Learning PHP, Part 3: Authentication, objects, exceptions, and streaming

Nicholas Chase
(ibmquestions@nicholaschase.com)
Founder
NoTooMi

Skill Level: Intermediate

Date: 12 Jul 2005
(Updated 03 Jan 2013)

This tutorial is Part 3 of a three-part "Learning PHP" series teaching you how to use PHP through building a simple workflow application. In this tutorial, you will learn about using HTTP authentication, streaming files, and how to create objects and exceptions.
03 Jan 2013 - *Nicholas Chase updated content throughout this tutorial to reflect current PHP technology.*

View more content in this series

Section 1. Before you start

In this tutorial you will learn how to use HTTP authentication, streaming files, and how to create objects and exceptions in PHP.

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
- 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.

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.

Prerequisites

You need to have a web server, PHP, and a database installed and available. If you have a hosting account, you can use it as long as the server has PHP V5 installed and has access to a MySQL database. Otherwise, download and install the following packages:

XAMPP

Whether you're on Windows, Linux, or even Mac, the easiest way to get all of the necessary pieces of software for this tutorial is to install XAMPP, which includes a web server, PHP, and the MySQL database engine. If you choose to go this route, install and then run the control panel to start up the Apache and MySQL processes. You also have the option of installing the various pieces separately. Keep in mind that you will have to configure them to work together—a step already completed with XAMPP.

Web server

If you choose not to use XAMPP, you have several options for a web server. If you use PHP 5.4 (as of this writing, XAMPP is only using PHP 5.3.8) you can use the built-in web server for testing. For production, however, I assume that you're using the Apache Web server, version 2.x.

PHP 5.x

If you do not use XAMPP, you need to download PHP 5.x separately. The standard distribution includes everything you need for this tutorial. Feel free to download the binaries; you d0 not need the source for this tutorial (or ever, unless you want to hack on PHP itself). This tutorial was written and tested on PHP 5.3.8.

MySQL

Part of this project involves saving data to a database, so you'll need one of those, as well. Again, if you install XAMPP, you can skip this step, but if you choose to, you can install a database separately. In this tutorial, I concentrate on MySQL because it's so commonly used with PHP. If you choose to go this route, you can download and install the Community Server.

Section 2. The story so far

In this section I review the progress you've made in this series, create a welcome page, and create some restrictions using PHP.

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, email 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 through 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 uses an alternative format, JavaScript Object Notation (JSON) to both save and display the data.

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—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 a non-web-accessible location.

- Ensure 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.
- Manage the approval process.

To start, you'll put 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 code from Listing 1.

Listing 1. The index page

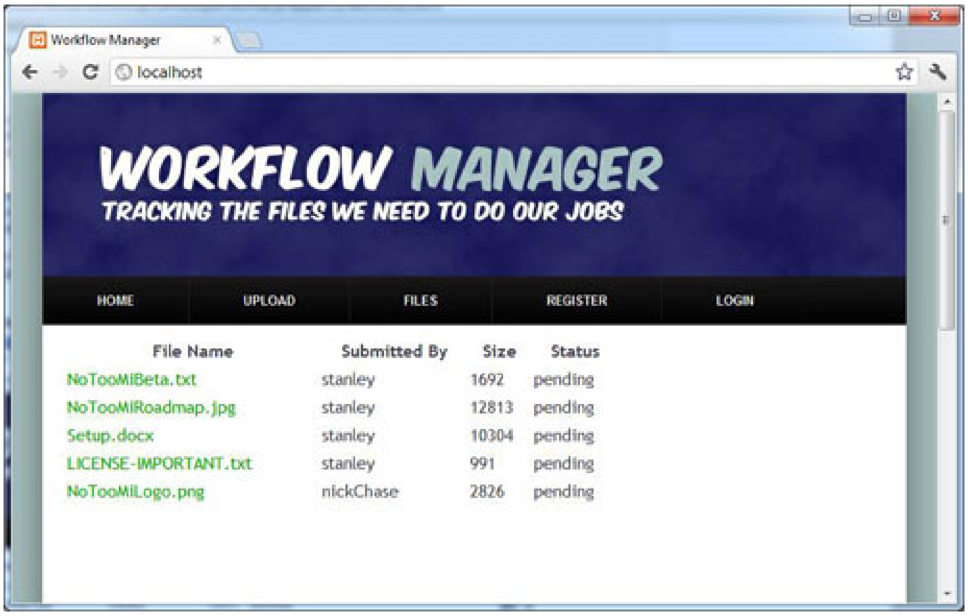
```
<?php

   session_start();

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

   display_files();

   include ("bottom.txt");

?>
```

After starting the session so it is available later, the page's first `include()` function loads the top interface elements for the page, if applicable. The second loads all the scripts that you 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 the bottom of the HTML page.

Save the file in the same directory as the other files that you created. For example, you might put the file in the document root of your server. Once you start the HTTP server, you can see the page by pointing your browser to http://localhost/index.php.

Figure 1 shows the simple page.

Figure 1. The basic listing page



File Name	Submitted By	Size	Status
NoTooMiBeta.txt	stanley	1692	pending
NoTooMiRoadmap.jpg	stanley	12813	pending
Setup.docx	stanley	10304	pending
LICENSE-IMPORTANT.txt	stanley	991	pending
NoTooMiLogo.png	nickChase	2826	pending

Restricting file access

In the next section, you learn to control who sees what with authentication. You first need to put some restrictions in place. At this point, all users can see all files, whether they're approved or not, and that's not what you want. Instead, you want `display_files()` to show users only files that are approved, unless a user is the one who uploaded it.

Open `scripts.txt` and make the additions in Listing 2.

