Learning PHP, Part 3

Presented by developerWorks, your source for great tutorials

`ibm.com/developerWorks`

Table of contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Before you start

About this tutorial

This tutorial finishes the simple workflow application you began in the first part of this series about learning PHP. You will add HTTP authentication, the ability to stream documents from a non-Web-accessible location, and exception handling. You'll also organize some of the application into objects.

Overall, you will add the ability for an administrator to approve a file, making it generally available to users. Along the way, the following topics will be discussed:

- Enabling and using browser-based HTTP authentication
- Streaming data from a file
- Creating classes and objects
- Using object methods and properties
- Creating and handling exceptions
- Using XML ID attributes
- Validating an XML document using a Document Type Definition (DTD)
- Controlling access to data based on the requesting page

Who should take this tutorial?

This tutorial is Part 3 of a three-part series designed to teach you the basics of programming in PHP while building a simple workflow application. It is for developers who want to learn more about advanced topics, such as using PHP for object-oriented programming. This tutorial also touches on HTTP authentication, streaming, classes and objects, and exception handling, as well as provides another look at manipulating XML.

This tutorial assumes familiarity with the basic concepts of PHP, such as syntax, form handling, and accessing a database. You can get all the information you will need by taking "*Learning PHP, Part 1*" and "*Learning PHP, Part 2*," and by checking the Resources on page45 .

Prerequisites

To follow along with the sample code, you need to be sure the following tools are installed and tested:

HTTP server -- You can install PHP on a variety of HTTP servers, such as Apache and Microsoft IIS, and on Windows, Linux, UNIX, Mac OSX, and other platforms. In general, your choice of server doesn't matter, but this tutorial will cover some configurational issues regarding HTTP authentication using Apache 2.X as an example. You can download the Apache HTTP server from *Apache* (http://httpd.apache.org/download.cgi) .

PHP -- Of course, you will also need a distribution of PHP. Both PHP V4 and V5 are in use at the time of this writing, but this tutorial concentrates on V5 because of its enhancements. *Download PHP* (http://us4.php.net/downloads.php) .

Database -- Part of this project involves saving data to a database, so of course you'll need one of those, as well. This tutorial covers MySQL because it's so commonly used with PHP. You can download MySQL from *http://dev.mysql.com/downloads/index.html*.

About the authors

Tyler Anderson graduated with a degree in computer science from Brigham Young University in 2004 and is currently in his last year as a master's student studying computer engineering. In the past, he has worked as a database programmer for DPMG.COM, and he is currently an engineer for Stexar Corp. in Beaverton, Ore. He can be reached at *tyleranderson5@yahoo.com*.

Nicholas Chase (http://www.chaosmagnet.com?ibm) has been involved in Web-site development for companies such as Lucent Technologies, Sun Microsystems, Oracle, and the Tampa Bay Buccaneers. He has been a high school physics teacher, a low-level-radioactive-waste facility manager, an online science fiction magazine editor, a multimedia engineer, an Oracle instructor, and the chief technology officer of an interactive communications company. He is the author of several books, including *XML Primer Plus* (Sams, 2002). He loves to hear from readers and can be reached at *ibmquestions@nicholaschase.com*.

Section 2. The story so far

Where things stand right now

You've been building a simple workflow application through the course of these tutorials. The application enables users to upload files to the system and to see those files, as well as files approved by an administrator. So far, you've built:

- A registration page that enables a user to use an HTML form to sign up for an account by entering a unique username, e-mail address, and password. You built the PHP page that analyzes the submitted data, checks the database to make sure the username is unique, and saves the registration in the database.
- A login page that takes a username and password, checks them against the database, and, if they're valid, creates a session on the server so the server knows which files to display.
- Simple interface elements that detect whether the user is logged in to display appropriate choices.
- An upload page that enables users to send a file to the server via a browser. You also built the page that takes this uploaded file and saves it to the server, then adds information about it to an XML file for later retrieval, using the Document Object Model (DOM).
- A display function that reads the XML file and displays the information using the Simple API for XML (SAX).

You can download the files that represent where the application left off in "*Learning PHP, Part 2*."

What you're going to do

Before you're through with this tutorial, you'll have a complete -- though extremely simple, of course -- workflow application. In this tutorial, you will:

- Add HTTP authentication, controlled by the Web server. You'll also integrate your registration process so it adds new users to the Web server.
- Add links to the function that displays the available files so users can download them. You'll create a function that streams these files to the browser from the non-Web-accessible location.
- Confirm that users download files from the appropriate page. You'll use the fact that files must be streamed by the application, instead of simply served by the HTTP server, to enable control over the circumstances in which users download files.
- Create a class that represents a document, and use object-oriented methods to access and download it.
- Create and use custom exceptions to help pinpoint problems.

- Add information to the XML file that will enable you to uniquely identify each file. This step will enable you to add check boxes to the display form that administrators can use to determine which files to approve.
- Manage the approval process, adjusting your XML file so you can request specific `fileInfo` elements directly.

Let's start by putting a public face on what you already have.

The welcome page

Up to now, you've concentrated on building the individual pieces of your application. Now it's time to start putting them together, so start with a simple welcome page you can use as a "landing strip" for visitors. Create a new file called *index.php* and add the following:

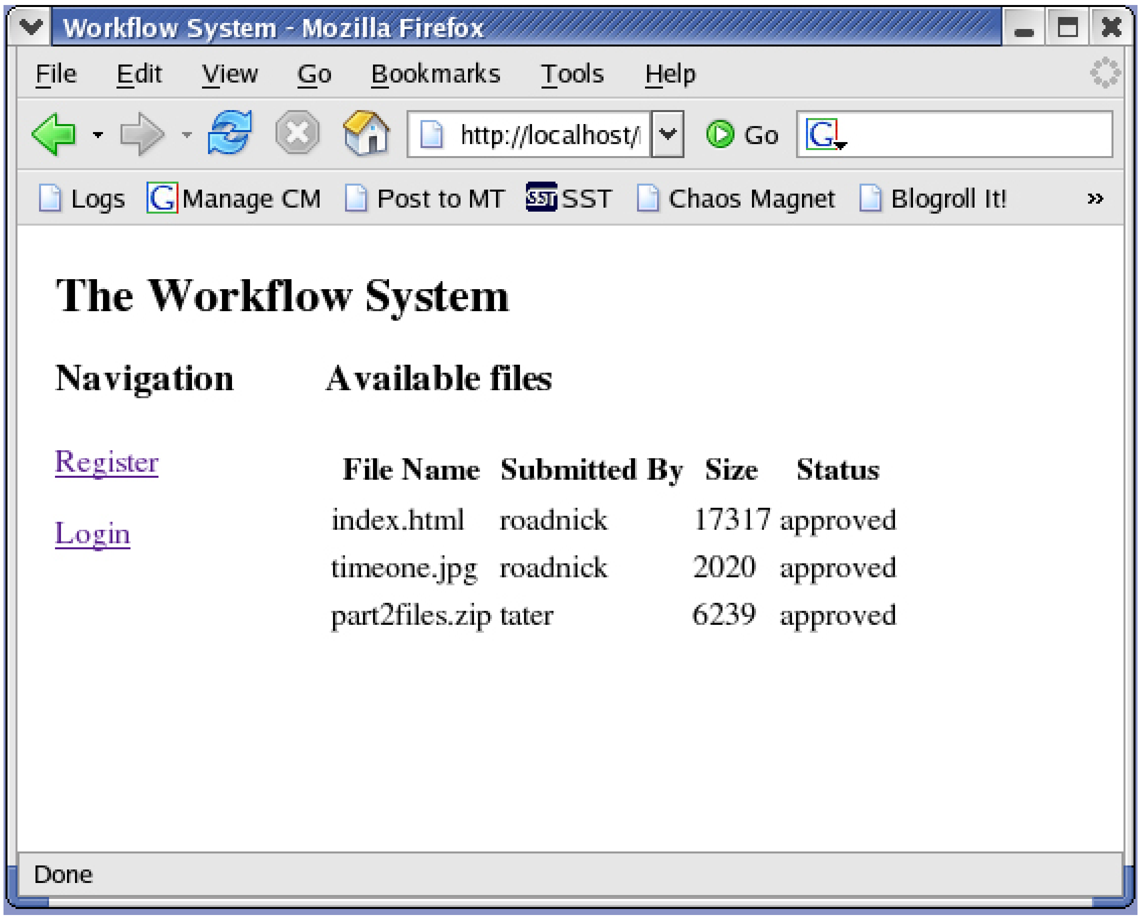
```
<?php

   include ("top.txt");
   include ("scripts.txt");

   display_files();

   include ("bottom.txt");

?>
```

The first `include()` function loads the top interface elements for the page and sets up a session, if applicable. The second loads all the scripts you've created so far, including the `display_files()` function you created in "*Learning PHP, Part 2*," which lists all the files uploaded by the current user or approved by an administrator. The final include is just the bottom of the HTML page.

Save the file in the same directory the other files you've created are in. For example, you could put the file in the document root of your server, so once you've started the HTTP server, you could see the page by pointing your browser to http://localhost/index.php.

As you can see in Figure 1, the page is pretty simple, and, in fact, if you've been following along, you shouldn't see *any* files unless you're logged in because none have been approved yet. The files here have been approved for demonstration purposes.

Figure 1. The basic listing page


Workflow System - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost/
Go
Logs Manage CM Post to MT SST Chaos Magnet Blogroll It!
Done

The Workflow System

Navigation

Register

Login

Available files

File Name	Submitted By	Size	Status
index.html	roadnick	17317	approved
timeone.jpg	roadnick	2020	approved
part2files.zip	tater	6239	approved

If you've just started up your browser, you should see the **Register** and **Login** links because you're not logged in. In the next section, you'll look at another way to handle that process.

Section 3. Using HTTP authentication

HTTP authentication

Up to now, you've used a login system in which the user enters a username and password into a form, and when the user submits the form, that information is checked against the MySQL database. If it matches, the application creates a session within PHP and assigns a username to the `$_SESSION` array for later use.

While this process works just fine, you run into a problem when you integrate with other systems. For example, if your workflow application was part of an intranet in which users might be logging in with usernames from other systems, you may not want to require them to log in again. Instead, you want them to already be logged in when they get there, if they've already logged in elsewhere. This is known as a *single sign-on* system.

To accomplish that here, you're going to switch over to a system in which the Web server actually controls the login process. Instead of simply serving the page, the server checks for a username and password within the request from the browser, and if it doesn't see them, it tells the browser to pop up a username and password box so you can enter that information. Once you enter the information, you won't have to do it again because the browser sends it with subsequent requests.

Let's start by setting up the server.

Enabling HTTP authentication

Before you get started, be aware that if you use a server other than Apache 2.X, you should check the documentation for HTTP authentication to see what you need to do to set it up. (Alternatively, you can simply skip this section. You'll build in the appropriate steps so the application works with either type of authentication.)

But how does HTTP authentication actually work? First of all, the server knows what kind of security it needs to provide for each directory. One way to change that for a particular directory is to set things up in the main configuration for the server. Another way is to use an .htaccess file, which contains instructions for the directory in which it resides.

For example, you want the server to make sure all users who access your user-specific files have valid usernames and passwords, so first create a directory called *loggedin* inside the directory in which you currently have your files. For example, if your files reside in /usr/local/apache2/htdocs, you would create /usr/local/apache2/htdocs/loggedin.

Now you need to tell the server that you're going to override the overall security for that directory, so open the httpd.conf file and add the following to it:

```
<Directory /usr/local/apache2/htdocs/loggedin>
 AllowOverride AuthConfig
</Directory>
```

(You should, of course, use the correct directory for your own setup.)

