Learning PHP, Part 2

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 teaches you how to use PHP by demonstrating the construction of a Web-based workflow application. *Learning PHP, Part 1* covered the basics, such as syntax, functions, working with HTML forms submissions and databases, and creating a process by which a new user can register for an account.

In this tutorial, you're going to enable users to upload files to the system by using their browsers, and you're going to use XML to store and display information about each file.

Part 3 will look at using HTTP authentication, as well as protecting files by streaming them from a non-Web-accessible directory. You'll also look at creating objects and using exceptions.

In the course of this tutorial, you'll examine:

- Creating and using sessions and session information.
- Uploading files from the browser.
- Creating XML using the Document Object Model (DOM).
- Manipulating XML data using DOM.
- Creating a Simple API for XML (SAX) content handler.
- Reading XML data using SAX.

Who should take this tutorial?

This tutorial is Part 2 in a three-part series designed to teach you the various aspects of working with PHP while building a workflow application. Take this tutorial if you have a basic understanding of PHP and want to learn about uploading files from the browser, sessions, or using PHP to process XML.

This tutorial assumes a basic familiarity with PHP to the level discussed in *Part 1* of this series. That includes basic understanding of control structures, such as loops and if-then statements, as well as functions, and working with HTML forms submissions and databases. Familiarity with XML is helpful, but not required. (You can find more information about these topics in Resources on page38 .)

Prerequisites

To follow along, you need a Web server, PHP, and a database installed and available. Unless you will be using an external hosting account, download and install the following packages:

Web server -- Whether you're on Windows or Linux (or Mac OS X, for that matter), you have the option of using the Apache Web server. Feel free to choose V1.3 or V2.0, but the instructions in this tutorial will concentrate on 2.0. Download *Apache* (http://www.apache.org/) . If you're on Windows, you also can use Internet Information Services, which is part of Windows.

PHP -- Of course, you will also need a distribution of PHP. Both PHP V4 and PHP V5 are in use at the time of this writing, but because of changes in PHP V5, we'll concentrate on that version. (The version isn't terribly important in this tutorial, but it makes a difference for the last part of this series.) Download *PHP* (http://us4.php.net/downloads.php) .

Database -- Part of this project involves saving data to a database, so you'll need one of those, as well. In this tutorial, we'll concentrate on MySQL because it's so commonly used with PHP. Download *MySQL* (http://dev.mysql.com/downloads/) .

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 of science student in computer engineering. In the past, he 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 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. Getting started: creating a session

The login process, Part 1

Part 1 of this series examined working with information submitted from an HTML form and interacting with a database by creating a registration system for new users. You can find the final files in Resources on page38 .

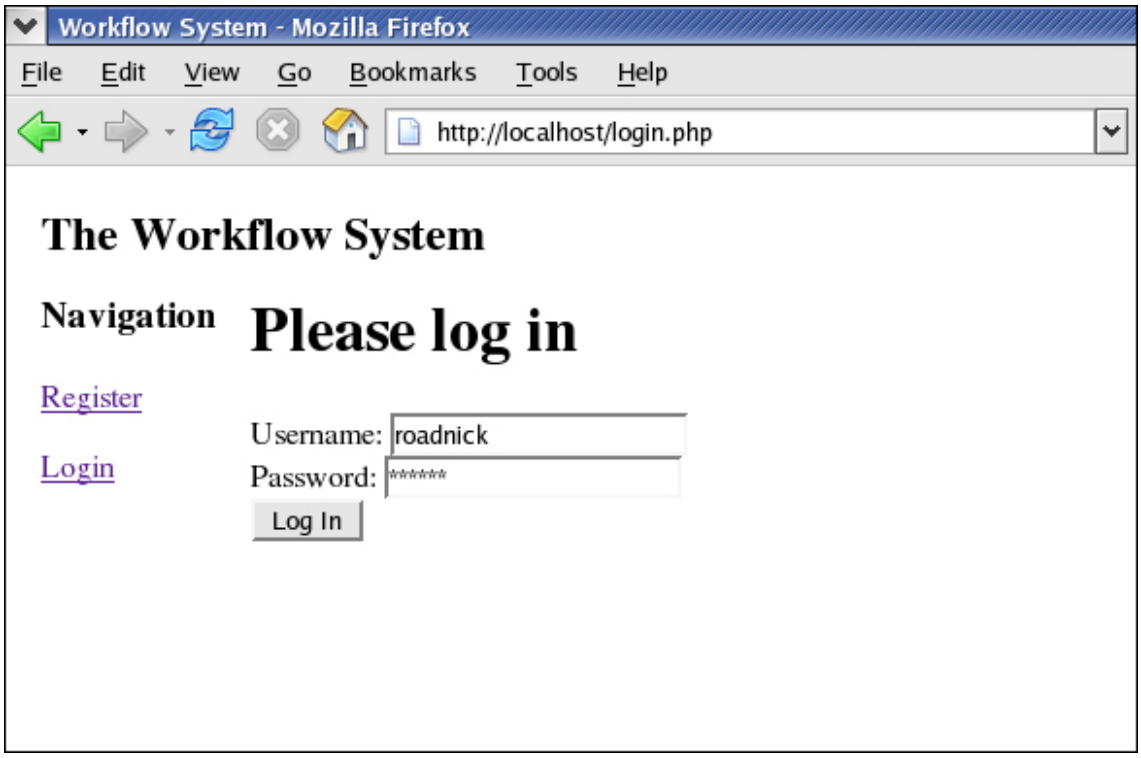
One thing you did not do, however, was create a page by which the user could log into the system. You'll take care of that now, with a brief review of what has already been covered.