Listing 2. Restricting access to files

```
    for ($i = 0; $i < count($workflow["fileInfo"]); $i++) {
        $thisFile = $workflow["fileInfo"][$i];
        if (
            ($thisFile["approvedBy"] != null) ||
            (
                    isset($_SESSION["username"]) &&
                    ($thisFile["submittedBy"] == $_SESSION["username"])
            )
        ) {

            echo "<tr>";
            echo "<td>" . $thisFile["fileName"] . "</td>";
            echo "<td>" . $thisFile["submittedBy"] . "</td>";
            echo "<td>" . $thisFile["size"] . "</td>";
            echo "<td>" . $thisFile["status"] . "<td>";
            echo "</tr>";
        }
    }
```

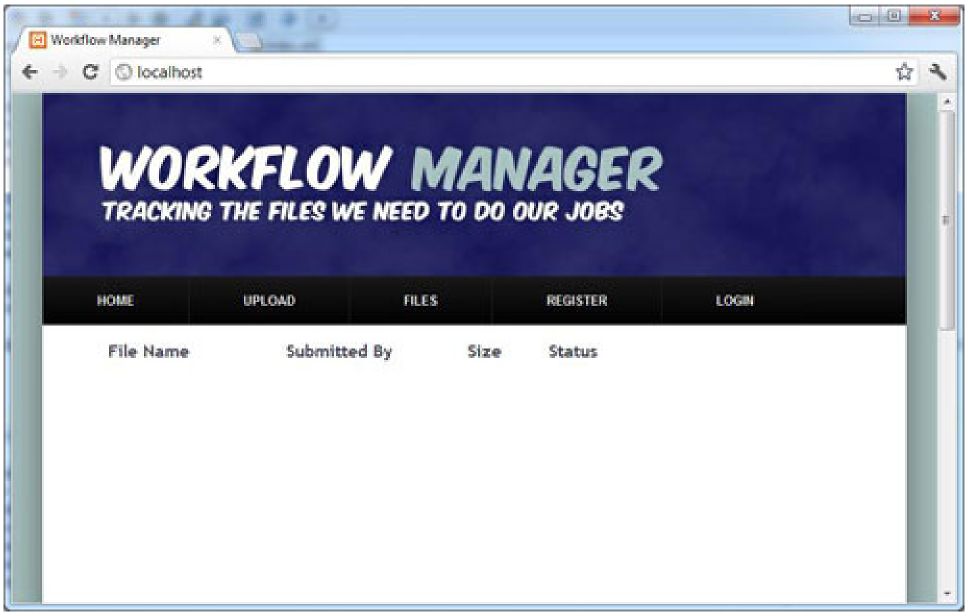
In Listing 2, you combine three different conditions to determine whether to list a particular file. First, if a file is approved, then `$thisFile["approvedBy"]` will have a value, so that condition is true. The double-pipe (`||`) means "or," so if the first test comes up false, you get a second chance with the second half of the condition.

The second half of the condition also consists of two parts, but because you use the double ampersand (`&&`)—which means "and"—they both need to be true for the second half to be true. The first test is to see if the session knows about the username. If it does, the username has to match the `$thisFile["submittedBy"]` value.

If the overall condition evaluates to "true"—in other words, if the file is approved, or if the user is logged in and is the originator of the file—the system displays it. If not, it doesn't.

So if you're not logged in (you might have to restart your browser to test this) you should see an empty page, as in Figure 2.

Figure 2. The basic listing page, with restrictions



If you just started 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

In this section, you'll set up the server for HTTP authentication, so your web server can control the login process for the PHP application.

HTTP authentication

Up to now, you 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 log 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 will switch 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 start, be aware that if you use a server other than Apache 2.X, you must check the documentation for HTTP authentication to learn how to set it up. XAMPP uses HTTP authentication, so if you use XAMPP, you're all set. (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. 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 create a /usr/local/apache2/htdocs/loggedin directory.

Now you need to tell the server that you want to override the overall security for that directory, so open the httpd.conf file and add the code in Listing 3 to it.

Listing 3. Overriding security for a directory

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

(You should 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 code in Listing 4 to it.

Listing 4. Creating the .htaccess file

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

In Listing 4, 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. (If you create a high-security application you'll want to investigate other options.) 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 say 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 XAMPP, you can do this from the XAMPP control menu, or from the Services control panel, if you've set it up as a service. For all installations of Apache V2.0, you can also call `<APACHE_HOME>/bin/apachectl stop`, followed by `<APACHE_HOME>/bin/apachectl start`.)

Next, create the password file.

Creating the password file

For all this to work, you need to have a password file that the server can check. In Adding new users to the password file, 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, you need the htpasswd application, which comes with Apache. If you use XAMPP, look for it in `<XAMPP_HOME>/apache/bin`. Execute the following command in Listing 5, 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 Listing 5.

Listing 5. Creating the .htpasswd file

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

The `-c` switch tells the server to create a new file, so after you add the new user, the file looks something like this: `NickChase:IpoRzCGnsQv.Y`.

Note that this version of the password is encrypted, and you must 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.

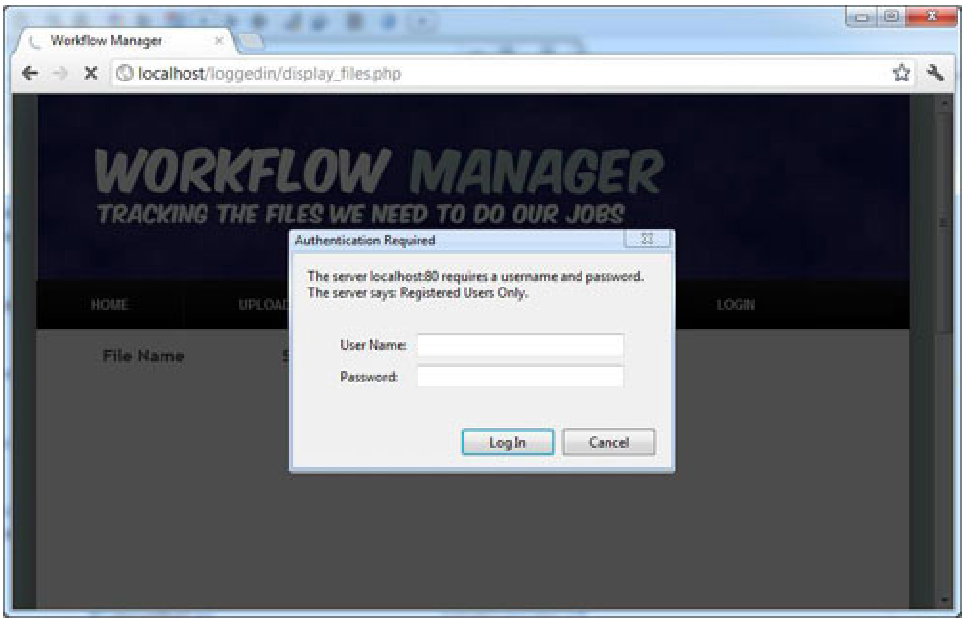
In each of the three files, change the `include()` statements to account for the new location, as in Listing 6.