Now it's time to prepare the actual directory.

Setting authentication

Next, create a new text file and save it in the loggedin directory with the name .htaccess. Add the following to it:

```
AuthName "Registered Users Only"
AuthType Basic
AuthUserFile /usr/local/apache2/password/.htpasswd
Require valid-user
```

Let's take this from the top. The `AuthName` is the text that appears at the top of the username and password box. The `AuthType` specifies that you're using `Basic` authentication, which means that you'll send the username and password in clear text. The `AuthUserFile` is the file that contains the allowable usernames and passwords. (You'll create that file in a moment.) Finally, the `Require` directive lets you specify who actually gets to see this content. Here, you're saying that you will show it to any valid user, but you also have the option to require specific users or user groups.

Restart the HTTP server so these changes can take effect.

(For Apache V2.0, call `<APACHE_HOME>/bin/apachectl stop`, followed by `<APACHE_HOME>/bin/apachectl start`.)

Next, you'll create the password file.

Creating the password file

For all this to work, you need to have a password file the server can check. In Adding new users to the password file on page 13, you'll look at manipulating this file from within PHP, but for now, if you have access to the command line, you can create the file directly.

First, choose a location for your .htpasswd file. It should *not* be in a directory that's Web accessible. It's not very secure if someone can simply download and analyze it. It should also be in a location where PHP can write to it. For example, you might create a password directory in your apache2 directory. Whichever location you choose, make sure you have the correct information in your .htaccess file.

To create the password file, execute the following command, substituting your own directory and username:

```
htpasswd -c /usr/local/apache2/password/.htpasswd roadnick
```

You are then prompted to type, then repeat the password, as in:

```
htpasswd -c /usr/local/apache2/password/.htpasswd roadnick
New password:
Re-type new password:
Adding password for user roadnick
```

The `-c` switch tells the server to create a new file, so after you've added the new user, the file looks something like this:

```
roadnick:IpoRzCGnsQv.Y
```

Note that this version of the password is encrypted, and you have to keep that in mind when you add passwords from your application.

Now let's see it in action.

Logging in

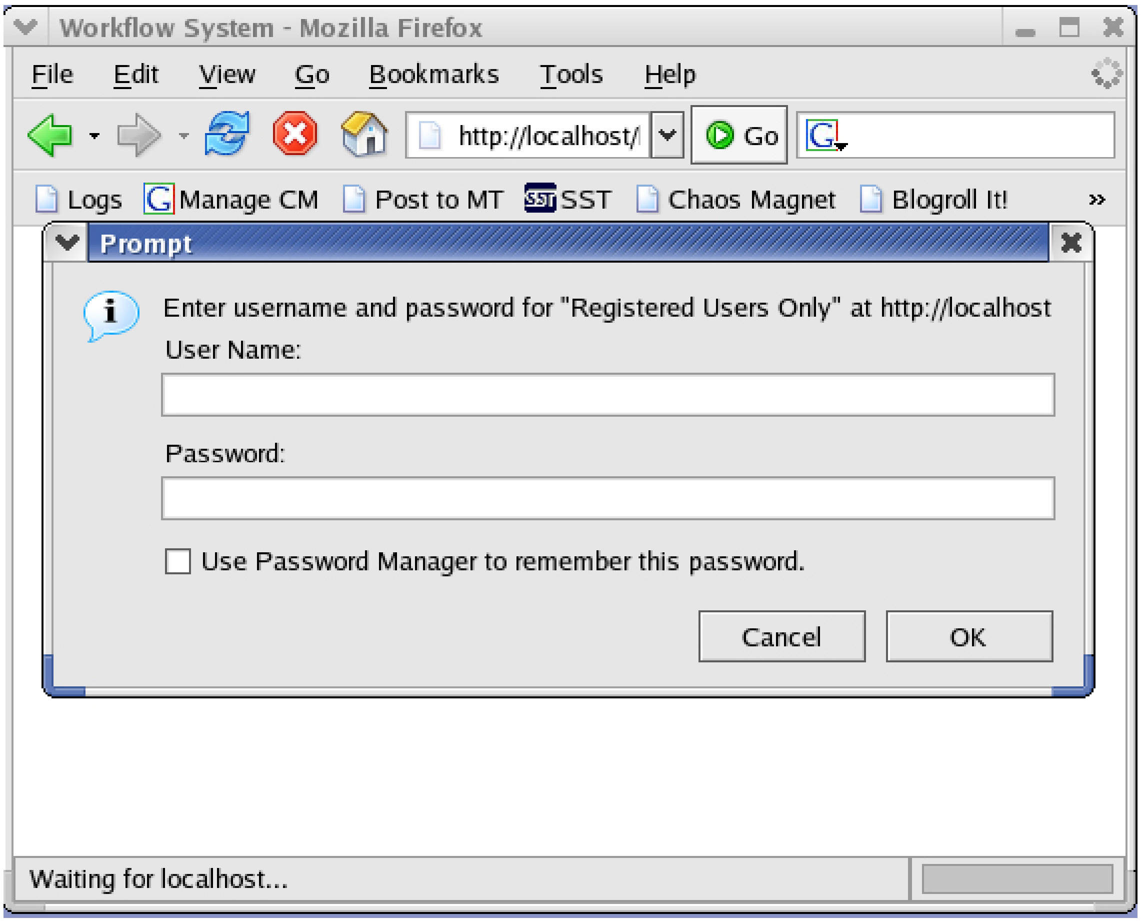
To see this in action, you need to access a file in the protected directory. Move the uploadfile.php and uploadfile_action.php files into the loggedin directory, and copy index.php into the loggedin directory as display_files.php.

In each of the three files, change the `include()` statements to account for the new location, as in:

```
<?php

   include ("../top.txt");
   include ("../scripts.txt");

   echo "Logged in user is ".$_SERVER['PHP_AUTH_USER'];

   display_files();

   include ("../bottom.txt");

?>
```

In this case, you fix the references to the included files, but you also reference a variable that should be set when the browser sends the username and password. Point your browser to http://localhost/loggedin/display_files.php to see this in action. As you can see in Figure 2, you should get a username and password box.

Figure 2. Username and password box



Enter username and password for "Registered Users Only" at http://localhost

User Name:

Password:

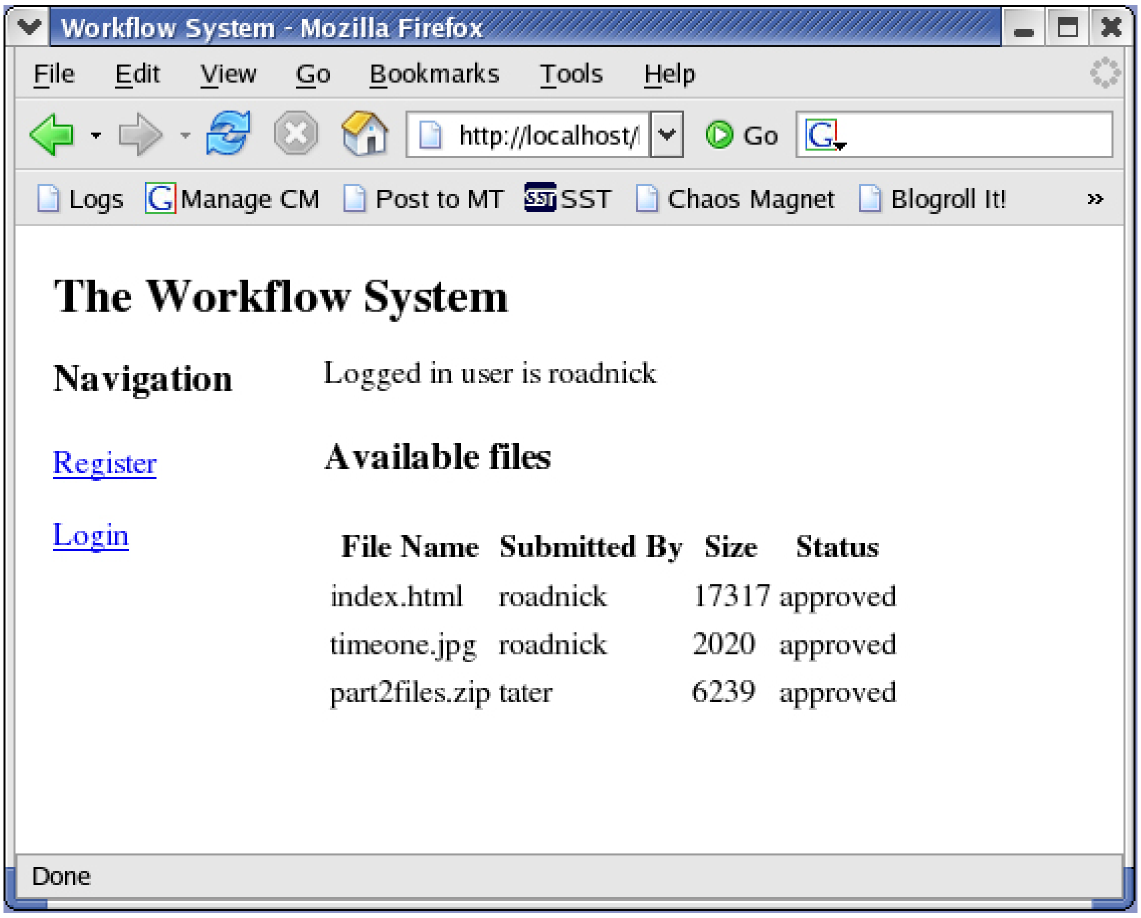
☐ Use Password Manager to remember this password.

Enter the username and password you used in Creating the password file on page 8 to see the actual page.

Using the login information

At this point, you've entered the username and login, so you can see the page. But as you can see in Figure 3, despite the message saying the user has logged in, the actual content doesn't seem to agree. You still see the **Register** and **Login** links, and the list of files still shows only those that an administrator has approved -- and not those that the current user has uploaded and are still pending.

Figure 3. Logged in ... sort of



The Workflow System

Navigation

Register

Login

Logged in user is roadnick

Available files

File Name	Submitted By	Size	Status
index.html	roadnick	17317	approved
timeone.jpg	roadnick	2020	approved
part2files.zip	tater	6239	approved

To solve these problems, you have two choices. The first is to go back and recode every instance in which the application references the username to look for the `$_SERVER['PHP_AUTH_USER']`, instead of `$_SESSION["username"]`. Good programmers are inherently lazy, however, so that's not a particularly attractive option.

