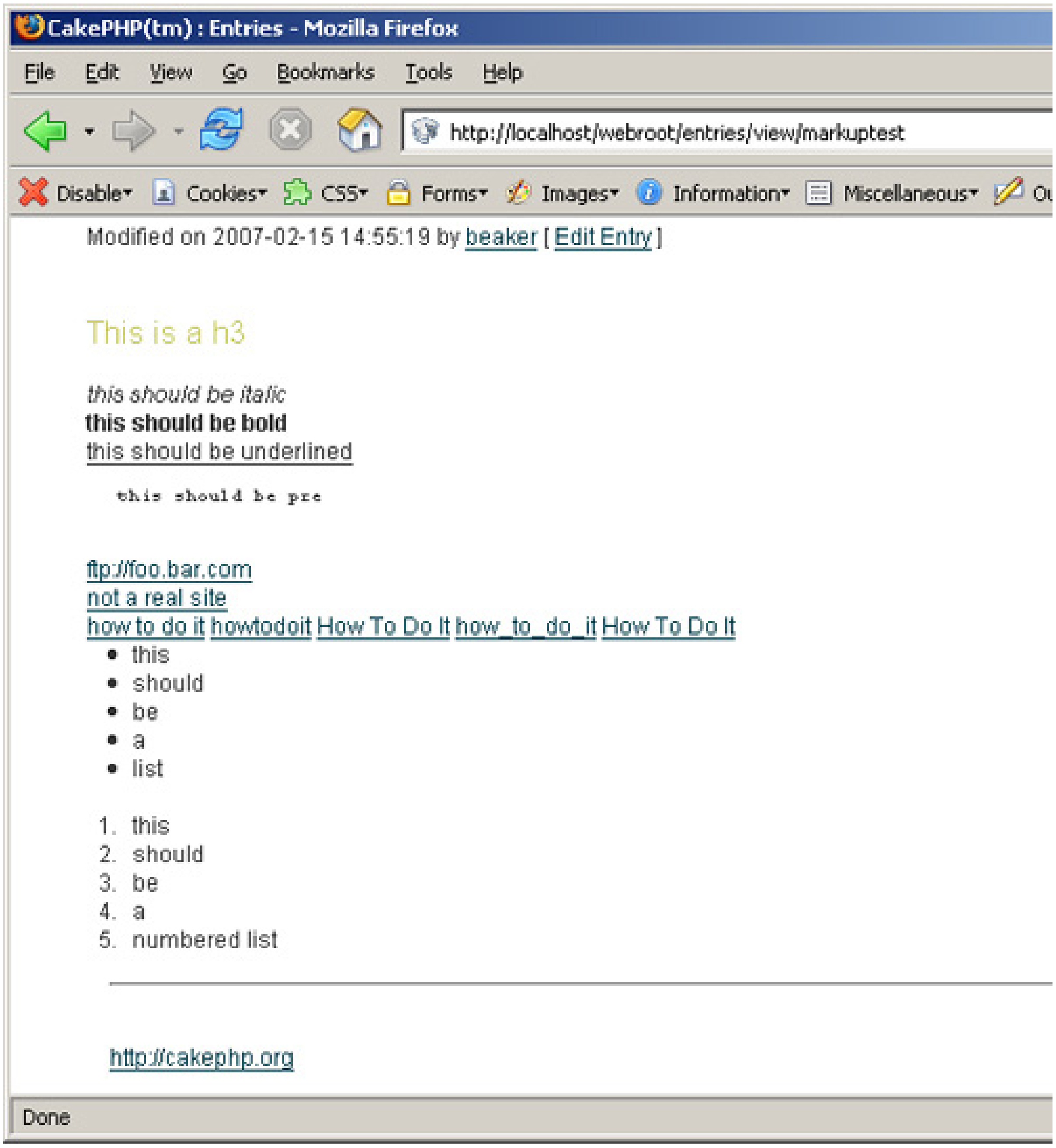
```
      $word .= "</a>";
      $processing["[[["] = false;
      $processing["]]]"] = false;
    }
    break;
```

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


CakePHP(tm) : Entries - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost/webroot/entries/view/markuptest
Disable Cookies CSS Forms Images Information Miscellaneous
Done

Modified on 2007-02-15 14:55:19 by beaker [Edit Entry]

This is a h3

this should be italic
this should be bold
this should be underlined

```
this should be pre
```

ftp://foo.bar.com
not a real site
how to do it howtodoit How To Do It how_to_do_it How To Do It

- this
- should
- be
- a
- list

1. this
2. should
3. be
4. a
5. numbered list

http://cakephp.org

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	19KB	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.