Listing 6. Displaying files

```
<?php

   session_start();
   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 3, you should get a username and password box.

Figure 3. Username and password box



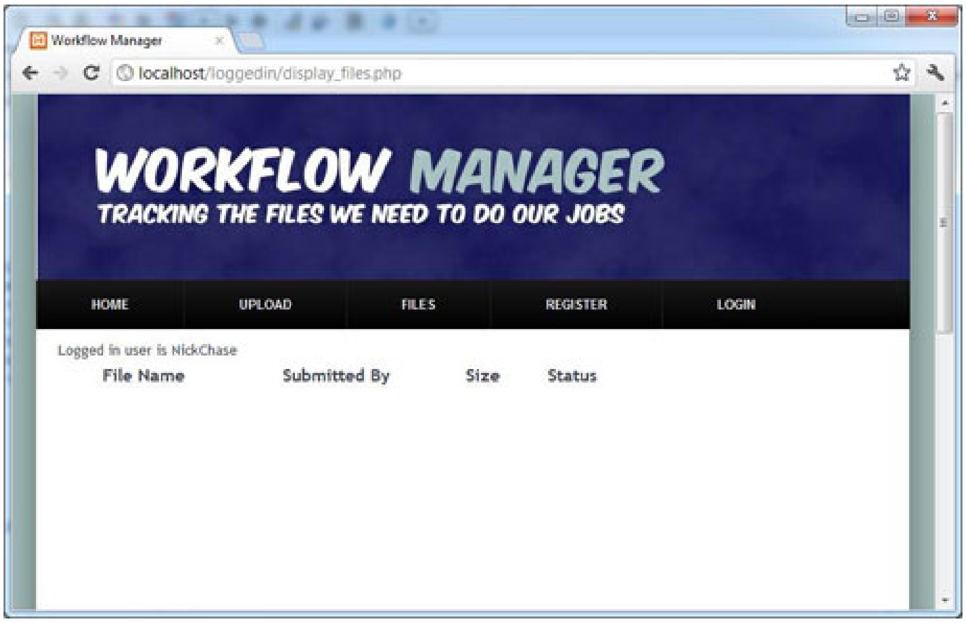
The server localhost:80 requires a username and password.
The server says: Registered Users Only.

Enter the username and password you used in Creating the password file 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. As Figure 3 shows, while the message says 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, as you can see in Figure 4.

Figure 4. Logged in ... sort of


Workflow Manager
localhost/loggedin/display_files.php
WORKFLOW MANAGER
TRACKING THE FILES WE NEED TO DO OUR JOBS
HOME
UPLOAD
FILES
REGISTER
LOGIN

Logged in user is NickChase

File Name	Submitted By	Size	Status

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 (see Listing 7).

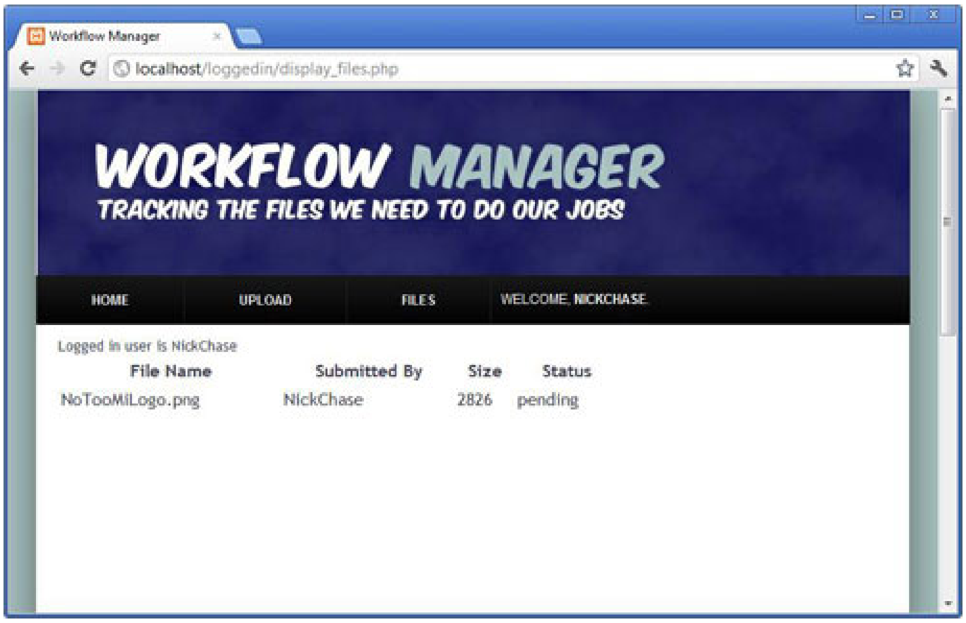
Listing 7. Setting the current user

```
<?php

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

?>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
  <head>
...
```

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 5.

Figure 5. The corrected page


Workflow Manager
localhost/loggedin/display_files.php
WORKFLOW MANAGER
TRACKING THE FILES WE NEED TO DO OUR JOBS
HOME
UPLOAD
FILES
WELCOME, NICKCHASE.

Logged in user is NickChase

File Name	Submitted By	Size	Status
NoTooMiLogo.png	NickChase	2826	pending

Fixing the interface

Before you add 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 (see Listing 8).

Listing 8. Adjusting the navigation