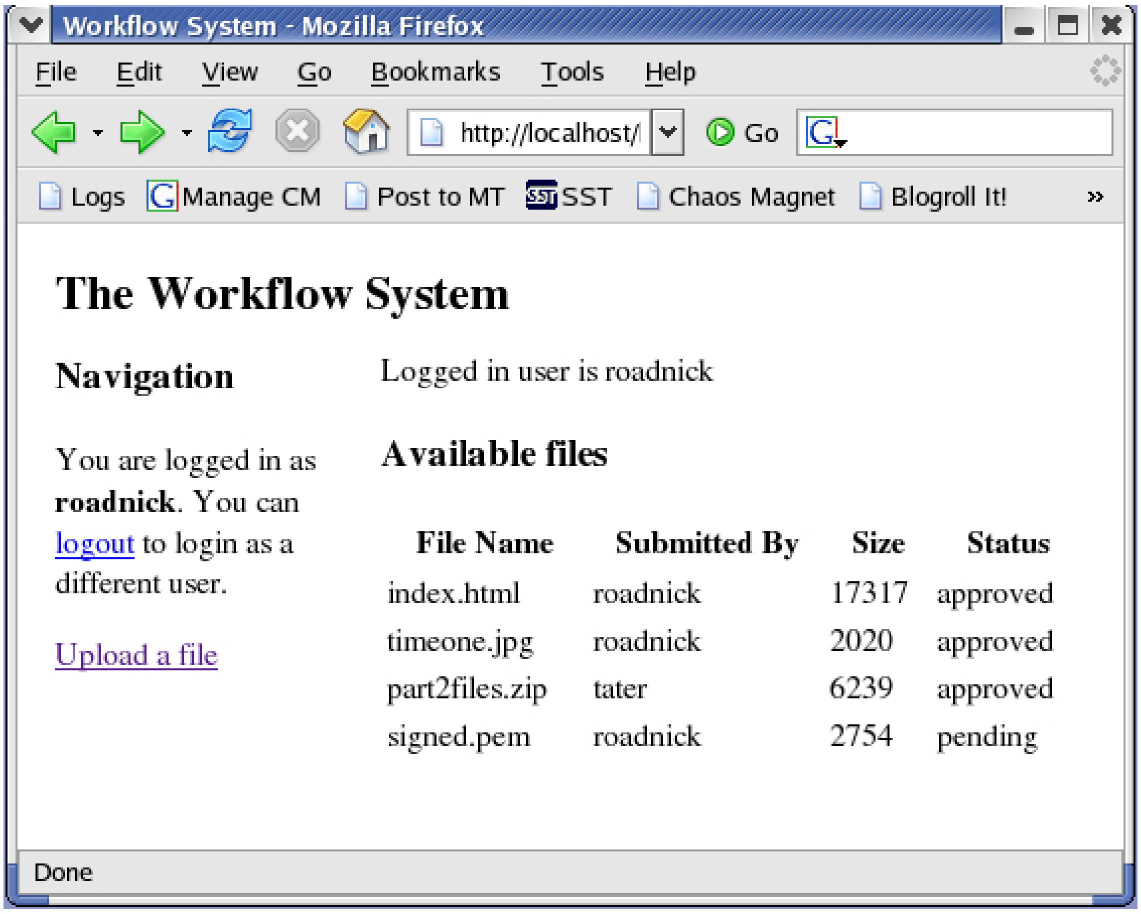
The second choice is to simply set `$_SESSION["username"]` based on `$_SERVER['PHP_AUTH_USER']` so everything will continue to work as it did before. You can do this in top.txt, right after you start a new session or join the existing one:

```
<?
   session_start();
   if (isset($_SESSION["username"])){
      //Do nothing
   } elseif (isset($_SERVER['PHP_AUTH_USER'])) {
      $_SESSION["username"] = $_SERVER['PHP_AUTH_USER'];
   }

?>
<html>
<head>
<title>Workflow System</title>
</head>
<body>
```

First off, the only way to make the browser "forget" the username and password you entered is to close the browser so you give the `$_SESSION["username"]` variable precedence. That way, you have the option to enable users to log in as someone else. (You won't do that here, but you do have the option.)

Next, if neither the `$_SESSION` nor `$_SERVER` variable is set, nothing happens, and the page continues on as though the user isn't logged in -- which happens to be the case. Making this one simple change fixes your login problem, as you can see in Figure 4.

Figure 4. The corrected page


Workflow System - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost/
Go
Logs Manage CM Post to MT SST Chaos Magnet Blogroll It!
Done

The Workflow System

Navigation

You are logged in as **roadnick**. You can logout to login as a different user.

Upload a file

Logged in user is roadnick

Available files

File Name	Submitted By	Size	Status
index.html	roadnick	17317	approved
timeone.jpg	roadnick	2020	approved
part2files.zip	tater	6239	approved
signed.pem	roadnick	2754	pending

Fixing the interface

Before you move on to adding a new user, you need to make a couple of quick fixes to top.txt to accommodate the new structure. For one thing, you need to change the **Login** link so that rather than pointing to your old login.php page, it points to the newly secured display_files.php file. When the user attempts to access it, the browser will provide a way to log in:

```
...
<tr>
   <td width="30%" valign="top">
```

```
      <h3>Navigation</h3>

<?php
   if (isset($_SESSION["username"]) || isset($username)){
?>
      <p>You are logged in as <b><?=$_SESSION["username"].$username?></b>. <!--
You can <a href="/logout.php">logout</a> to login as a different user.--></p>

      <p><a href="/loggedin/uploadfile.php">Upload a file</a></p>
      <p><a href="/loggedin/display_files.php">Display files</a></p>

<?php
   } else {
?>
      <p><a href="/registration.php">Register</a></p>
      <p><a href="/loggedin/display_files.php">Login</a></p>
<?php
   }
?>
   </td>
```

Notice that in addition to fixing the login reference and adding a new option for displaying the list of files, we commented out the message about logging out, only because that subject is beyond the scope of this tutorial.

Now you just need to integrate the registration process with the password file.

Adding new users to the password file

The last step in this process is to integrate your registration with the .htpasswd file. To do that, you simply need to add a new entry once you have saved the user to the database. Open registration_action.php and add the following:

```
...
           $passwords = $_POST["pword"];
           $sql = "insert into users (username, email, password) values ('"
                                    .$_POST["name"]."', '".$_POST["email"]
                                    ."', '".$passwords[0]."')";
           $result = mysql_query($sql);

           if ($result){
              echo "It's entered!";

              $pwdfile = '/usr/local/apache2/password/.htpasswd';
              if (is_file($pwdfile)){
                 $opencode = "a";
              } else {
                 $opencode = "w";
              }
              $fp = fopen($pwdfile, $opencode);
              $pword_crypt = crypt($passwords[0]);
              fwrite($fp, $_POST['name'].":".$pword_crypt."\n");
              fclose($fp);

           } else {
               echo "There's been a problem: ".mysql_error();
```

```
                 }
             } else {

                 echo "There is already a user with that name.  Please try again. <br />";
   ...
```

Before you start, if you have a .htpasswd file already, make sure the user can write to it on your Web server. If not, make sure the user can write to the appropriate directory.

First off, check to see if the file exists and use that information to determine whether you're going to write a new file or append information to an existing file. Once you know, go ahead and open the file.

As you saw in Creating the password file on page 8, the password is stored in encrypted form, so you can use the `crypt()` function to get that string. Finally, write the username and password out to the file and close the file.

To test this, quit the browser to clear out any cached passwords, then open http://localhost/index.php.

Click **Register** and create a new account. When you're finished creating the account, quit the browser again and try to access a protected page. Your new username and password should work.

Section 4. Using streams

What are streams?

Now that you've got the system set up, you're ready to enable the user to actually download the available files. From the very beginning, however, these files have been stored in a non-Web-accessible directory, so a simple link to them is out of the question.

Instead, in this section, you're going to create a function that streams the file from its current location to the browser.

Now, the way in which you actually access a resource, such as a file, depends on where and how it's stored. Accessing a location file is very different from accessing one on a remote server via HTTP or FTP.

Fortunately, however, PHP provides *stream wrappers*. In other words, you make a call to a resource, wherever it is, and if PHP has an available wrapper, it will figure out just how to make that call.

You can find out which wrappers are available by printing the contents of the array returned by the `stream_get_wrappers()` function, like so:

```
<?php

print_r(stream_get_wrappers());

?>
```

The `print_r()` function is extremely handy for seeing the contents of an array. For example, your system might give you:

```
Array
(
    [0] => php
    [1] => file
    [2] => http
    [3] => ftp
)
```

This would enable you to easily store your files on a remote Web server or FTP server as an alternative to storing them as files on the local server, and the code you use in this section will still work.

Let's take a look.

Downloading the file

For the user to be able to see a file, the browser has to receive it. It also has to know what the file is in order to display it properly. You can take care of both of these issues. Create a new file called *download_file.php* and save it in the loggedin directory. Add the following:

```
<?php

   include ("../scripts.txt");

   $filetype = $_GET['filetype'];
   $filename = $_GET['file'];
   $filepath = UPLOADEDFILES.$filename;

   if($stream = fopen($filepath, "rb")){
      $file_contents = stream_get_contents($stream);
      header("Content-type: ".$filetype);
      print($file_contents);
   }

?>
```

Despite its power, the process here is actually quite straightforward. First, you open the file for reading and for buffering. What you're actually doing with the `fopen()` function is creating a resource that represents the file. You can then pass that resource to `stream_get_contents()`, which reads the entire file out into a single string.

Now that you have the content, you can send it to the browser, but the browser won't know what to do with it and will likely display it as text. That's fine for a text file, but not so good for an image, or even an HTML file. So, rather than just sending it raw, you first send a `header` to the browser with information on the `Content-type` of the file, such as `image/jpeg`.

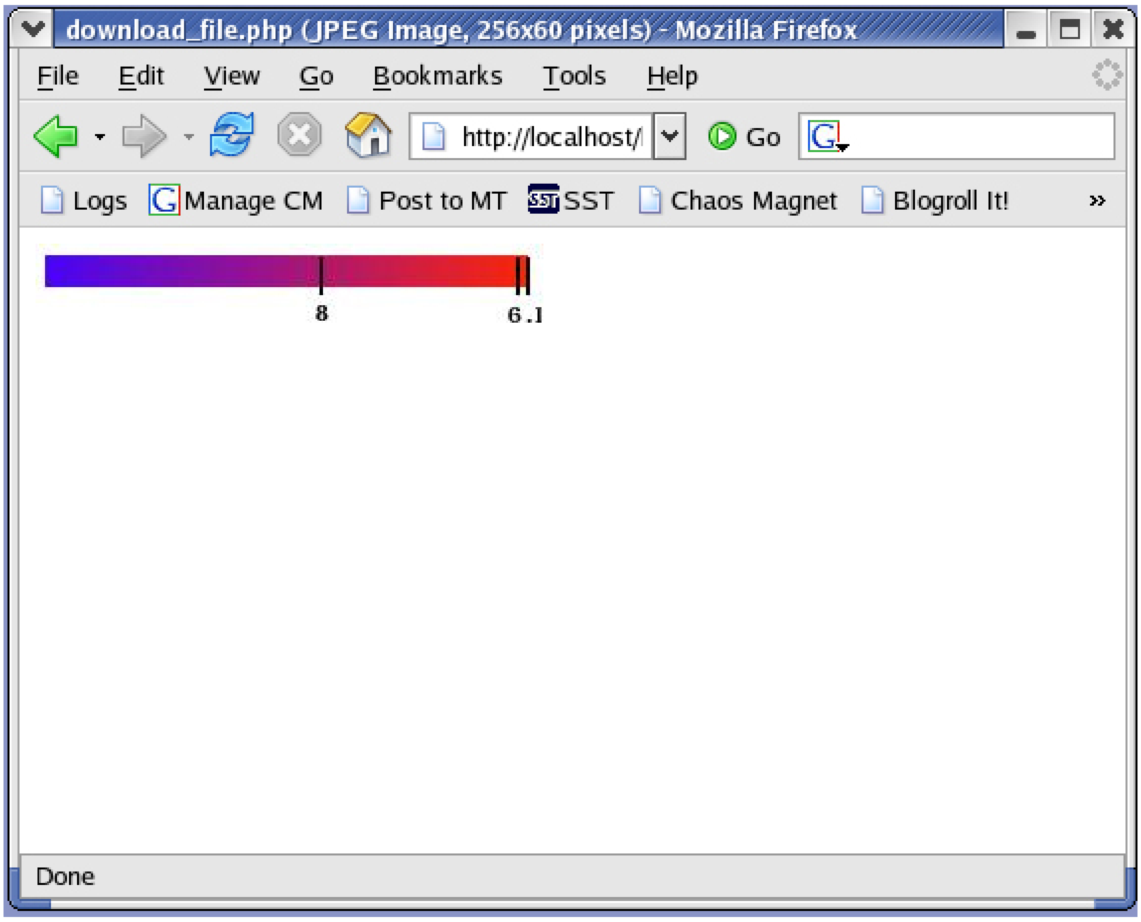
Finally, you simply output the contents of the file to the browser. Having received the `Content-type` header, the browser will know not to simply treat it as a Web page (unless, of course, it *is* a Web page).