First, create a new, blank file, and save it as login.php in the same directory as registration.php. Add the following code:

```
<?php

   include("top.txt");
   require("scripts.txt");
?>

<h1>Please log in</h1>
<form action="login_action.php" method="post">
Username: <input type="text" name="username" /><br />
Password: <input type="password" name="password" /><br />
<input type="submit" value="Log In" />
</form>

<?php
   include("bottom.txt");
?>
```

To review, when the user submits the form, PHP will create an array of information, `$_POST`, with two entries: `username` and `password`. You'll check that information against the database. When you load this page in the browser, the server includes the interface elements.

Figure 1. Login.php page with interface elements



The login process, Part 2

Create another new page and save it as login_action.php. Add the following code:

```
<?php

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

   db_connect();

   $username = $_POST["username"];
   $password = $_POST["password"];
   $sql = "select * from users where username='".$username.
                           "' and password='".$password."'";

   $result = mysql_query($sql);
   $row = mysql_fetch_row($result);
   if ($row) {
      echo "You are logged in.  Thank you!";
   } else {
      echo "There is no user account with that username and password.";
   }

   mysql_close();

   include("bottom.txt");
?>
```

After including the interface elements for the top of the page, open a connection to the database using the `db_connect()` function, which is defined in the scripts.txt file. Once you have a connection, use the submitted information to create a SQL statement that looks for a matching account and execute it against the database.

Starting a session

You ask the user to log in so the site knows which user he is. This way, the site knows which files the user is allowed to see. You have to do more than simply log the user in, however. As things stand right now, once you leave this page, the user's information is gone.

What you want to do is create a *session* so the server knows which requests to group together. You can also associate information -- such as the username -- with the session.

To start with, you'll have to create the session. The `session_start()` function checks to see if a session exists, and if no session exists, the function starts one.

The use of `session_start()` or, in fact, anything done to set session information has a major restriction. It must happen before any content, including interface elements, goes to the browser.

So, first off, let's add the session and restructure login_action.php:

```
<?php
   session_start();

   require("scripts.txt");

   db_connect();

   $username = $_POST["username"];
   $password = $_POST["password"];
   $sql = "select * from users where username='".$username.
                           "' and password='".$password."'";

   $loginOk = false;

   $result = mysql_query($sql);
   $row = mysql_fetch_row($result);
   if ($row) {
     $loginOk = true;
   }

   mysql_close();

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

```
   if ($loginOk) {
      echo "You are logged in.  Thank you!";
   } else {
      echo "There is no user account with that username and password.";
   }

   include("bottom.txt");

?>
```

What you've done here is restructure the page so you can do all your session work before outputting anything to the browser.

Populating the session

Of course, there's no point in creating a session unless you have information to pass around from request to request. In this case, you want to pass the username and the e-mail address, so you add them to the session:

```
<?php
   session_start();

   require("scripts.txt");

   db_connect();

   $username = $_POST["username"];
   $password = $_POST["password"];
   $sql = "select * from users where username='".$username.
                           "' and password='".$password."'";

   $result = mysql_query($sql);
   $row = mysql_fetch_row($result);
   $loginOk = false;
   if ($row) {

      $loginOk = true;
      $_SESSION["username"] = $row["username"];
      $_SESSION["email"] = $row["email"];
   }

   mysql_close();

   include("top.txt");

   if ($loginOk) {
      echo "You are logged in.  Thank you, ".$username."!";
   } else {
      echo "There is no user account with that username and password.";
   }

   include("bottom.txt");

?>
```

Like `$_POST`, `$_SESSION` is an associative array. It's also a *super global* variable, which holds its value through multiple requests. This way, you can reference it from another page.

Now, if you're using a version of PHP later than V4.3.2, you may see a message telling you that your script may be relying on a bug that enables you to set the `$_SESSION` variable in the global scope. If so, you need to find your php.ini file and make sure the following variables are set:

```
session.bug_compat_42 = 1
session.bug_compat_warn = 0
```

After changing these values, you need to stop and restart your Web server for the changes to take effect.

Using an existing session

The `session_start()` function is smart enough to know whether there's an existing session, so you can go ahead and add it to the include for the top of the top.txt file:

```
<?
   session_start();
?>
<html>
<head>
<title>Workflow System</title>
</head>
<body>
<table cellspacing="10">
<tr><td colspan="2"><h2>The Workflow System</h2></td></tr>
...
```

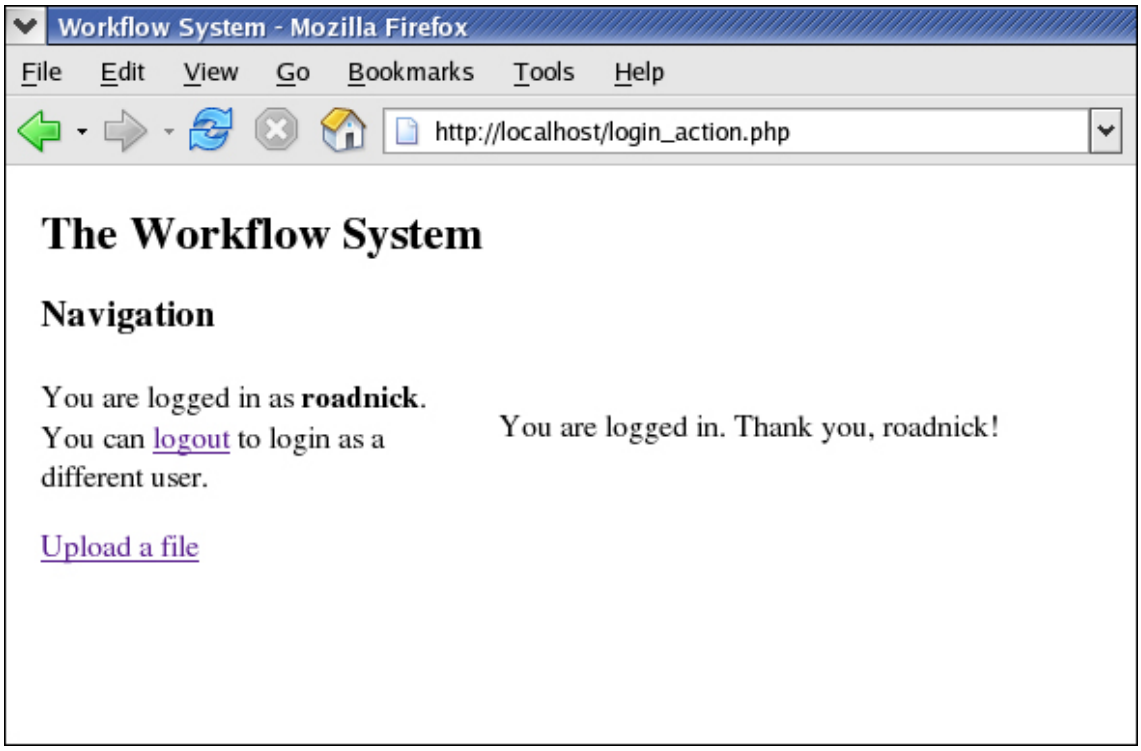
When PHP encounters the `session_start()` function, it joins the existing session in progress, or starts a new one if necessary. This way, other pages have access to the `$_SESSION` array. For example, you can look for the `$_SESSION["username"]` value in the navigation section:

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

```
      <p><a href="login.php">Login</a></p>
<?php
   }
?>
   </td>
   <td>
      <td>
```

Now, you may be wondering why you're also checking for the `$username` variable. The reason is this: When you set the `$_SESSION` variable, the changes don't become available until the next time `session_start()` gets called. So, on the actual login_action.php page, it won't be available yet, but `$username` will, as shown in Figure 2.

Figure 2. Login_action.php page with `$username` variable



Clearing data

When you've finished your session, you can clear out its data or end the session altogether. For example, you may want to log a user out without ending the session. Create a new file called logout.php and add the following code:

```
<?
   session_start();
   unset($_SESSION["username"]);
   unset($_SESSION["email"]);
```

```
   include("top.txt");
   echo "Thank you for using the Workflow System.  You may <a href=\"login.php\
">log in again</a>.";
   include("bottom.txt");
?>
```

Executing this page by pointing the browser to http://localhost/logout.php shows that the values have been cleared, as shown in Figure 3.

Figure 3. Page with values cleared

A simpler way to clear out data is to end the session:

```
<?
   session_start();
   session_destroy();

   include("top.txt");
   echo "Thank you for using the Workflow System.  You may <a
   href=\"login.php\">log in again</a>.";
   include("bottom.txt");
?>
```

Notice that you still have to use `session_start()` to join the current session before you can destroy it. This step clears all values associated with the session.

Section 3. Uploading files

How uploading works

In addition to text information, you can use HTML forms to send documents, and that's how you enable users to add files to the system. Here's how the process works:

1. Users load a form that enables them to choose a file to upload.
2. Users submit the form.
3. The browser sends the file and information about it to the server as part of the request.
4. The server saves the file in a temporary storage location.
5. The PHP page processing the form submission moves the file from temporary to permanent storage.

Let's start the process by creating the actual form.

The upload form

The form being used for uploading files is similar to those used for the registration and login pages, with two important exceptions:

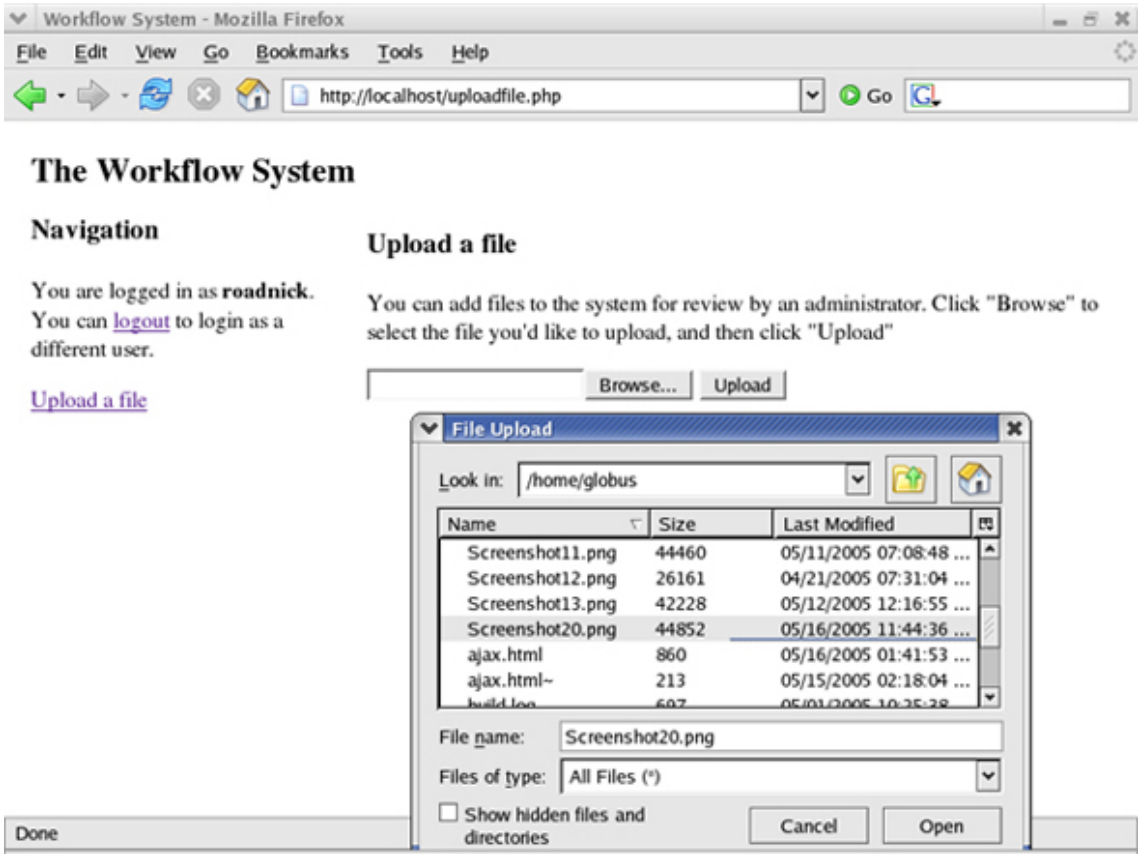
```
<?php
   include("top.txt");
?>