```
...
<div id="nav1">
   <ul style='float: left'>
      <li><a href="#" shape="rect">Home</a></li>
      <li><a href="/uploadfile.php" shape="rect">Upload</a></li>
      ><li><a href="/loggedin/display_files.php" shape="rect">Files
</a></li>>
       <?php
          if (isset($_SESSION["username"]) || isset($username)){
             if (isset($_SESSION["username"])){
                $usernameToDisplay =  $_SESSION["username"];
             } else {
                $usernameToDisplay = $username;
             }
      ?>
      ><!--> <li><a href="logout.php" shape="rect">Logout</a>
</li>   -->
            <li><p style='color:white;'>&nbsp;&nbsp;&nbsp;
Welcome,
                   <b><?=$usernameToDisplay?></b>.</p></li>
       <?php
          } else {
       ?>
            <li><a href="/registration.php" shape="rect">
Register</a></li>
            <li><a href=">/loggedin/display_files.php"
shape="rect">Login</a></li>
       <?php
          }
       ?>
   </ul>
</div>
...
```

Notice that in addition to fixing the login reference and adding a new option for displaying the list of files, I commented out the message about logging out, 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 to .htpasswd once you save the user to the database. Open registration_action.php and add the contents of Listing 9.

Listing 9. Creating the user on the server at registration

```
...
    if ($checkUserStmt->rowCount() == 0) {

        $stmt = $dbh->prepare("insert into users (username, email, password) ".
                                                                  "values (?, ?, ?)");

        $stmt->bindParam(1, $name);
        $stmt->bindParam(2, $email);
        $stmt->bindParam(3, $pword);

        $name = $_POST["name"];
        $email = $_POST["email"];
        $pword = $passwords[0];

        $stmt->execute();

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

        echo "<p>Thank you for registering!</p>";

    } else {

        echo "<p>There is already a user with that name: </p>";
...
```

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

First, check to see if the file exists and use that information to determine whether you will 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, 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 finish 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

Now that you've set up the system\, you're ready to enable the user to actually download the available files. From the very beginning, 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 will create a function that streams the file from its current location to the browser.

What are streams?

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 through HTTP or FTP.

Fortunately, PHP provides *stream wrappers*. 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, as in Listing 10.

Listing 10. Displaying available stream wrappers

```
<?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 Listing 11.

Listing 11. Available stream wrappers

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

This enables 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. The code you use in this section will still work.

Let's take a look.

Downloading the file

For the user 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 code in Listing 12.

Listing 12. Sending the file

```
<?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 actually do with the `fopen()` function is create 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`.

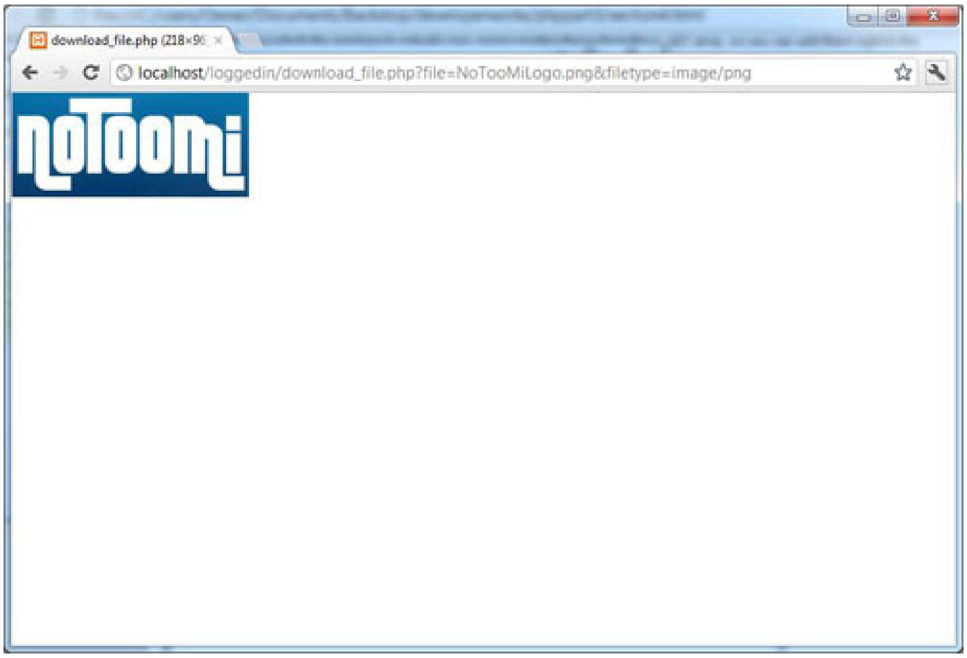
Finally, you output the contents of the file to the browser. Having received the `Content-type` header, the browser will know how to treat it.

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=NoTooMiLogo.png&filetype=image/png
```

Enter this URL (with an appropriate file name and type, of course) into your browser to see the results in Figure 6.

Figure 6. Downloading a file


download_file.php (218×96
localhost/loggedin/download_file.php?file=NoTooMiLogo.png&filetype=image/png
noToomi