As far as deciding which file and type to actually use, you're reading these from the `$_GET` array, so you can add them right to the URL, as in:

```
http://localhost/loggedin/download_file.php?file=timeone.jpg&filetype=image/jpeg
```

Enter this URL (with an appropriate file name and type, of course) into your browser to see the results shown in Figure 5.

Figure 5. Downloading a file


download_file.php (JPEG Image, 256x60 pixels) - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost/
Go
Logs
Manage CM
Post to MT
SST
Chaos Magnet
Blogroll It!
8
6.1
Done

Adding a link to the file

Because all the information the download page needs can be added to the URL, it's simple to add a link enabling the user to download a file. You create the display of available files using a SAX stream, which means that the actual output is controlled by a content handler class called, in this case, `Content_Handler`. Open the scripts.txt file and add the following code to the `Content_Handler` class:

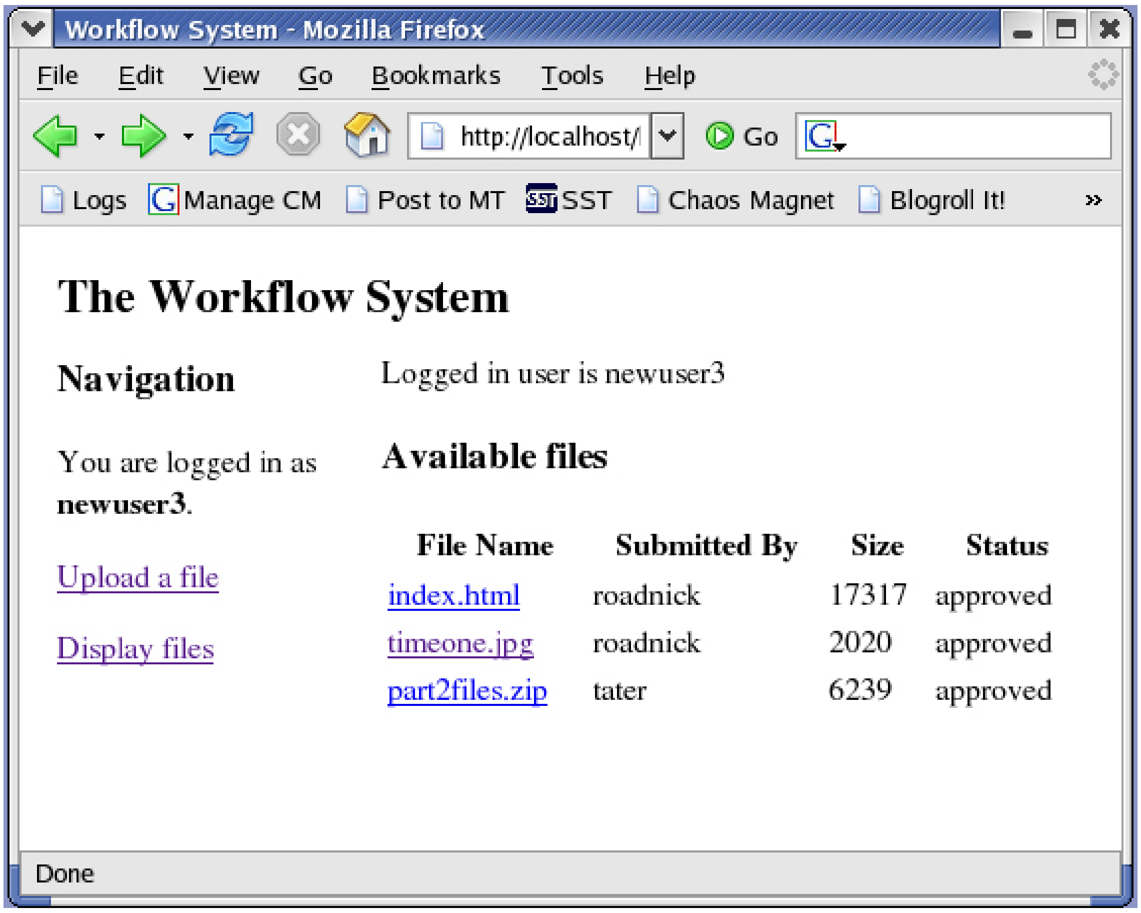
```
class Content_Handler{

  private $available = false;
  private $submittedBy = "";
  private $status = "";

  private $currentElement = "";

  private $fileName = "";
  private $fileSize = "";
  private $fileType = "";

  function start_element($parser, $name, $attrs){
...
```

```
  }

  function end_element($parser, $name){

     if ($name == "workflow"){
        echo "</table>";
     }

     if ($name == "fileInfo"){
        echo "<tr><td><a href='download_file.php?file=".$this->fileName."
&filetype=".$this->fileType."'>"
                     .$this->fileName."</a></td>".
               "<td>".$this->submittedBy."</td>".
               "<td>".$this->fileSize."</td>".
               "<td>".$this->status."</td></tr>";

        $this->fileName = "";
        $this->submittedBy = "";
        $this->fileSize = "";
        $this->status = "";
        $this->fileType = "";

        $this->available = false;
     }

     $this->currentElement = "";

  }

  function chars($parser, $chars){

     if ($this->available){
         if ($this->currentElement == "fileName"){
            $this->fileName = $this->fileName . $chars;
         }
         if ($this->currentElement == "fileType"){
            $this->fileType = $this->fileType . $chars;
         }
         if ($this->currentElement == "size"){
            $this->fileSize = $this->fileSize . $chars;
         }
      }

  }
}
```

In addition to the information you were already tracking for each `fileInfo` element, you now need to track the `fileType`, so you need to add a property for that. (We talk more about properties in on page .)

Down in the `chars()` function, you store the value when you get to it. When you get to the end of the `fileInfo` element and it's time to display the information, you use it, along with the `fileName`, to create a link that points to the download page. You can see the results in Figure 6.

Figure 6. Linking to the file



The Workflow System

Navigation

You are logged in as **newuser3**.

Upload a file

Display files

Logged in user is newuser3

Available files

File Name	Submitted By	Size	Status
index.html	roadnick	17317	approved
timeone.jpg	roadnick	2020	approved
part2files.zip	tater	6239	approved

Click a link to verify the file.

Next, you'll look at encapsulating this process into an object.

Section 5. Using objects

What are objects, anyway?

In this section, you're going to look at the use of *objects*. So far, almost everything you've done has been *procedural*, meaning you have a script that pretty much runs from beginning to end. Now you're going to move away from that.

The central concept of object-oriented programming is the idea that you can represent "things" as a self-sufficient bundle. For example, an electric kettle has properties, such as its color and maximum temperature, and capabilities, such as heating the water and turning itself off.

If you were to represent that kettle as an object, it would also have properties, such as `color` and `maximumTemperature`, and capabilities -- or *methods* -- such as `heatWater()` and `turnOff()`. If you were writing a program that interfaced with the kettle, you would simply call the kettle object's `heatWater()` method, rather than worrying about how it's actually done.

To make things a bit more relevant, you're going to create an object that represents a file to be downloaded. It will have properties, such as the name and type of the file, and methods, such as `download()`.

Having said all that, however, we need to point out that you don't actually define an object. Instead, you define a *class* of objects. A class acts as a kind of "template" for objects of that type. You then create an *instance* of that class, and that instance is the object.

Let's start by creating the actual class.

Creating the WFDocument class

The first step in dealing with objects is to create the class on which they are based. You could add this definition to the scripts.txt file, but you're trying to make the code more maintainable, not less. So, create a separate file, WFDocument.php, and save it in the main directory. Add the following:

```
<?php

include_once("scripts.txt");

class WFDocument {

   function download($filename, $filetype) {

      $filepath = UPLOADEDFILES.$filename;
```

```
        if($stream = fopen($filepath, "rb")){
          $file_contents = stream_get_contents($stream);
          header("Content-type: ".$filetype);
          print($file_contents);
        }
     }
  }

  ?>
```

First, you need the `UPLOADEDFILES` constant, so you include the scripts.txt file. Next, you create the actual class. The `WFDocument` class has only a single method, `download()`, which is the same as the code in download_file.php, with the exception of receiving the file name and type as inputs to the function rather than directly extracting them from the `$_GET` array.

Now let's look at instantiating this class.

Calling the WFDocument-type object

You've actually already instantiated several objects when you were working with DOM in *Part 2* of this series, but we didn't say much about why or how. We will remedy that now.

Open the download_file.php page and change the code so it reads as follows:

```
  <?php

     include ("../WFDocument.php");

     $filetype = $_GET['filetype'];
     $filename = $_GET['file'];

     $wfdocument = new WFDocument();
     $wfdocument->download($filename, $filetype);

  ?>
```

First off, rather than including the scripts.txt file, you're including the definition of the `WFDocument` class, which you put into the WFDocument.php file. (Some developers find it useful to simply create a page that includes all their classes, then include that page rather than including individual classes all over the place.)

Now you're ready to create a new object, which you do using the `new` keyword. This line creates a new object of the type `WFDocument` and assigns it to the `$wfdocument` variable.

Once you have a reference to that object, you can call any of its public methods. In this case, there's only one method, `download()`, and you call it using the `->` operator. Basically, this symbol says, "Use the method (or property) that belongs to this object."

Save the file and test it by clicking one of the links on your page. The code is exactly the same as it was before. The only difference is how you're calling it.

Creating properties

Of course, methods are only part of the story. The whole point of an object is that it's encapsulated. In other words, it should contain all its own information, so rather than feeding the name and file type to the `download()` method, you can set them as properties on the object. But first you have to create them in the class:

```
<?php

include_once("../scripts.txt");

class WFDocument {

   public $filename;
   public $filetype;

   function download() {

      $filepath = UPLOADEDFILES.$this->filename;

      if($stream = fopen($filepath, "rb")){
        $file_contents = stream_get_contents($stream);
        header("Content-type: ".$this->filetype);
        print($file_contents);
      }
   }
}

?>
```

Notice that you declare the variables outside the function; they're part of the class and not the function. You're also declaring them as `public`, which means you can access them from outside the class itself. You can also set a property as `private`, which means you can use it only within the class itself, or `protected`, which means you can use it only within the class or any classes based on this one. (If you're unfamiliar with this idea, hang on for a little while. We will talk more about this concept, *inheritance*, in Creating a custom exception on page29 .)

Finally, to reference an object property, you have to know which object it belongs to. Within an object itself, you can just use the keyword `$this`, which refers to the object itself. This way, you can use `$this->filename` to refer to the `filename` property of the object executing this code.

Now let's look at setting values for these properties.

Setting properties

Rather than passing information to an object, you want to actually set the properties of the object:

```
<?php

   include ("../WFDocument.php");

   $filetype = $_GET['filetype'];
   $filename = $_GET['file'];

   $wfdocument = new WFDocument();
   $wfdocument->filename = $filename;
   $wfdocument->filetype = $filetype;
   $wfdocument->download();

?>
```

Notice the notation here. You're using the object name, `$wfdocument`, the `->` operator, and the name of the property. Once these properties have been set, they're available from inside the object, so you don't have to pass them to the `download()` method.