<h3>Upload a file</h3>

<p>You can add files to the system for review by an administrator.
Click <b>Browse</b> to select the file you'd like to upload,
and then click <b>Upload</b>.</p>

<form action="uploadfile_action.php" method="POST"
enctype="multipart/form-data">
<input type="file" name="ufile" \>
<input type="submit" value="Upload" \>
</form>

<?php
   include("bottom.txt");
?>
```

The `enctype` attribute tells the browser that the information it sends must be in a particular format that allows for multiple sections of information rather than just a list of name-value pairs.

The file input provides a box that enables the user to click **Browse...** and

choose the file, as shown in Figure 4.

Figure 4. Choose a file to upload



Notice that a link to this file has been added, `uploadfile.php` to the `top.txt` page, so it appears in the browser.

The uploaded information

When you upload a file through the browser, PHP receives an array of information about it. You can find this information in the `$_FILE` array, based on the name of the input field. For example, your form has a file input with the name *ufile*, so all the information about that file is contained in the array `$_FILE['ufile']`.

This array allows the user to upload multiple files. As long as each of the files has its own name, it will have its own array.

Now, notice "`$_FILE`" is being called an array. In *Part 1* of this series, you had a situation in which an array value was itself an array when you passed multiple form values with the same name for the password. In this case, each value of the `$_FILE` array is itself an associative array. For example, your ufile file has the following information:

- `$_FILE['ufile']['name']` -- The name of the file (for example, uploadme.txt)
- `$_FILE['ufile']['type']`: The type of the file (for example, `image/jpg`)
- `$_FILE['ufile']['size']` -- The size, in bytes, of the file that was uploaded
- `$_FILE['ufile']['tmp_name']` -- The temporary name and location of the file uploaded on the server
- `$_FILE['ufile']['error']` -- The error code, if any, that resulted from this upload

Knowing what information should be present, you should check to see whether a file was actually uploaded before you perform any processing.

Check for a file

Before you take any action regarding the file, you need to know whether a file was uploaded in the first place. Create the action page for this form, uploadfile_action.php, and add the following code:

```
<?php
   include("top.txt");

   if(isset($_FILES['ufile']['name'])){
       echo "Uploading: ".$_FILES['ufile']['name']."<br />";
   } else {
       echo "You need to select a file.  Please try again.";
   }

   include("bottom.txt");
?>
```

Next you'll save the file.

Save the file

Before you even start the process of saving the uploaded file, you need to decide where to put it. Until the file's been approved, you don't want it accessible from the Web site, so create a directory that's not in the main document root.

In this case, you will use /var/www/hidden/. That's where all your files will go, so it's probably a good idea to define a constant. Open scripts.txt and add the following definition:

```
...
  mysql_select_db($db);

}
   define(UPLOADEDFILES, "/var/www/hidden/");
?>
```

Now you can use this definition in the upload page, as long as you include scripts.txt in that page, as well:

```
<?php
   include("top.txt");
   include("scripts.txt");

   if(isset($_FILES['ufile']['name'])){
       echo "Uploading: ".$_FILES['ufile']['name']."<br>";

       $tmpName = $_FILES['ufile']['tmp_name'];
       $newName = UPLOADEDFILES . $_FILES['ufile']['name'];

       if(!is_uploaded_file($tmpName) ||
                           !move_uploaded_file($tmpName, $newName)){
            echo "FAILED TO UPLOAD " . $_FILES['ufile']['name'] .
                 "<br>Temporary Name: $tmpName <br>";
       } else {
            echo "File uploaded.  Thank you!";
       }