Adding a link to the file

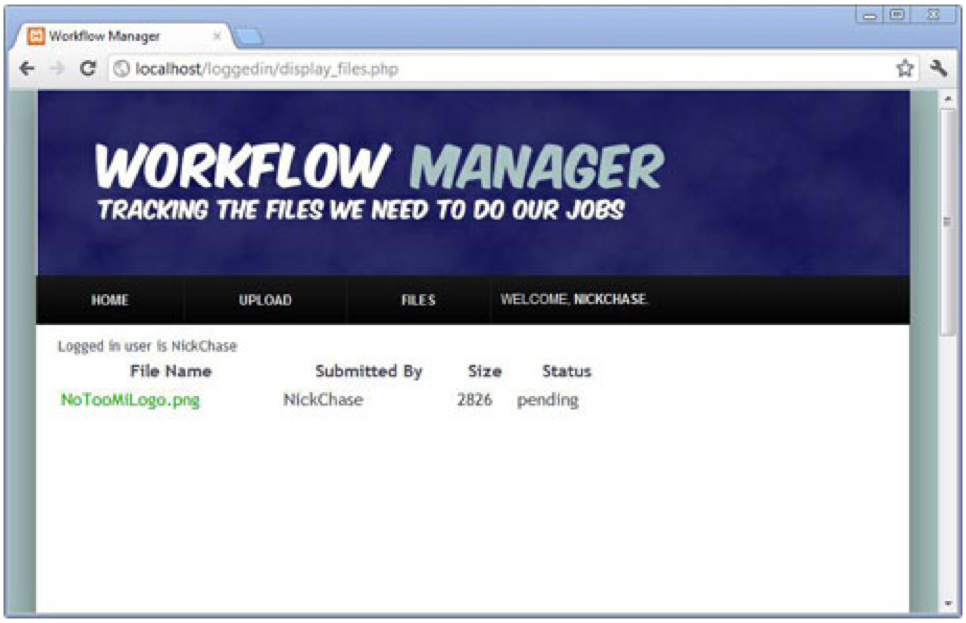
Because all the information the download page needs can be added to the URL, it's simple to add a link enabling the user to download a file. You create the display of available files using the `display_files()` function, so you can add the link as in Listing 13.

Listing 13. Adding the link

```
...
for ($i = 0; $i < count($workflow["fileInfo"]); $i++){
    $thisFile = $workflow["fileInfo"][$i];
    echo "<tr>";
    echo "<td><a href='/loggedin/download_file.php?file="
.$thisFile["fileName"].
            "&filetype=".$thisFile["fileType"]."'>".$thisFile["fileName"]
."</a></td>";
    echo "<td>".$thisFile["submittedBy"]."</td>";
    echo "<td>".$thisFile["size"]."</td>";
    echo "<td>".$thisFile["status"]."<td>";
    echo "</tr>";

}
...
```

You can see the results in Figure 7.

Figure 7. Linking to the file


Workflow Manager
localhost/loggedin/display_files.php
WORKFLOW MANAGER
TRACKING THE FILES WE NEED TO DO OUR JOBS
HOME
UPLOAD
FILES
WELCOME, NICKCHASE

Logged in user is NickChase

File Name	Submitted By	Size	Status
NoTooMiLogo.png	NickChase	2826	pending

Click a link to verify the file.

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

Section 5. Using objects

In this section, you explore 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 will move away from that.

What are objects, anyway?

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, I 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 code in Listing 14.

Listing 14. The basic document Object

```
<?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 actually already instantiated several objects when you worked with DOM in Part 2 of this series, but little was said about why or how. I will remedy that now.

Open the download_file.php page and change the code so it reads as in Listing 15.

Listing 15. Sending the file information to the function

```
<?php

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

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

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

?>
```

First, rather than include the scripts.txt file, you include 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 call it.

Creating properties

Methods are only part of the story. The whole point of an object is that it's encapsulated. It should contain all its own information, so rather than feed the name and file type to the `download()` method, you can set them as properties on the object. First you have to create them in the class (see Listing 16).

Listing 16. Using object properties

```
<?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 also declare 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. I will talk more about this concept, *inheritance*, in Creating a custom exception.)

Finally, to reference an object property, you must know which object owns the property. 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 pass information to an object, you want to actually set the properties of the object (see Listing 17).

Listing 17. Setting object properties

```
<?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 are 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 section, 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, as in Listing 18.

Listing 18. Using private properties

```
<?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 did earlier, 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 (see Listing 19).

Listing 19. Using setters

```
<?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

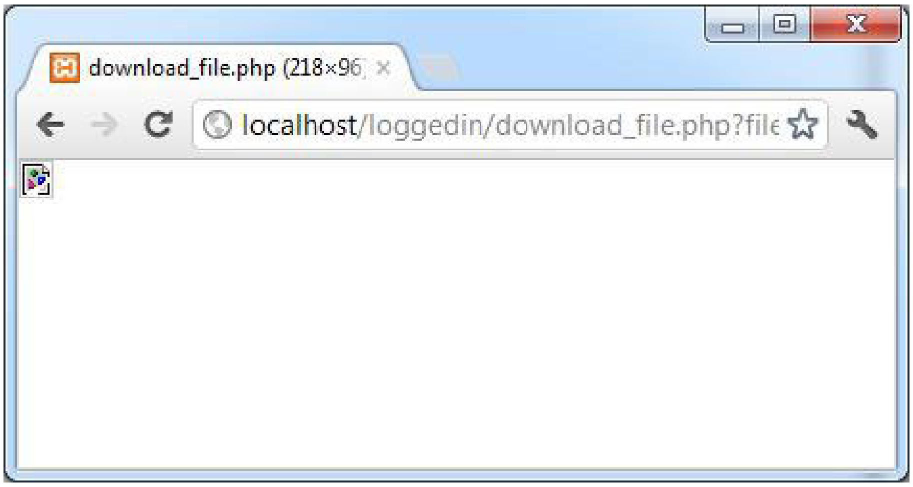
If an object has a constructor, it gets called every time you create a new instance of that particular class. For example, you can create a simple constructor, as in Listing 20.

Listing 20. 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 8.

Figure 8. Error after running script


download_file.php (218×96
localhost/loggedin/download_file.php?file

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 (see Listing 21).

Listing 21. A more complex constructor

```
...
   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 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 (see Listing 22).

Listing 22. Using a 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

An exception is what happens when something unexpected occurs in a program. When an exception occurs, a program is frequently designed to stop or display errors. 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.

A generic exception

Let's start with a simple generic exception in the definition of the `WFDocument` class (see Listing 23).