Now, having done all that, there is actually a better way to handle this kind of thing, so let's look at an alternative.

Hiding properties

Although it's certainly *possible* to set the value of a property directly, as you did in the previous panel, it's not the best way to handle things. Instead, the general practice is to hide the actual properties from the public and use *getters* and *setters* to get and set their values, like so:

```
<?php

include_once("../scripts.txt");

class WFDocument {

   private $filename;
   private $filetype;

   function setFilename($newFilename){
      $this->filename = $newFilename;
   }
   function getFilename(){
      return $this->filename;
   }

   function setFiletype($newFiletype){
      $this->filetype = $newFiletype;
   }
   function getFiletype(){
      return $this->filetype;
   }

   function download() {

      $filepath = UPLOADEDFILES.$this-> getFilename()

      if($stream = fopen($filepath, "rb")){
        $file_contents = stream_get_contents($stream);
        header("Content-type: ".$this->getFiletype())
        print($file_contents);
      }
   }
}

?>
```

First, you define the properties as `private`. That means that if you try to set them directly, as you've been doing, you'll get an error. But you still have to set these values, so instead you use the `getFilename()`, `setFilename()`, `getFiletype()`, and `setFiletype()` methods. Notice that you use them here in the `download()` method, just as you would have used the original property.

Using getters and setters is handy because it gives you more control over what's happening to your data. For example, you might want to perform certain validation checks before you allow a particular value to be set for a property.

Calling hidden properties

Now that you've hidden the properties, you need to go back and modify the download_file.php page so you don't get an error:

```
   <?php

      include ("../WFDocument.php");

      $filetype = $_GET['filetype'];
      $filename = $_GET['file'];

      $wfdocument = new WFDocument();
      $wfdocument->setFilename($filename);
      $wfdocument->setFiletype($filetype);
      $wfdocument->download();

   ?>
```

Handy as this approach is, there are easier ways to set properties on an object.

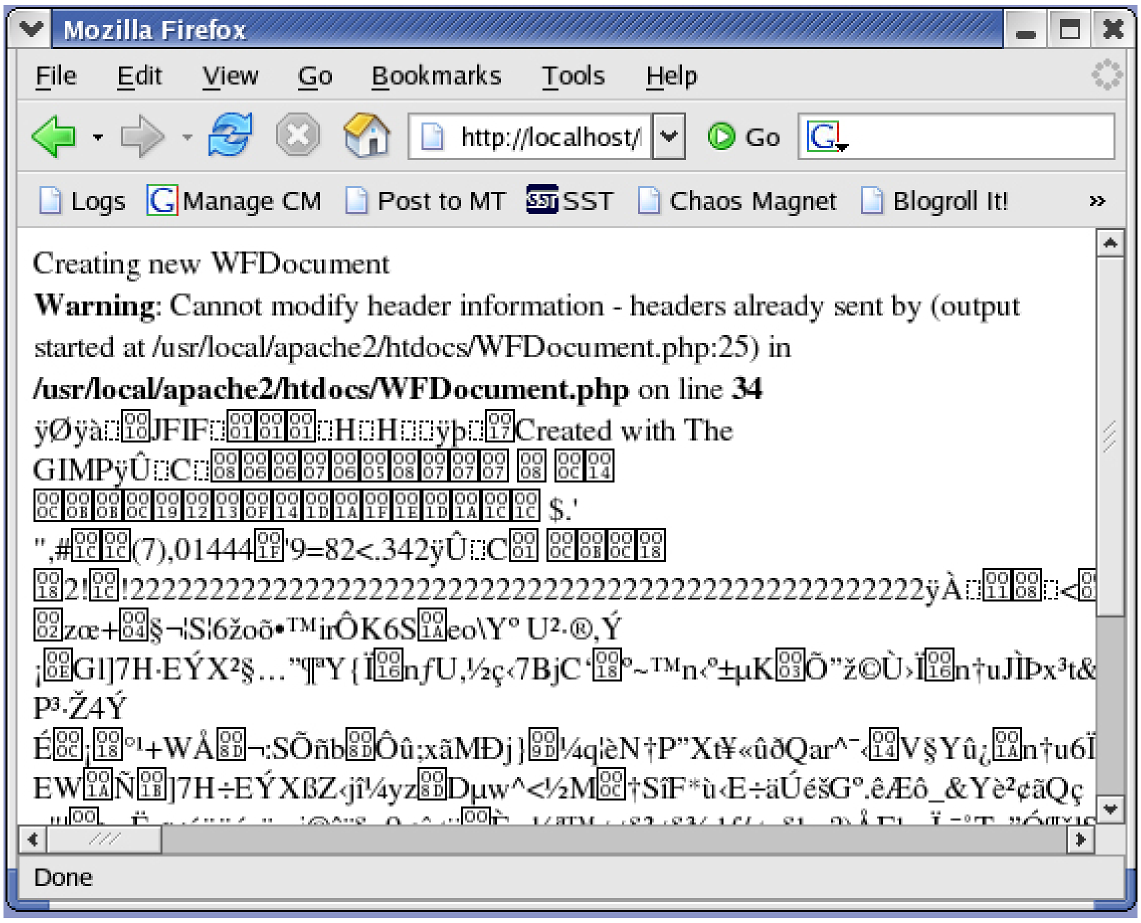
Creating a constructor

If an object has a constructor, it gets called every time you create a new instance of that particular class. For example, you could create a simple constructor:

```
...
   function getFiletype(){
      return $this->filetype;
   }

   function __construct(){
      echo "Creating new WFDocument";
   }

   function download() {

      $filepath = UPLOADEDFILES.$this->filename;
...
```

If you try to run this script as is, you'll see an error because the object outputs the text (`Creating new WFDocument`) before it outputs the headers, as you can see in Figure 7.

Figure 7. Error after running script


Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost/
Go
Logs Manage CM Post to MT SST Chaos Magnet Blogroll It!
Done

```
Creating new WFDocument
Warning: Cannot modify header information - headers already sent by (output
started at /usr/local/apache2/htdocs/WFDocument.php:25) in
/usr/local/apache2/htdocs/WFDocument.php on line 34
ÿØÿà JFIF H H ÿþ Created with The
GIMPÿÛ C
$.'
",#(7),01444'9=82<.342ÿÛ C
2!!22222222222222222222222222222222222222222222222222ÿÀ <
zœ+§¬¦S¦6žoõ•™irÔK6Seo\Y° U²·®,Ý
[illegible]
P³·Ž4Ý
[illegible]
[illegible]
```

So, even though you never explicitly called the `__construct()` method, the application called it as soon as the object was instantiated. You can use that to your advantage by adding information to the constructor.

Creating an object with information

One of the most common uses for a constructor is to provide a way to initialize various values when you create the object. For example, you can set up the `WFDocument` class so that you set the `filename` and `filetype` properties when you create the object:

```
...
   function getFiletype(){
      return $this->filetype;
   }

   function __construct($filename = "", $filetype = ""){
      $this->setFilename($filename);
      $this->setFiletype($filetype);
   }

   function download() {

      $filepath = UPLOADEDFILES.$this->filename;

...
```

When you create the object, PHP carries out any instructions in the constructor before moving on. In this case, that constructor is looking for the `filename` and `filetype`. If you don't supply them, you still won't get an error, because you've specified default values to use if no value is given when the function is called.

But how do you explicitly call the `__construct()` function?

Creating the object: Calling the constructor

You don't actually call the constructor method explicitly. Instead, you call it implicitly every time you create an object. That means you use that specific moment to pass information for the constructor:

```
<?php

   include ("../WFDocument.php");

   $filetype = $_GET['filetype'];
   $filename = $_GET['file'];

   $wfdocument = new WFDocument($filename, $filetype);
   $wfdocument->download();
```

```
?>
```

Any information passed to the class when you create the new object gets passed to the constructor. This way, you can simply create the object and use it to download the file.

Section 6. Handling exceptions

A generic exception

Because exceptions come into play when something is not quite right with an application, they are often confused with errors. Exceptions are, however, much more flexible. In this section, you'll see how to define different types of exceptions and use them to determine what's going on with the application.

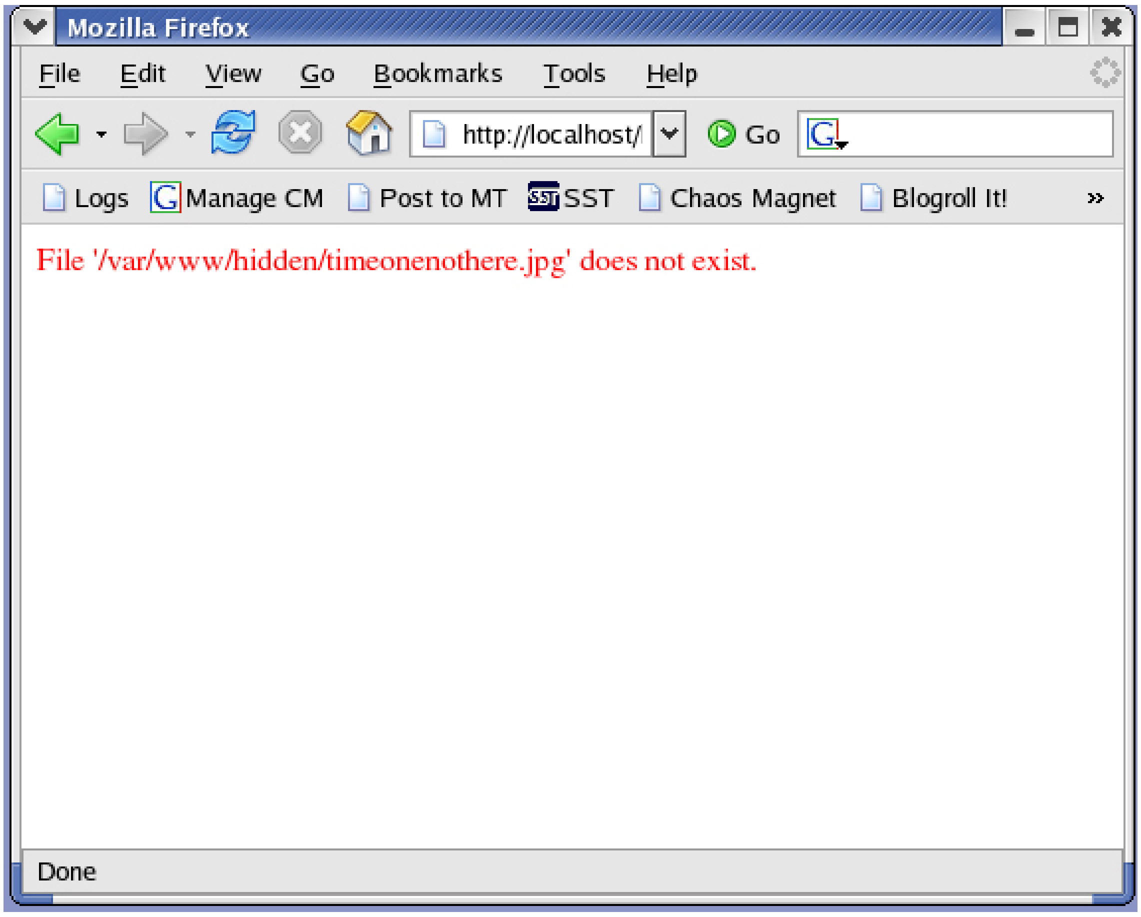
Let's start with a simple generic exception in the definition of the `WFDocument` class:

```
<?php

include_once("../scripts.txt");

class WFDocument {
...
   function download() {

      $filepath = UPLOADEDFILES.$this->filename;

      try {

         if(file_exists($filepath)){
           if ($stream = fopen($filepath, "rb")){
              $file_contents = stream_get_contents($stream);
              header("Content-type: ".$this->filetype);
              print($file_contents);
           }
         } else {
           throw new Exception ("File '".$filepath."' does not exist.");
         }

      } catch (Exception $e) {

         echo "<p style='color: red'>".$e->getMessage()."</p>";

      }
   }
}

?>
```

First off, exceptions don't just happen, they are *thrown*. And, of course, if you throw something, you have to catch it, so you create a *try-catch* statement. In the `try` section, you put your code. If something untoward happens, such as, in this case, a file doesn't exist, and you throw an exception, PHP moves immediately to the `catch` block to catch the exception.