   } else {
     echo "You need to select a file.  Please try again.";
  }
   include("bottom.txt");
?>
```

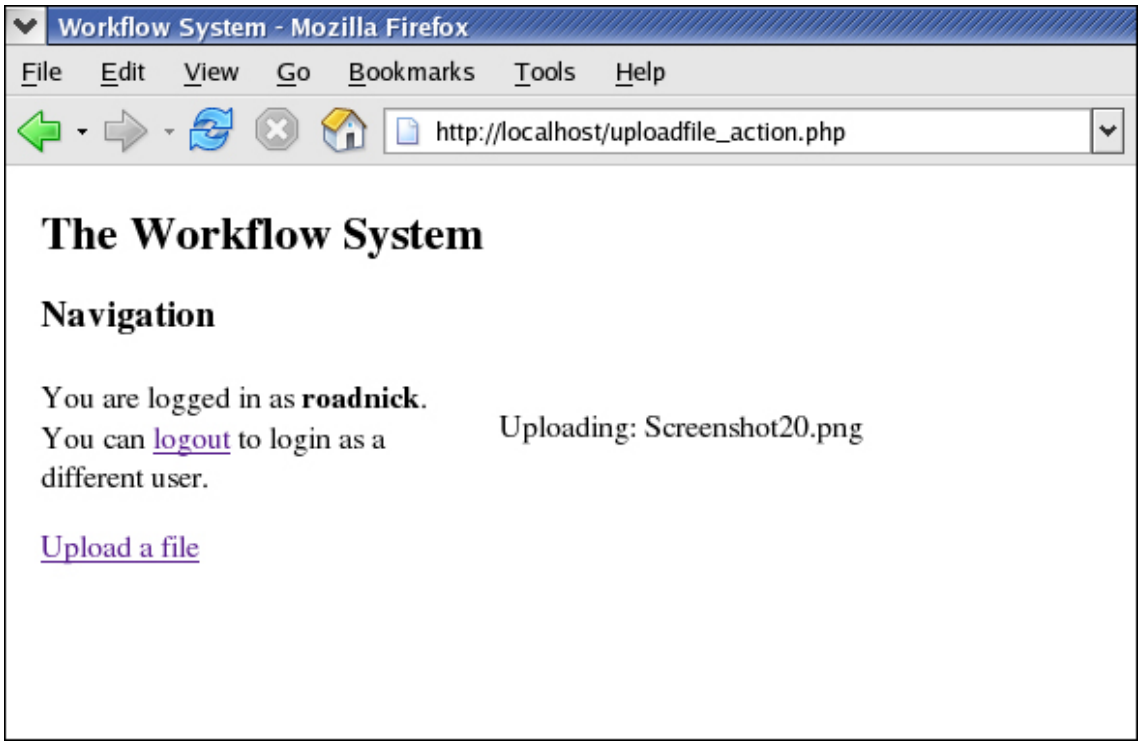
First you're getting the current location of the file and its temporary name (`tmp_name`), and determining where you want it to go using the constant you defined.

Next, right there in the if-then statement, you're doing two things. You're checking to make sure that the file you're trying to move is actually a file that's been uploaded to the server, rather than a file such as /etc/passwd that the user's tricked us into acting upon. If the `is_uploaded_file()` function comes back false, then its opposite, `!is_uploaded_file()`, is true, and PHP moves on to display the error message.

If, on the other hand, that test comes back true, which means it is an uploaded file, you can then attempt to move it from its current location to a new location. If that move doesn't work, it returns false, and again its opposite is true, so you display the error message.

In other words, if it's not an uploaded file or you can't move it, you'll display an error message. Otherwise, you will display the success message (see Figure 5).

Figure 5. The success message



Now that you have the file, you need to record its information for later retrieval.

Section 4. Using XML: DOM

What is XML?

These days, it's hard to do very much programming without running into XML in some form. Fortunately, it's not a difficult subject to understand. In fact, chances are, you've already been dealing with a relative of XML: HTML. Consider this HTML V4.01 page:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
            "http://www.w3.org/TR/html4/loose.dtd">
<HTML>
<HEAD>
   <TITLE>Workflow System</TITLE>
</HEAD>
<BODY>
   <H1 align="center">Welcome to the Workflow System!</H1>
   We're glad you could make it.
   <P>
   Don't forget to log in!
</BODY>
</HTML>
```

This page has a lot in common with XML. First off, it's made up of *elements*, such as `HEAD` or `BODY`, marked off by tags, such as `<BODY>` to start, and `</BODY>` to end. These elements can contain other elements (such as `TITLE`) or text (such as `Workflow System`). They can also have attributes, such as `align="center"`.

But XML has some restrictions that don't apply to HTML. For example, XML must be *well formed*, which means that for every start tag (such as `<H1>`), you must have an end tag (such as `</H1>`. That means your `<P>` tag must have an end tag: `</P>`. (Alternatively, you can write it as the shortcut *empty* element: `<P />`.) Also, XML is case-sensitive, so you'd have to close `<P>` with `</P>`, and not `</p>`.

In this section, you'll create an XML file that records information about documents uploaded by users.

How you're going to store information

Over the course of the next two sections, you'll deal with an XML file that lists information about documents users have uploaded. The file will list each document, its status, who uploaded it, and information about the document itself. The file will also list statistical information about the overall system.

The file will look something like this:

```
<?xml version="1.0"?>
<workflow>
   <statistics total="2" approved="1"/>
   <fileInfo status="approved" submittedBy="roadnick2">
      <approvedBy>tater</approvedBy>
      <fileName>timeone.jpg</fileName>
      <location>/var/www/hidden/</location>
      <fileType>image/jpeg</fileType>
      <size>2020</size>
   </fileInfo>
   <fileInfo status="pending" submittedBy="roadnick">
      <approvedBy/>
      <fileName>timeone.jpg</fileName>
      <location>/var/www/hidden/</location>
      <fileType>image/jpeg</fileType>
      <size>2020</size>
   </fileInfo>
</workflow>
```

We added spacing here to make things more clear, but this is the overall structure of the file. Each document will have its own `fileInfo` element, which includes attributes on its status and owner, and contains information about the file itself. The statistics document also lists information about the file itself.