Listing 23. Throwing an exception

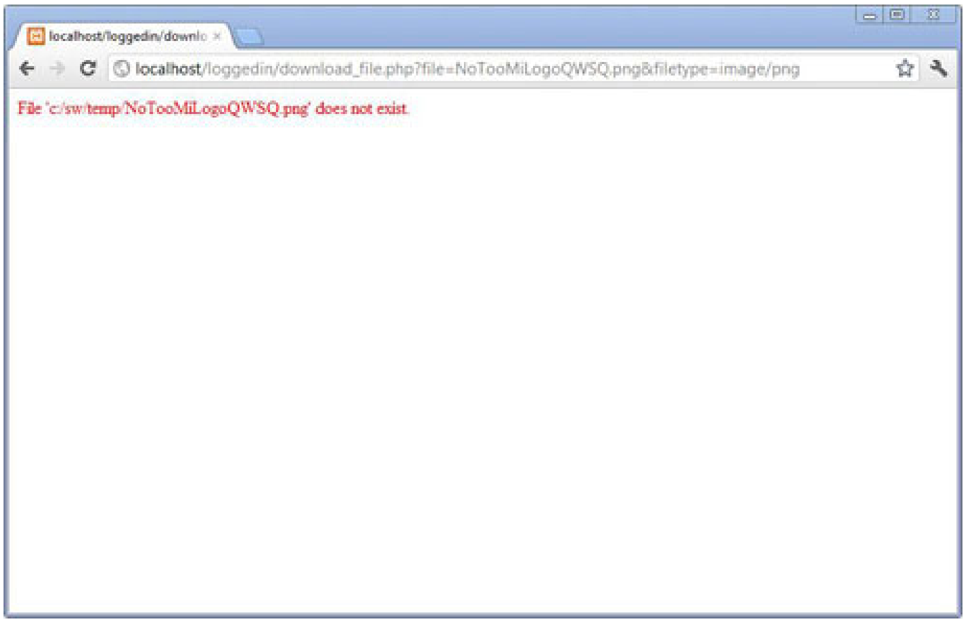
```
<?php

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>";

      }
   }
}

?>
```

Exceptions don't just happen, they are *thrown*. 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 `File 'c:/sw/temp/NoTooMiLogoQWSQ.png' does not exist.` (see Figure 9).

Figure 9. The basic exception


localhost/loggedin/downlo
localhost/loggedin/download_file.php?file=NoTooMiLogoQWSQ.png&filetype=image/png

File 'c:/sw/temp/NoTooMiLogoQWSQ.png' 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 I left out one very important aspect of them: inheritance.

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 (see Listing 24).

Listing 24. Creating a custom exception

```
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, I added this code to the WFDocument.php file, but you can add it wherever it's accessible when you need it.)

Here, you 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

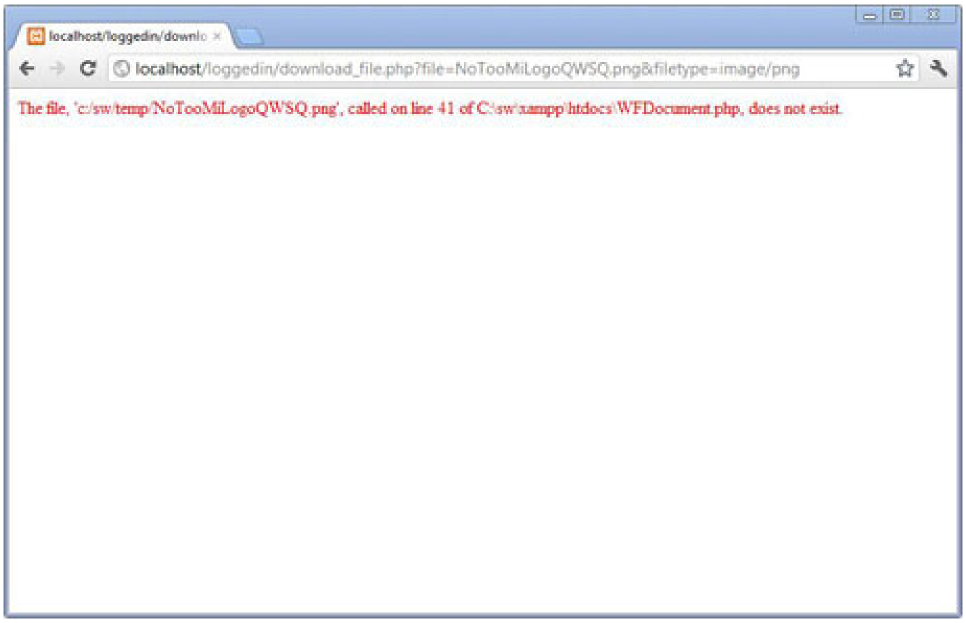
The easiest way to use the new exception type is to simply throw it just as you would throw a generic `Exception` (see Listing 25).

Listing 25. Throwing and catching a custom 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, the full message is returned: `The file, 'c:/sw/temp/NoTooMiLogoQWSQ.png', called on line 41 of C;\sw\xampp\htdocs\WFDocument.php, does not exist.` (see Figure 10).

Figure 10. Using a custom exception


localhost/loggedin/downlo
localhost/loggedin/download_file.php?file=NoTooMiLogoQWSQ.png&filetype=image/png

The file, 'c:/sw/temp/NoTooMiLogoQWSQ.png', called on line 41 of C:\sw\xampp\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` (see Listing 26).

Listing 26. Distinguishing between exceptions

```
...
   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.

You can also use exceptions to trap for situations that technically are errors, but shouldn't actually stop your program. For example, you might attempt to process an image, and if it's not successful, leave it as it is and move on rather than exiting.