An exception has many properties, such as the line and file from which the exception was thrown, and a message. Typically, the application sets the message when it throws the exception, as you see here. The exception itself, `$e`, can then provide that text using the `getMessage()` method. For example, if you try to download a file that doesn't exist, you'll see the message displayed, in red, as in Figure 8.

Figure 8. The basic exception


Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost/
Go
Logs Manage CM Post to MT SST Chaos Magnet Blogroll It!
Done

File '/var/www/hidden/timeonenothere.jpg' does not exist.

The real power of exceptions, though, comes from creating your own.

Creating a custom exception

In the last section, you examined objects, but we left out one very important aspect of them: inheritance. Let's look at that now.

One advantage to using classes is the ability to use one class as the basis for another. For example, you can create a new exception type, `NoFileExistsException`, which extends the original `Exception` class:

```
class NoFileExistsException extends Exception {

   public function informativeMessage(){
      $message = "The file, '".$this->getMessage()."', called on line ".
           $this->getLine()." of ".$this->getFile().", does not exist.";
      return $message;
   }
}
```

(For simplicity's sake, we added this code to the WFDocument.php file, but you can add it wherever it's accessible when you need it.)

Here, you've created a new class, `NoFileExistsException`, with a single method: `informativeMessage()`. In actuality, this class is *also* an `Exception`, so all the public methods and properties for an `Exception` object are also available.

For example, notice that within the `informativeMessage()` function, you call the `getLine()` and `getFile()` methods, even though they're not defined here. They're defined in the base class, `Exception`, so you can use them.

Now let's see it in action.

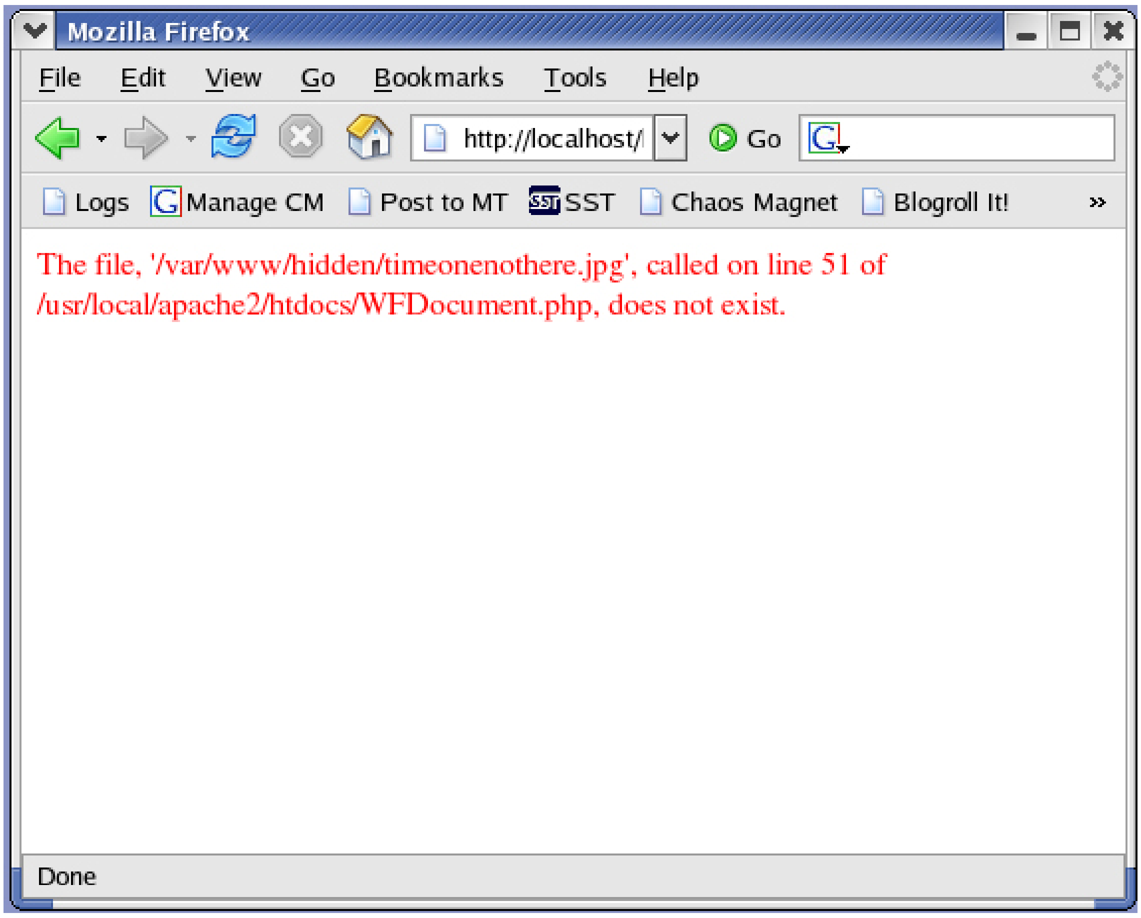
Catching a custom exception

The easiest way to use the new exception type is to simply throw it just as you would throw a generic `Exception`:

```
function download() {

      $filepath = UPLOADEDFILES.$this->filename;

      try {

         if(file_exists($filepath)){
           if ($stream = fopen($filepath, "rb")){
              $file_contents = stream_get_contents($stream);
              header("Content-type: ".$this->filetype);
              print($file_contents);
           }
         } else {
           throw new NoFileExistsException ($filepath);
         }

      } catch (NoFileExistsException $e) {

         echo "<p style='color: red'>".$e->informativeMessage()."</p>";

      }
   }
```

Notice that even though you pass only the `$filepath` when you create the exception, you get the full message back, as shown in Figure 9.

Figure 9. Using a custom exception


Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost/
Go
Logs Manage CM Post to MT SST Chaos Magnet Blogroll It!
Done

The file, '/var/www/hidden/timeonenothere.jpg', called on line 51 of /usr/local/apache2/htdocs/WFDocument.php, does not exist.

Working with multiple exceptions

One reason to create custom exception classes is so you can use PHP's ability to distinguish between them. For example, you can create multiple `catch` blocks for a single `try`:

```
...
   function download() {

      $filepath = UPLOADEDFILES.$this->filename;

      try {

         if(file_exists($filepath)){
           if ($stream = fopen($filepath, "rb")){
              $file_contents = stream_get_contents($stream);
              header("Content-type: ".$this->filetype);
              print($file_contents);
           } else {
              throw new Exception ("Cannot open file ".$filepath);
           }
         } else {
           throw new NoFileExistsException ($filepath);
```

```
            }

         } catch (NoFileExistsException $e) {

            echo "<p style='color: red'>".$e->informativeMessage()."</p>";

         } catch (Exception $e){

            echo "<p style='color: red'>".$e->getMessage()."</p>";
         }
      }
   }
```

In this case, you attempt to catch problems before they happen by checking for the existence of the file and throwing a `NoFileExistsException`. If you get past that hurdle and something else keeps you from opening the file, you throw a generic exception. PHP detects which type of exception you throw and executes the appropriate `catch` block.

All of this might seem a little overboard for simply outputting messages, but there's nothing that says that's all you can do. You can create custom methods for your exception that, for example, send notifications for particular events. You can also create custom `catch` blocks that perform different actions depending on the situation.

Of course, just because you defined all these different exceptions doesn't mean you have to catch each one individually, as you'll see next.

Propagating exceptions

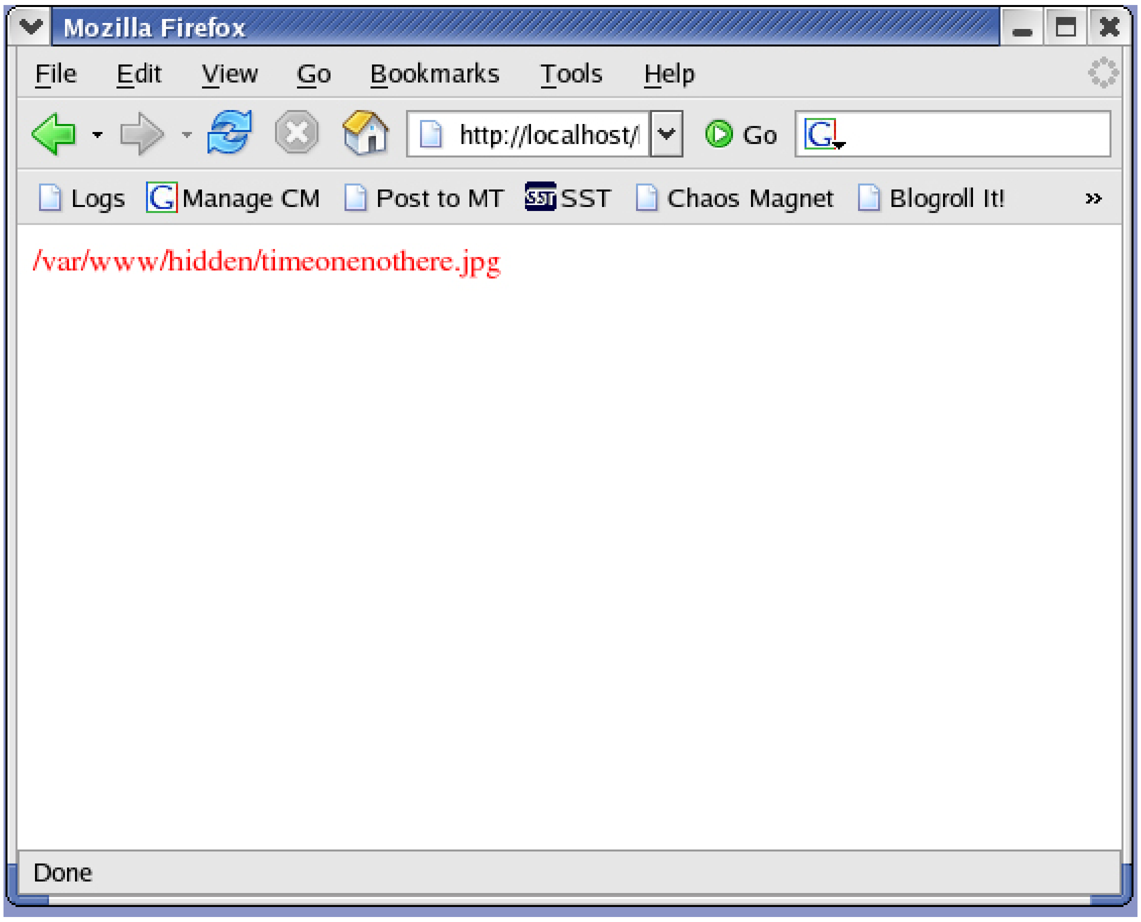
Another handy feature of inheritance is the ability to treat an object as though it were a member of its base class. For example, you can throw a `NoFileExistsException` and catch it as a generic `Exception`:

```
   ...
      function download() {

         $filepath = UPLOADEDFILES.$this->filename;

         try {

            if(file_exists($filepath)){
              if ($stream = fopen($filepath, "rb")){
                 $file_contents = stream_get_contents($stream);
                 header("Content-type: ".$this->filetype);
                 print($file_contents);
              } else {
                 throw new Exception ("Cannot open file ".$filepath);
              }
            } else {
              throw new NoFileExistsException ($filepath);
            }
```

```
        } catch (Exception $e){

           echo "<p style='color: red'>".$e->getMessage()."</p>";
        }
     }
  }
```

In this case, when you throw the exception, PHP works its way down the list of `catch` blocks, looking for the first one that applies. Here you have only one, but it will catch *any* `Exception`, as shown in Figure 10.