To manipulate this information, you'll use the Document Object Model (DOM).

What is DOM?

The Document Object Model (DOM) is a way of representing XML data as a hierarchical tree of information. Take, for example, this `fileInfo` element:

```
<fileInfo status="approved" submittedBy="roadnick2">
   <approvedBy>tater</approvedBy>
   <fileName>timeone.jpg</fileName>
   <location>/var/www/hidden/</location>
   <fileType>image/jpeg</fileType>
   <size>2020</size>
</fileInfo>
```

Each piece of information in an XML document is represented as a type of node. For example, here you see a `fileInfo` element node; a `status` attribute node; and multiple text nodes, including `tater`, `timeone.jpg`, and `2020`. Even each piece of whitespace between elements is considered a text node.

DOM arranges nodes in a parent-child relationship. For example, the `approvedBy` element is a child of the `fileInfo` element, and the `tater` text is a child of the `approvedBy` element. This relationship means that the `fileInfo` element has *11* child nodes; five elements, and six whitespace text nodes.

Various APIs exist to enable you to work with DOM objects. Earlier versions of PHP implemented a DOM-like structure, but the structure didn't really mesh well with the methods and definitions in the actual DOM recommendations (maintained by the W3C). So, PHP V5 has a new set of DOM operations that are much closer to the standard.

You'll use this new API to create and modify the document information file.

A quick word about objects

Part 3 of this series will talk about objects and object-oriented programming in PHP in more detail, but both the DOM API and the SAX API discussed in the next sections use them, so before going any further, you'll need to have at least a basic understanding of what objects are and how they work.

Think of an object as a collection of functions; for example, you might create a class, or object template, that looks something like this:

```
class Greeting{
    function getGreeting(){
       return "Hello!";
   };
   function getPersonalGreeting($name){
       return "Hello, ".$name."!";
   };
}
```

When you create an object, you can then reference its functions. For example:

```
$myObject = new Greeting();
$start = $myObject->getPersonalGreeting("Nick");
```

In this case, you create the `myObject` object, which you can reference with a variable, just like a string or a number. You can then reference its functions using the `->` notation. Other than that, it's just like calling a regular function.

That's all you need to know about objects for now.

Preparing to save the information

Now you're ready to start creating the document information file. Ultimately, you'll create a function called `save_document_info()`, so start by adding a call to this function in uploadfile_action.php:

```
    if(!is_uploaded_file($tmpName) ||
```

```
        !move_uploaded_file($tmpName, $newName)){
       echo "FAILED TO UPLOAD " . $_FILES['ufile']['name'] .
            "<br>Temporary Name: $tmpName <br>";
    } else {

       save_document_info($_FILES['ufile']);

    }
  ...
```

You want to pass information only on the file that's been uploaded, so you'll make things easier for `save_document_info()` by passing only the information on the ufile file, rather than the entire `$_FILES` array.

Now it's time to create the function.

Section 5. Functions

Create the DOM document

First, you create a DOM `Document` object you can use to manipulate the data. Open the scripts.txt file and add the following code:

```
...
   define(UPLOADEDFILES, "/var/www/hidden/");

   function save_document_info($fileInfo){

       $doc = new DOMDocument('1.0');

   }
?>
```

The actual process of creating a `Document` object, in this case called `$doc`, is straightforward. The `DOMDocument` class is part of the PHP V5 core. You use this object to perform most of your actions on the data.

Create an element

Now that you've got a working `Document`, you can use it to create the main, or *root*, element of the document:

```
...
function save_document_info($fileInfo){

   $doc = new DOMDocument('1.0');
   $root = $doc->createElement('workflow');
   $doc->appendChild($root);

}
...
```

Here you tell the `$doc` object to create a new element node and return it to the `$root` variable. You then tell the `Document` to add that element as a child of itself.

But what happens if you save this document as a file?

Save the document to a file

One advantage of using PHP to process your XML is that PHP provides an easy way to save the contents of a `Document` to a file. (Believe it or not, it's not always this easy.) To see what is actually being generated, add the following code to `save_document_info()`:

```
...
function save_document_info($fileInfo){

   $doc = new DOMDocument('1.0');
   $root = $doc->createElement('workflow');
   $doc->appendChild($root);

   $doc->save(UPLOADEDFILES."docinfo.xml");

}
...
```

You defined the `UPLOADEDFILES` constant earlier, so you can just go ahead and reference it now, placing the new file in the same directory. If you now upload a file through the browser, the docinfo.xml file should look like this:

```
<?xml version="1.0"?>
<workflow/>
```

Don't worry about the first line; it's the XML declaration, and it's standard, but optional (in most cases).

Notice that your element's been saved, but because you haven't actually added any children, or content, to it yet, it's written as an empty element.

Now let's add a more complicated element.

Create attributes

Start adding actual information to the file. You've already created a file by virtue of testing the previous step, but until you've got the first file's information completely saved, you can assume you're still creating the docinfo.xml file.