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

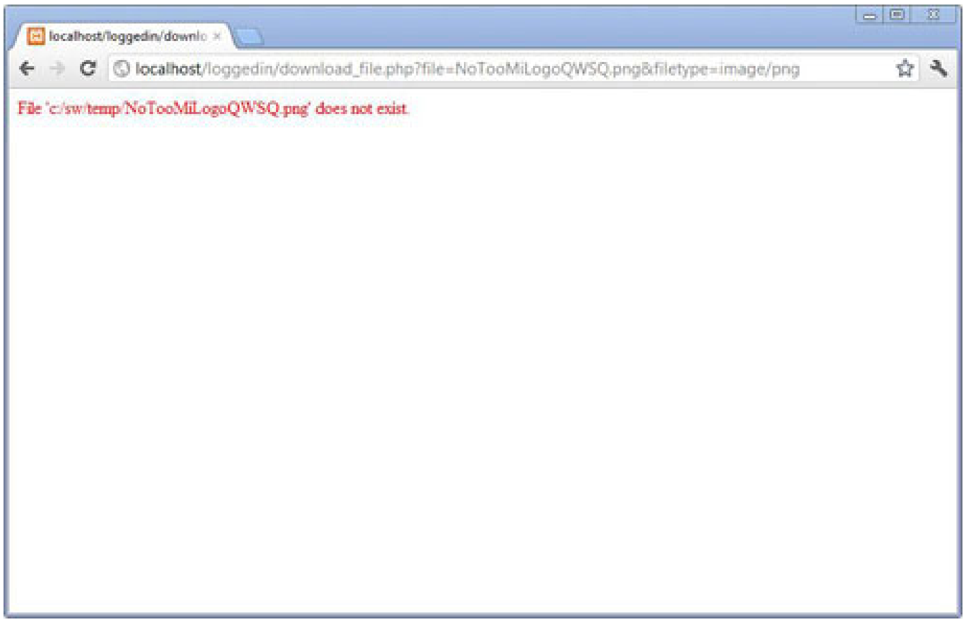
Another handy feature of inheritance is the ability to treat an object as though it were a member of its base class. For example, you can throw a `NoFileExistsException` and catch it as a generic `Exception` (see Listing 27).

Listing 27. Combining exception catching

```
...
   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 you can see by the `File 'c:/sw/temp/NoTooMiLogoQWSQ.png' does not exist.` message in Figure 11.

Figure 11. Propagating exceptions



localhost/loggedin/downlo
localhost/loggedin/download_file.php?file=NoTooMiLogoQWSQ.png&filetype=image/png

File 'c:/sw/temp/NoTooMiLogoQWSQ.png' does not exist

Section 7. Putting it together

Now that you have the file download process in place, it's time to put everything together and finish off the application. In this section, you will take care of some uncompleted miscellaneous tasks:

- 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

First you need to set up the administrators who will approve files.

Detecting administrators

When you originally created the users table in the database, you didn't take into consideration the fact that you need 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 status of the current user. Add this function to scripts.txt as shown in Listing 28.

Listing 28. Detecting user status

```
function getUserStatus()
{
    $dbh = new PDO('mysql:host=localhost;dbname=workflow', 'wfuser', 'wfpass');
    $stmt = $dbh->prepare("select * from users where username= :username");

    $stmt->bindParam("username", $username);

    $username = $_SESSION["username"];

    $stmt->execute();

    $status = "NONE";
    if ($row = $stmt->fetch()) {
        $status = $row["status"];
    }

    $dbh = null;

    return $status;
}
```

To review how this process works, you create a connection to the appropriate database. Then prepare a SQL statement using a parameter for the username. Set that parameter to the actual username, stored in the `$_SESSION` variable. Next, execute that statement and attempt to get the first (and presumably only) row of data.

You start by defining the `$status` as `NONE`. If no row exists, this variable will simply stay as it is. On the other hand, if a row exists, you set the status equal to the value of the `status` column. Finally, close the connection and return the value.

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 `display_files()` function in scripts.txt handles that (see Listing 29).

Listing 29. Adding admin functions

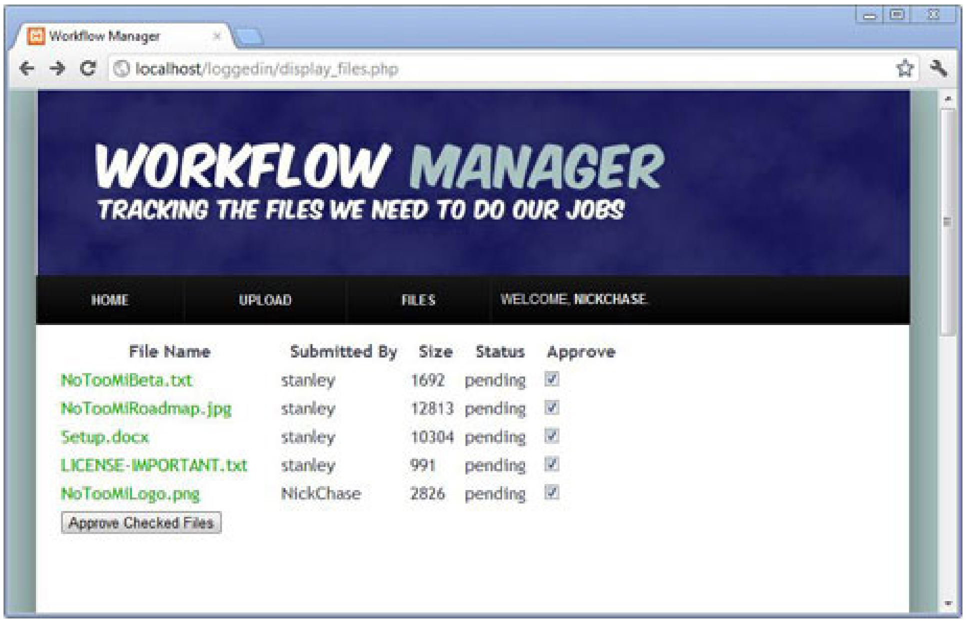
```
function display_files()
{

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

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

    $workflow = json_decode(file_get_contents(UPLOADEDFILES . "docinfo.json"), true);

    echo "<table width='100%'>";