Figure 10. Propagating exceptions



```
/var/www/hidden/timeonenothere.jpg
```

Section 7. Putting it together

What you need to do

Now that you've got the file download process in place, it's time to put everything together and finish off the application. In this section, you're going to take care of some miscellaneous tasks that still need doing:

- Creating individual file identifiers
- Detecting administrators
- Creating the form that enables an administrator to approve files
- Checking downloads to make sure they're not being called from another server

Start by creating a `Counter` class.

Identifying individual documents

So far, you haven't been at all concerned about identifying specific files, except when you're downloading them, but now you need to pay a bit more attention. Ultimately, you'll be processing a form that enables an administrator to approve specific files, so it would be helpful to have an easy way to refer to them.

What you're going to do here is create a `Counter` class that lets you generate a unique *key* for each file. You then add that key to the XML file, enabling you to directly request the appropriate `fileInfo` element. You start by creating the `Counter` class definition. You might, for example, put it in the scripts.txt file:

```
class Counter{

   function getNextId(){
     $filename = "/usr/local/apache2/htdocs/counter.txt";
     $handle = fopen($filename, "r+");
     $contents = fread($handle, filesize($filename));

     $nextid = $contents + 1;
     echo $nextid;
     rewind($handle);
     fwrite($handle, $nextid);
     fclose($handle);

     return $nextid;
   }

}
```

Here you have a single function, `getNextId()`, that reads an existing file, counter.txt, and increments the contents by 1. (So, before you start, create the file with the single entry `0`.) It then goes back to the beginning of the file and writes the new value so it will be present the next time you call the function.

You use this class when you add the file information to the XML file.

Adding the identifier to the XML file

Ultimately, you want to be able to retrieve a single `fileInfo` element by its `ID` attribute, so go ahead and add this information to the docinfo.xml file created in "*Learning PHP, Part 2*."

```
function save_document_info($fileInfo){

   $xmlfile = UPLOADEDFILES."docinfo.xml";
...
   $filename = $fileInfo['name'];
   $filetype = $fileInfo['type'];
   $filesize = $fileInfo['size'];

   $fileInfo = $doc->createElement("fileInfo");

   $counter = new Counter();
   $fileInfo->setAttribute("id", "_".$counter->getNextId());

   $fileInfo->setAttribute("status", "pending");

   $fileInfo->setAttribute("submittedBy", getUsername());
...
   $doc->save($xmlfile);

}
```

Every time a new document's information is saved, you create a `Counter` object and use its `getNextId()` method to provide a unique value for the `id` attribute. Because you're later going to specify this attribute as being of type `ID`, you're preceding the value with an underscore (_) because these values can't start with a number.

The results look something like the following (we added spacing to make it a bit easier to read):

```
<fileInfo id="_13" status="pending" submittedBy="roadnick">
   <approvedBy/>
   <fileName>timeone.jpg</fileName>
   <location>/var/www/hidden/</location>
   <fileType>image/jpeg</fileType>
   <size>2020</size>
</fileInfo>
```

Note that this process does not affect any of your existing data, so you need to manually add `id` attributes to all your `fileInfo` elements or delete the docinfo.xml file and start again, uploading files to work with.

Now you're ready to approve files, but first you need to set up the administrators who are going to do it.

Detecting administrators

When you originally created the users table in the database, you didn't take into consideration the fact that you needed to distinguish between regular users and administrators, so you have to take care of that now. Log into MySQL and execute the following commands:

```
alter table users add status varchar(10) default 'USER';
update users set status = 'USER';
update users set status = 'ADMIN' where id=3;
```

The first command adds the new column, `status`, to the users table. You didn't specify the user type on the registration page, so you simply specify a default value of `USER` for any new users added to the system. The second command sets this status for the existing users. Finally, you choose a user to make into an administrator. (Make sure to use the appropriate `id` value for your data.)

Now that you have the data, you can create a function that returns the current user's status:

```
function getUserStatus(){
   $username = $_SESSION["username"];
   db_connect();
   $sql = "select * from users where username='".$username."'";

   $result = mysql_query($sql);
   $row = mysql_fetch_array($result);

   $status = "";

   if ($row) {
     $status = $row["status"];
   } else {
     $status = "NONE";
   }

   mysql_close();

   return $status;

}
```

Let's review how this process works. First, you create a connection to the appropriate database using the script you created: `db_connect()`. You can then create a SQL statement using the username, stored in the `$_SESSION` variable. Next, you execute that statement and attempt to get the first (and presumably only) row of data.

If a row exists, you set the status equal to the value of the `status` column. If not, you set the status equal to `NONE`. Finally, close the connection and return the value.

Place this function in the scripts.txt file so you can access it when you display the file list.

Approving the file: The form

Now you're ready to add approval capabilities to the form. What you want is to display a check box for pending files if the user viewing the list of files is an administrator. The `Content_Handler` class handles this display:

```
class Content_Handler{

  private $available = false;
  private $submittedBy = "";
  private $status = "";

  private $currentElement = "";

  private $fileId = "";
  private $fileName = "";
  private $fileSize = "";
  private $fileType = "";

  private $userStatus = "";

  function start_element($parser, $name, $attrs){


     if ($name == "workflow"){

        $this->userStatus = getUserStatus($_SESSION["username"]);

        if ($this->userStatus == "ADMIN"){
           echo "<form action='approve_action.php' method='POST'>";
        }

        echo "<h3>Available files</h3>";
        echo "<table width='100%' border='0'><tr>".
                "<th>File Name</th><th>Submitted By</th>".
                "<th>Size</th><th>Status</th>";
        if ($this->userStatus == "ADMIN"){
            echo "<th>Approve</th>";
        }
        echo "</tr>";
     }

     if ($name == "fileInfo"){
        if ($attrs['status'] == "approved" ||
                  $attrs['submittedBy'] == $this->username){
           $this->available = true;
        }
        if ($this->available){
           $this->submittedBy = $attrs['submittedBy'];
           $this->status = $attrs['status'];
           $this->fileId = $attrs['id'];
        }
     }

     $this->currentElement = $name;

  }

  function end_element($parser, $name){

     if ($name == "workflow"){
        echo "</table>";

        if ($this->userStatus == "ADMIN"){

           echo "<input type='submit' value='Approve Checked Files' />";

           echo "</form>";
        }

     }

     if ($name == "fileInfo"){
        echo "<tr>";
        echo "<td><a href='download_file.php?file=".
                       $this->fileName."&filetype=".
                       $this->fileType."'>".
                            $this->fileName."</a></td>".
                 "<td>".$this->submittedBy."</td>".
                 "<td>".$this->fileSize."</td>".
                 "<td>".$this->status."</td><td>";

        if ($this->userStatus == "ADMIN"){
           if ($this->status == "pending") {
              echo "<input type='checkbox' name='toapprove[]' value='".
                         $this->fileId."' checked='checked' />";
           }
        }

        echo "</td></tr>";

        $this->fileId = "";
        $this->fileName = "";
        $this->submittedBy = "";
        $this->fileSize = "";
        $this->status = "";
        $this->fileType = "";

        $this->available = false;
     }

     $this->currentElement = "";

  }

  function chars($parser, $chars){
...
  }
}
```

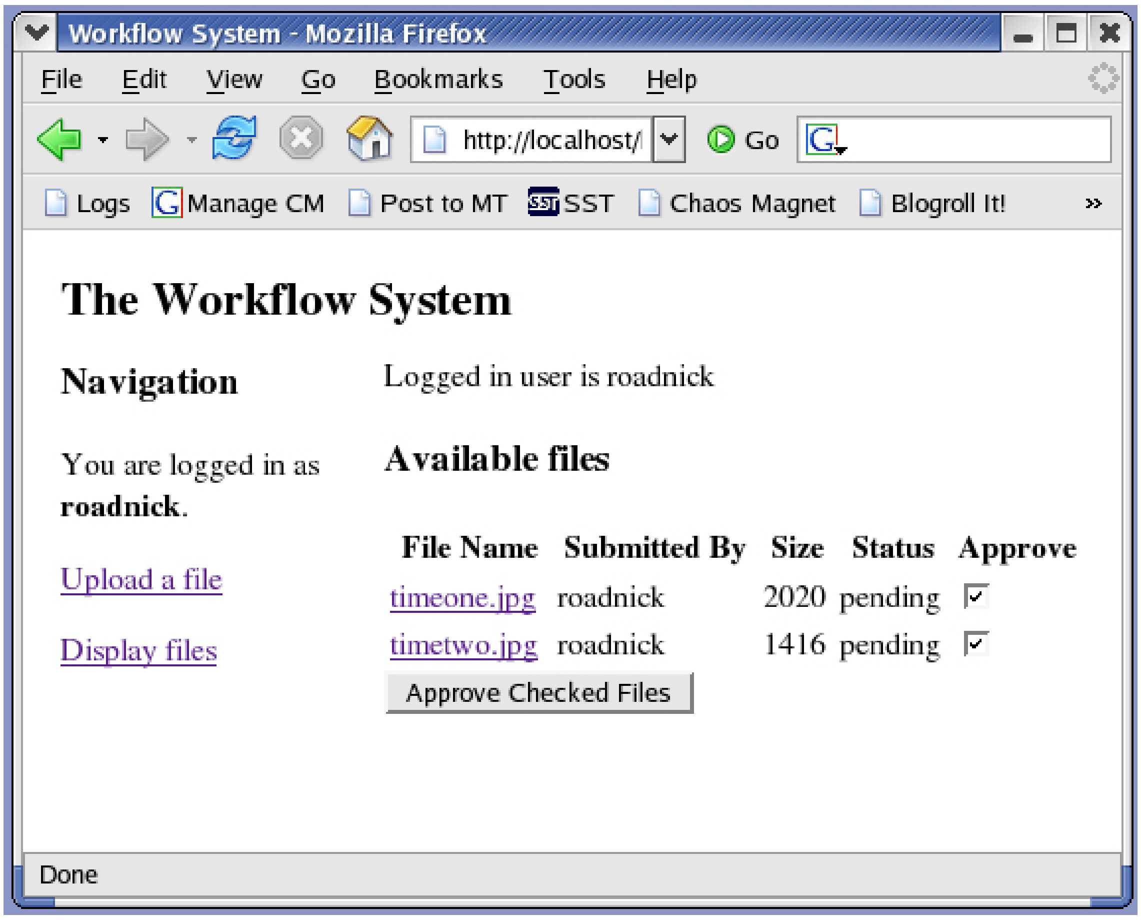
Starting at the top, you have two new properties to define: `$fileId` and `$userStatus`. The latter, you set once, when you process the start of the `workflow` element and, thus, the document. At this point, if the user *is* an administrator, you want to add a `form` element to the page and an `Approve` column to the table.

You close the form at the end of the document, when the content handler receives notification of the end of the `workflow` element.

As for the actual check boxes, you output those when you display the actual row of information, at the end of each `fileInfo` element. Because you have the potential for multiple entries, you name the field `toapprove[]`.

