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).

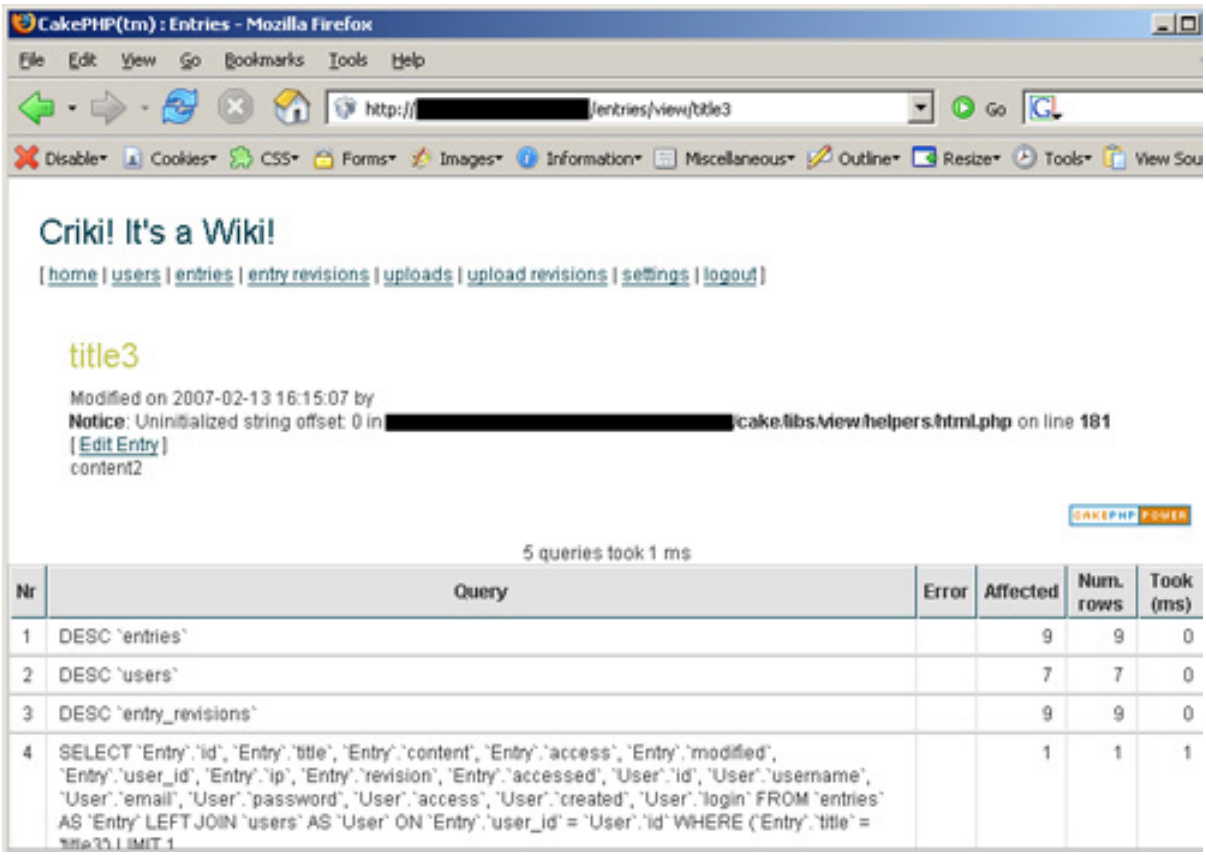
Section 2. Criki so far

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'])?>
```

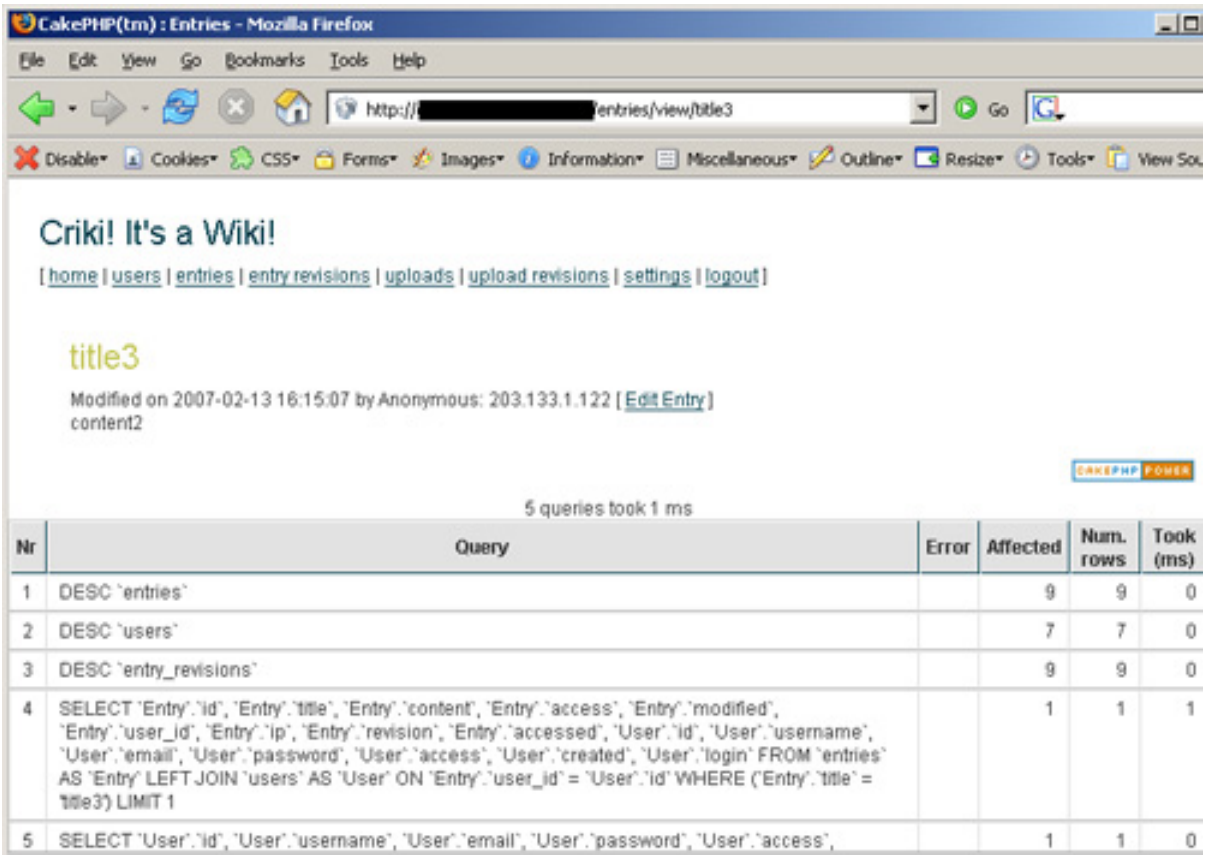
The best way to address this is to verify that the username is set and display the IP address if it's not.

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.