    $files = $workflow["fileInfo"];

    echo "<tr><th>File Name</th>";
    echo "<th>Submitted By</th><th>Size</th>";
    echo "<th>Status</th>";
    if ($userStatus == "ADMIN") {
        echo "<th>Approve</th>";
    }
    echo "</tr>";

    for ($i = 0; $i < count($workflow["fileInfo"]); $i++) {
        $thisFile = $workflow["fileInfo"][$i];
        if (
            >($userStatus == "ADMIN") ||>
            ($thisFile["approvedBy"] != null) ||
            (
                    isset($_SESSION["username"]) &&
                    ($thisFile["submittedBy"] == $_SESSION["username"])
            )
        ) {

            echo "<tr>";
            echo "<td><a href='/loggedin/download_file.php?file=" .
                    $thisFile["fileName"] . "&filetype=" . $thisFile["fileType"] .
                    "'>" . $thisFile["fileName"] . "</a></td>";
            echo "<td>" . $thisFile["submittedBy"] . "<lt;/td>";
            echo "<td>" . $thisFile["size"] . "</td>";
            echo "<td>" . $thisFile["status"] . "<td>";
            if ($userStatus == "ADMIN") {
                if ($thisFile["status"] == "pending") {
                    echo "<input type='checkbox' name='toapprove[]' ".
                               "value='" . $i . "' checked='checked' />";
                }
            }
            echo "</tr>";
        }
    }

    echo "</table>";
    if ($userStatus == "ADMIN") {
        echo "<input type='submit' value='Approve Checked Files' />";
        echo "</form>";
    }
}
```

Starting at the top, you first determine the user's status. This is important for two reasons. For one thing, if a user is an administrator, you need to display the approval form. For another, if a user is an administrator, you'll display all files, no matter who uploaded them or what their status is. To do that, you just add another condition to the `if` statement.

Within the actual file display, if the user is an administrator, **and** if the file is still pending, you display a pre-checked checkbox. The value of the checkbox is the number of the file, so you can refer to it later. Giving the user an array-like name (in this case `toapprove[]` tells the web server to expect multiple values for the same field name.

Figure 12 shows the result, a form with the appropriate fields (filename, submitter, file size, status) and check boxes to select for approval.

Figure 12. The approval form



File Name	Submitted By	Size	Status	Approve
NoTooMiBeta.txt	stanley	1692	pending	✓
NoTooMiRoadmap.jpg	stanley	12813	pending	✓
Setup.docx	stanley	10304	pending	✓
LICENSE-IMPORTANT.txt	stanley	991	pending	✓
NoTooMiLogo.png	NickChase	2826	pending	✓

Approving the file: Updating the JSON

The actual form page that accepts the approval check boxes, approve_action.php, is very simple (see Listing 30).

Listing 30. Processing the approval form

```
<?php

  session_start();

  include "/scripts.txt";

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

?>
```

For each `toapprove` check box, you simply call the `approveFile()` function, in scripts.txt (see Listing 31).

Listing 31. Approving a form

```
function approveFile($fileNumber){

    $workflow = json_decode(file_get_contents(UPLOADEDFILES . "docinfo.json"), true);

    $workflow["fileInfo"][$fileNumber]["approvedBy"] = $_SESSION["username"];
    $workflow["fileInfo"][$fileNumber]["status"] = "approved";

    $jsonText = json_encode($workflow);
    file_put_contents(UPLOADEDFILES . "docinfo.json", $jsonText);

}
```

You start by loading the data using `json_decode()`, just as you do when displaying the files. In this case, you string together a number of different references to set the `approvedBy` value. The `$workflow` variable contains all the data. Its `fileInfo` property is an array that includes all of your files, so the `$fileNumber` refers to the one that you want. Once you have that, you can set the `approvedBy` property. Similarly, set the `status` to `approved`.

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.

Test this out by approving some of the files and redisplaying the files.

Security checks on download

As the last step, you'll need to 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 (see Listing 32).

Listing 32. Disabling remote downloading

```
<?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>";

      }
   }
}

?>
```

In the `ImproperRequestException`, you create a new method, `logDownloadAttempt()`, that can send an email 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 from your blog, and you click that link, the IBM logs 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

This tutorial wrapped up the three-part series on "Learning PHP," in which you built a simple workflow application. Earlier parts focused on the basics, such as syntax, form handling, database access, file uploading, XML, and JSON. In this part, you took all of that a step further and put it together to create a form through which an administrator can 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 object inheritance
- Using exceptions
- Creating custom exceptions
- Performing additional security checks for downloads

Downloads

Description	Name	Size	Download method
Part 3 source code	Part3CodeFiles.zip	68KB	HTTP

Information about download methods

Resources

Learn

- Read Part 1 and Part 2 of this "Learning PHP" series
- Read the PHP Manual.
- Read about exceptions in PHP Manual: Exceptions.
- Read about security in PHP Manual: Security.
- Read about HTTP authentication in PHP Manual: HTTP authentication with PHP.
- Read about DOM functions in PHP Manual: DOM Functions.
- Learn more about stream functions in PHP Manual: Stream Functions.
- Learn more about string functions in PHP Manual: String Functions.
- Read about Apache HTTP Server Version 1.3 Authentication, Authorization, and Access Control and Apache HTTP Server Version 2.0 Authentication, Authorization and Access Control.
- Read the Apache HTTP Server Version 2.0 tutorial: .htaccess files.
- Visit IBM developerWorks' PHP project resources to learn more about PHP.
- 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.
- Listen to developerWorks podcasts for interesting interviews and discussions for software developers.
- Stay current with developerWorks technical events and webcasts.
- Follow developerWorks on Twitter.

Get products and technologies

- Download XAMPP.
- Download PHP 5.x.
- Download Apache Web server, version 2.x.
- Download MySQL.
- Access IBM trial software (available for download or on DVD) and innovate in your next open source development project using software especially for developers.

Discuss

- Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis. Help build the Real world open source group in the developerWorks community.

About the author

Nicholas Chase

Nicholas Chase is the founder and creator of NoTooMi. In addition to technical writing for large corporations, he has been involved in website 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).