The result is a form with the appropriate boxes, as you can see in Figure 11.

Figure 11. The approval form



The Workflow System

Navigation

You are logged in as **roadnick**.

Upload a file

Display files

Logged in user is roadnick

Available files

File Name	Submitted By	Size	Status	Approve
timeone.jpg	roadnick	2020	pending	✓
timetwo.jpg	roadnick	1416	pending	✓

Approve Checked Files

Setting up IDs

Now you've got the form, but to access the `fileInfo` elements by their `id` attributes, you need to take one more step. Unlike in an HTML document, just naming an attribute "id" isn't enough to make it act like an identifier. In an XML file, you have to provide some sort of schema (note the small "s") that defines the attribute. In this case, you'll add a Document Type Definition (DTD). First, you add a reference to it in the actual document:

```
function save_document_info($fileInfo){

   $xmlfile = UPLOADEDFILES."docinfo.xml";

   if(is_file($xmlfile)){
      $doc = DOMDocument::load($xmlfile);
      $workflowElements = $doc->getElementsByTagName("workflow");
      $root = $workflowElements->item(0);

      $statistics = $root->getElementsByTagName("statistics")->item(0);
      $total = $statistics->getAttribute("total");
      $statistics->setAttribute("total", $total + 1);

   } else{

      $domImp = new DOMImplementation;
      $dtd = $domImp->createDocumentType('workflow', '', 'workflow.dtd');

      $doc = $domImp->createDocument("", "", $dtd);

      $root = $doc->createElement('workflow');
      $doc->appendChild($root);

      $statistics = $doc->createElement("statistics");
      $statistics->setAttribute("total", "1");
      $statistics->setAttribute("approved", "0");
      $root->appendChild($statistics);
   }
...
}
```

Instead of creating the document directly by instantiating the `DOMDocument` class, you create a `DOMImplementation`, from which you create a DTD object. You then assign that DTD to the new document you're creating.

If you remove the docinfo.xml file and upload a new document, you'll see the new information:

```
<?xml version="1.0"?>
<!DOCTYPE workflow SYSTEM "workflow.dtd">
<workflow><statistics total="3" approved="0"/>
...
```

Now you need to create the workflow.dtd file.

The DTD

Explaining all the nuances of XML validation is well beyond the scope of this tutorial, but you do need to have a DTD that describes the structure of the docinfo.xml file. To do that, create a file and save it as workflow.dtd in the same directory docinfo.xml is in. Add the following:

```
<!ELEMENT workflow (statistics, fileInfo*) >
<!ELEMENT statistics EMPTY>
<!ATTLIST statistics total CDATA #IMPLIED
                     approved CDATA #IMPLIED >
<!ELEMENT fileInfo (approvedBy, fileName, location, fileType, size)>
<!ATTLIST fileInfo id ID #IMPLIED>
<!ATTLIST fileInfo status CDATA #IMPLIED>
<!ATTLIST fileInfo submittedBy CDATA #IMPLIED>
<!ELEMENT approvedBy (#PCDATA)>
<!ELEMENT fileName (#PCDATA)>
<!ELEMENT location (#PCDATA)>
<!ELEMENT fileType (#PCDATA)>
<!ELEMENT size (#PCDATA)>
```

Simply put, you define each element and its "content" model. For example, the `workflow` element must have one `statistics` element child and zero or more `fileInfo` children.

You also define attributes and their types. For example, the `statistics` element has two optional attributes, `total` and `approved`, and they're both character data.

The key here is the definition of the `fileInfo` element's `id` value, which you've defined as type `ID`.

Now you can use this information.

Approving the file: Updating the XML

The actual form page that accepts the approval check boxes, approve_action.php, is very simple:

```
<?php

  include "../scripts.txt";

  $allApprovals = $_POST["toapprove"];
  foreach ($allApprovals as $thisFileId) {
     approveFile($thisFileId);
  }
  echo "Files approved.";

?>
```

For each `toapprove` check box, you simply call the `approveFile()` function, in scripts.txt:

```
function approveFile($fileId){

   $xmlfile = UPLOADEDFILES."docinfo.xml";

   $doc = new DOMDocument();
   $doc->validateOnParse = true;
   $doc->load($xmlfile);
```

```
      $statisticsElements = $doc->getElementsByTagName("statistics");
      $statistics = $statisticsElements->item(0);

      $approved = $statistics->getAttribute("approved");
      $statistics->setAttribute("approved", $approved+1);

      $thisFile = $doc->getElementById($fileId);
      $thisFile->setAttribute("status", "approved");

      $approvedByElements = $thisFile->getElementsByTagName("approvedBy");
      $approvedByElement = $approvedByElements->item(0);
      $approvedByElement->appendChild($doc->createTextNode($_SESSION["username"]));

      $doc->save($xmlfile);

   }
```

Before you even load the document, you specify that you want the parser to validate it or check it against the DTD. This sets the nature of the `id` attribute. Once you've loaded the file, you get a reference to the `statistics` element so you can increment the number of approved files.

Now you're ready to actually approve the file. Because you set the `id` attribute as an `ID`-type value, you can use `getElementById()` to request the appropriate `fileInfo` element. When you have that element, you can set its `status` to `approved`.

You also need to get a reference to this element's `approvedBy` child. When you have that reference, you can add a new `Text` node child with the administrator's username.

Finally, save the file.

Note that while you did it this way for simplicity's sake, in a production application, it's more efficient to open and load the file just once, make all the changes, then save the file.

Security checks on download

As the last step, you add a security check to the download process. Because you control this process entirely through the application, you can use whichever checks you want. For this example, you'll check to make sure that the user clicked the link for a file on a page that is on your local server, preventing someone from linking to it from an external site, or even from bookmarking the link or sending someone else a raw link.

You start by creating a new exception, just for this occasion, in the WFDocument.php file:

```
   <?php
```

```
     include_once("../scripts.txt");

  class NoFileExistsException extends Exception {

     public function informativeMessage(){
        $message = "The file, '".$this->getMessage()."', called on line ".
             $this->getLine()." of ".$this->getFile().", does not exist.";
        return $message;
     }

  }

  class ImproperRequestException extends Exception {

     public function logDownloadAttempt(){
        //Additional code here
        echo "Notifying administrator ...";
     }

  }

  class WFDocument {

     private $filename;
     private $filetype;

     function setFilename($newFilename){
        $this->filename = $newFilename;
     }
     function getFilename(){
        return $this->filename;
     }

     function setFiletype($newFiletype){
        $this->filetype = $newFiletype;
     }
     function getFiletype(){
        return $this->filetype;
     }

     function __construct($filename = "", $filetype = ""){
        $this->setFilename($filename);
        $this->setFiletype($filetype);
     }

     function download() {

        $filepath = UPLOADEDFILES.$this->filename;

        try {

           $referer = $_SERVER['HTTP_REFERER'];
           $noprotocol = substr($referer, 7, strlen($referer));
           $host = substr($noprotocol, 0, strpos($noprotocol, "/"));
           if ( $host != 'boxersrevenge' &&
                                  $host != 'localhost'){
              throw new ImproperRequestException("Remote access not allowed.
                          Files must be accessed from the intranet.");
           }

           if(file_exists($filepath)){
             if ($stream = fopen($filepath, "rb")){
                $file_contents = stream_get_contents($stream);
                header("Content-type: ".$this->filetype);
                print($file_contents);
             } else {
                throw new Exception ("Cannot open file ".$filepath);
             }
           } else {
             throw new NoFileExistsException ($filepath);
           }
        } catch (ImproperRequestException $e){

           echo "<p style='color: red'>".$e->getMessage()."</p>";
           $e->logDownloadAttempt();

        } catch (Exception $e){

           echo "<p style='color: red'>".$e->getMessage()."</p>";

        }
     }
  }

  ?>
```

First off, in the `ImproperRequestException`, you create a new method, `logDownloadAttempt()`, that can send an e-mail or perform some other action. You use that method in this exception type's `catch` block.

In the actual `download()` function, the first thing you do is get the `HTTP_REFERER`. This optional header is sent with a Web request identifying the page from which the request was made. For example, if you link to *developerWorks* (http://www.ibm.com/developerworks) from your blog, and you click that link, the IBM logs would show the URL of your blog as the `HTTP_REFERER` for that access.

In your case, you want to make sure the request is coming from your application, so you first strip off the "http://" string at the beginning, then save all the text up to the first slash (/). This is the hostname in the request.

For an external request, this hostname might be something along the lines of boxersrevenge.nicholaschase.com, but you're looking for only internal requests, so you accept `boxersrevenge` or `localhost`. If the request comes from anywhere else, you throw the `ImproperRequestException`, which is caught by the appropriate block.

Note that this method is not foolproof as far as security is concerned. Some browsers don't send referrer information properly because either they don't support it or the user has altered what's being sent. But this example should give you an idea of the types of things you can do to help control your content.

Section 8. Summary and resources

Summary

This tutorial wrapped up the three-part series on "Learning PHP" while you built a simple workflow application. Earlier parts focused on the basics, such as syntax, form handling, database access, file uploading, and XML. In this part, you took all that a step further and put it together by creating a form through which an administrator could approve various files. We discussed the following topics:

- Using HTTP authentication
- Streaming files
- Creating classes and objects
- Object properties and methods
- Using object constructors
- Using exceptions
- Creating custom exceptions
- Using XML ID attributes
- Performing additional security checks for downloads

Resources

Access parts 1 and 2 of this series: "*Learning PHP, Part 1*" and "*Learning PHP, Part 2*."

In these tutorials, you've just scratched the surface of what you can do with PHP. Following are some good places to go for more information:

- *PHP Manual* (http://www.php.net/manual/en)
- *PHP Manual: Classes and Objects (PHP 4)* (http://www.php.net/manual/en/language.oop.php)
- *PHP Manual: Classes and Objects (PHP 5)* (http://www.php.net/manual/en/language.oop5.php)
- *PHP Manual: Exceptions*
- *PHP Manual: Security* (http://www.php.net/manual/en/security.php)
- *PHP Manual: HTTP authentication with PHP*
- *PHP Manual: DOM Functions* (http://www.php.net/manual/en/ref.dom.php)
- *PHP Manual: Stream Functions* (http://www.php.net/manual/en/ref.stream.php)
- *PHP Manual: String Functions* (http://www.php.net/manual/en/ref.strings.php)
- *PHP Manual: Streams API for PHP Extension Authors*

(http://www.php.net/manual/en/streams.php)
- *Apache HTTP Server Version 1.3 Authentication, Authorization, and Access Control* (http://httpd.apache.org/docs/howto/auth.html)
- *Apache HTTP Server Version 2.0 Authentication, Authorization and Access Control* (http://httpd.apache.org/docs-2.0/howto/auth.html)
- *Apache HTTP Server Version 2.0 tutorial: .htaccess files*
- *HTTP Authentication and WebSphere Application Server 4 Web applications* (developerWorks)
- Visit the developerWorks *Open source zone* (http://www.ibm.com/developerworks/opensource) for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM products.
- Innovate your next open source development project with *IBM trial software*, available for download or on DVD.
- Get involved in the developerWorks community by participating in *developerWorks blogs* (http://www.ibm.com/developerworks/blogs/) .

Feedback

Please let us know whether this tutorial was helpful to you and how we could make it better. We'd also like to hear about other tutorial topics you'd like *developerWorks* to cover.

For questions about the content of this tutorial, contact the authors, Nicholas Chase, at: *ibmquestions@nicholaschase.com*, or Tyler Anderson, at: *tyleranderson5@yahoo.com*.

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

For more information about the Toot-O-Matic, visit www-106.ibm.com/developerworks/xml/library/x-toot/ .