Start by creating the `statistics` element:

```
...
function save_document_info($fileInfo){

   $doc = new DOMDocument('1.0');
   $root = $doc->createElement('workflow');
   $doc->appendChild($root);

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

```
    $root->appendChild($statistics);

    $doc->save(UPLOADEDFILES."docinfo.xml");

}
...
```

Notice that you still use the `Document` to create the new element, this time called `statistics`. The `Document` acts as a "factory" for most of your objects.

Once you've got the `Element` object, `$statistics`, you can use its built-in functions to set two attributes: `total` and `approved`. Once you've done that, you want to add this element as a child of `$root`, rather than as a child of `$doc`. If you save the file, you can see the difference:

```
<?xml version="1.0">
<workflow><statistics total="1" approved="0"/></workflow>
```

You'll notice two things here. First, notice that the `statistics` element is a child of the `workflow` element. Notice also that there is no extraneous whitespace here. The `statistics` element is the first child of `workflow`.

Now let's look at adding real information.

Create the file information element

Now you can go ahead and create the actual document information. The process uses the techniques you just learned:

```
...
function save_document_info($fileInfo){

    $doc = new DOMDocument('1.0');
    $root = $doc->createElement('workflow');
    $doc->appendChild($root);

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

    $filename = $fileInfo['name'];
    $filetype = $fileInfo['type'];
    $filesize = $fileInfo['size'];

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

    $fileInfo->setAttribute("status", "pending");
    $fileInfo->setAttribute("submittedBy", $_SESSION["username"]);

    $approvedBy = $doc->createElement("approvedBy");
```

```
    $fileName = $doc->createElement("fileName");
    $fileNameText = $doc->createTextNode($filename);
    $fileName->appendChild($fileNameText);

    $location = $doc->createElement("location");
    $locationText = $doc->createTextNode(UPLOADEDFILES);
    $location->appendChild($locationText);

    $type = $doc->createElement("fileType");
    $typeText = $doc->createTextNode($filetype);
    $type->appendChild($typeText);

    $size = $doc->createElement("size");
    $sizeText = $doc->createTextNode($filesize);
    $size->appendChild($sizeText);

    $fileInfo->appendChild($approvedBy);
    $fileInfo->appendChild($fileName);
    $fileInfo->appendChild($location);
    $fileInfo->appendChild($type);
    $fileInfo->appendChild($size);

    $root->appendChild($fileInfo);

    $doc->save(UPLOADEDFILES."docinfo.xml");

}
...
```

Even though there's a lot of code here, very little is new. First you extract the actual information about the file from the information passed to the function. Then you create the `fileInfo` element that will contain all the information you're adding. You set the `status` and `submittedBy` attributes on this element, and then look at creating its children.

The `approvedBy` element is easy. It's not approved yet, so that will stay empty. The `fileName` element, on the other hand, is a bit more difficult because you need to add a text child to it. Fortunately, that's also pretty straightforward. You create the element, then use the `Document` to create a new text node that has as its content the name of the file. You can then add that text node as a child of the `fileName` element.

You progress in this way, creating all the elements that will eventually be children of `fileInfo`. After you're finished, you append all of them as children of the `fileInfo` element. Finally, you add the `fileInfo` element itself to the root element, `workflow`.

The results, with spacing added for clarity, look something like this:

```
<?xml version="1.0"?>
<workflow>
   <statistics total="1" approved="0"/>
   <fileInfo status="pending" submittedBy="roadnick">
      <approvedBy/>
      <fileName>signed.pem</fileName>
      <location>/var/www/hidden/</location>
```

```
      <fileType>application/octet-stream</fileType>
      <size>2754</size>
   </fileInfo>
</workflow>
```

Of course, you can't keep overwriting the information file every time someone uploads a document, so next, you'll look at working with an existing structure.

Loading an existing document

Now that you know how to add information to the file, you need to look at working with the file on subsequent uploads. First you need to see whether the file already exists, then act accordingly:

```
...
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);
   } else{

      $doc = new DOMDocument('1.0');
      $root = $doc->createElement('workflow');
      $doc->appendChild($root);

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

   $filename = $fileInfo['name'];
   $filetype = $fileInfo['type'];
   $filesize = $fileInfo['size'];

   $fileInfo = $doc->createElement("fileInfo");
...
   $fileInfo->appendChild($size);

   $root->appendChild($fileInfo);

   $doc->save($xmlfile);

}
```

Starting at the top, you create a variable to represent the location of the file, since you'll now reference it in more than one place. Next, you check to see if that file already exists. If it does, you call the `load()` function, rather than creating a new object.

This static function -- which we will talk about more in Part 3 of this series; for now, understand that they're functions you can call from the class, rather than from an object -- returns a `Document` object that's already populated with all the elements, text, and so on that are represented in the file.

Once you have the `Document` object, you need the `workflow` element because you'll ultimately need to add the new `fileInfo` element to it. You get the `workflow` element by first retrieving a list of all the elements in the document called `workflow`, and then selecting the first one in the list.

From there, you can simply add the new `fileInfo` element, and it will appear after the original. Again, with space added for clarity:

```
<?xml version="1.0"?>
<workflow>
   <statistics total="1" approved="0"/>
   <fileInfo status="pending" submittedBy="roadnick">
      ...
   </fileInfo>
   <fileInfo status="pending" submittedBy="roadnick">
      <approvedBy/>
      <fileName>timeone.jpg</fileName>
      <location>/var/www/hidden/</location>
      <fileType>image/jpeg</fileType>
      <size>2020</size>
   </fileInfo>
</workflow>
```

But what about the `statistics`? Obviously, they're no longer correct. That will have to be fixed.

Manipulating existing data

In addition to adding information to the document, you can alter information that's already there. For example, you can update the `total` attribute on the `statistics` element:

```
...
   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{
...
```

First off, you get a reference to the existing `statistics` element the same

way you got one to the existing `workflow` element, only you combine both steps into one. Once you have a reference to the element, you can get the current value of the `total` attribute using the `getAttribute()` function. Then you can use that value to provide an updated value for the `total` attribute using `setAttribute()`.

The results are as you might expect, this time with no space added:

```
<?xml version="1.0"?>
<workflow><statistics total="2" approved="0"/><fileInfo status="pending"
submittedBy="roadnick">...
```

Now that you know how to use DOM to create the file, let's look at reading it back using another XML API, SAX.

Section 6. Using XML: SAX

What is SAX?

In the previous section, you learned about DOM, which treats XML data like a series of objects arranged in a hierarchy. As such, it's known as an *object-based API*. In this section, you're going to learn about the Simple API for XML, or SAX, which is an *event-based API*. You'll use this API to display information about the files available for a particular user to see.

SAX works by using a *content handler* to monitor a stream of events. Consider, for example, this XML document:

```
<workflow>
   <statistics total="3" approved="2" />
   <fileInfo submittedBy="roadnick" status="approved">
     <fileName>timeone.jpg</fileName>
   </fileInfo>
</workflow>
```

In parsing this document, a SAX content handler would see the following events:

```
Start document
Start element (workflow)
Characters (whitespace)
Start element (statistics)
End element (statistics)
Characters (whitespace)
Start element (fileInfo)
Characters (whitespace)
Start element (fileName)
Characters (timeone.jpg)
End element (fileName)
Characters (whitespace)
End element (fileInfo)
Characters (whitespace)
End element (workflow)
End document
```

Because the content handler sees these events, you can assign it a function to perform for each of them.

Create the content handler

The first step is to create the content handler itself. In this case, you're dealing with a class called `Content_Handler`, but in PHP, the name is entirely arbitrary. Open scripts.txt and add the following:

```
class Content_Handler{
  function start_element($parser, $name, $attrs){

     echo "Start element: ".$name."<br />";

  }
  function end_element($parser, $name){

     echo "End element: ".$name."<br />";

  }
  function chars($parser, $chars){

     echo "Text: ".$chars."<br />";

  }
}
```

For the moment, you want the functions to output just enough so you know they've been called. Now you need to tell the parser where to find the content handler.

Assign handlers

In order for PHP to parse the file, it needs to know which functions to execute for which events. Add the `display_files()` function, seen below, to scripts.txt:

```
...
function display_files(){

   $handler = new Content_Handler();

   $doc_parser = xml_parser_create();

   xml_set_object($doc_parser, $handler);

   xml_set_element_handler($doc_parser,
                           "start_element",
                           "end_element");
   xml_set_character_data_handler($doc_parser, "chars");

   xml_parser_set_option($doc_parser,
                         XML_OPTION_CASE_FOLDING, 0);

}
...
```

First, you create a new `Content_Handler` object, `$handler`. Next, you get a reference to the XML parser that's built into PHP, and use the `xml_set_object()` function to tell it where to find the functions you're going to reference in a moment.

Next, tell the parser which functions you want to use to handle various events. In some APIs, these function names are set in stone, but in PHP, they're arbitrary. So, you have to specifically tell PHP, using the `xml_set_element_handler()` function, that it should handle the "start element" event with the `start_element()` function.

Note that you could have skipped `xml_set_object()` and simply used:

```
    xml_set_element_handler($doc_parser,
                            "$handler->start_element",
                            "$handler->end_element");
```

Finally, you turn off the case folding option so the parser doesn't convert all element names to all capital letters. Now you're ready to parse the actual file.

Open and parse the file

At this point, you're ready to parse the file, but nothing's going to happen until you do two things. The first is to actually add the parsing of the file to `display_files()`:

```
...
    xml_parser_set_option($doc_parser,
                          XML_OPTION_CASE_FOLDING, 0);

    $xmlfile = UPLOADEDFILES."docinfo.xml";

    $file_to_parse = fopen($xmlfile, "r");
    if (!$file_to_parse) die("Can't open XML file.");

    while($data = fread($file_to_parse, 4096)){
            xml_parse($doc_parser, $data, feof($file_to_parse));
    }
}
...
```

For the process to be efficient (and more maintainable), you also use the `UPLOADEDFILES` constant here to specify the location of the document information file. From there, you use PHP's built-in file management capabilities to create a file handle for that file, `$file_to_parse`, and if that doesn't work, you display a message to that effect.

Once you have the reference to the file, you run through it in chunks of up to 4,096 bytes. For each chunk, you send the data to the parser. The loop continues until it gets to the end of the file and stops.

The second thing you need to do for anything to happen is to call `display_files()` from uploadfile_action.php:

```
...
    echo "FAILED TO UPLOAD " . $_FILES['ufile']['name'] .
         "<br>Temporary Name: $tmpName <br>";
  } else {

    save_document_info($_FILES['ufile']);
    display_files();

  }
...
```

Now it's time to see what happens.

The most basic results

At this point, if you upload a file, PHP will save the file, add the information about it to the docinfo.xml file, and call the parser (and with it, the `Content_Handler` object) to parse the data. When that process happens, each function gets called for its own specific event, and you get results like these:

```
Start element: workflow
Start element: statistics
End element: statistics
Start element: fileInfo
Start element: approvedBy
End element: approvedBy
Start element: fileName
Text: timeone.jpg
End element: fileName
Start element: location
Text: /var/www/hidden/
End element: location
Start element: fileType
Text: image/jpeg
End element: fileType
Start element: size
Text: 2020
End element: size
End element: fileInfo
Start element: fileInfo
Start element: approvedBy
End element: approvedBy
Start element: fileName
Text: timeone.jpg
End element: fileName
Start element: location
Text: /var/www/hidden/
End element: location
Start element: fileType
Text: image/jpeg
End element: fileType
Start element: size
Text: 2020
End element: size
```

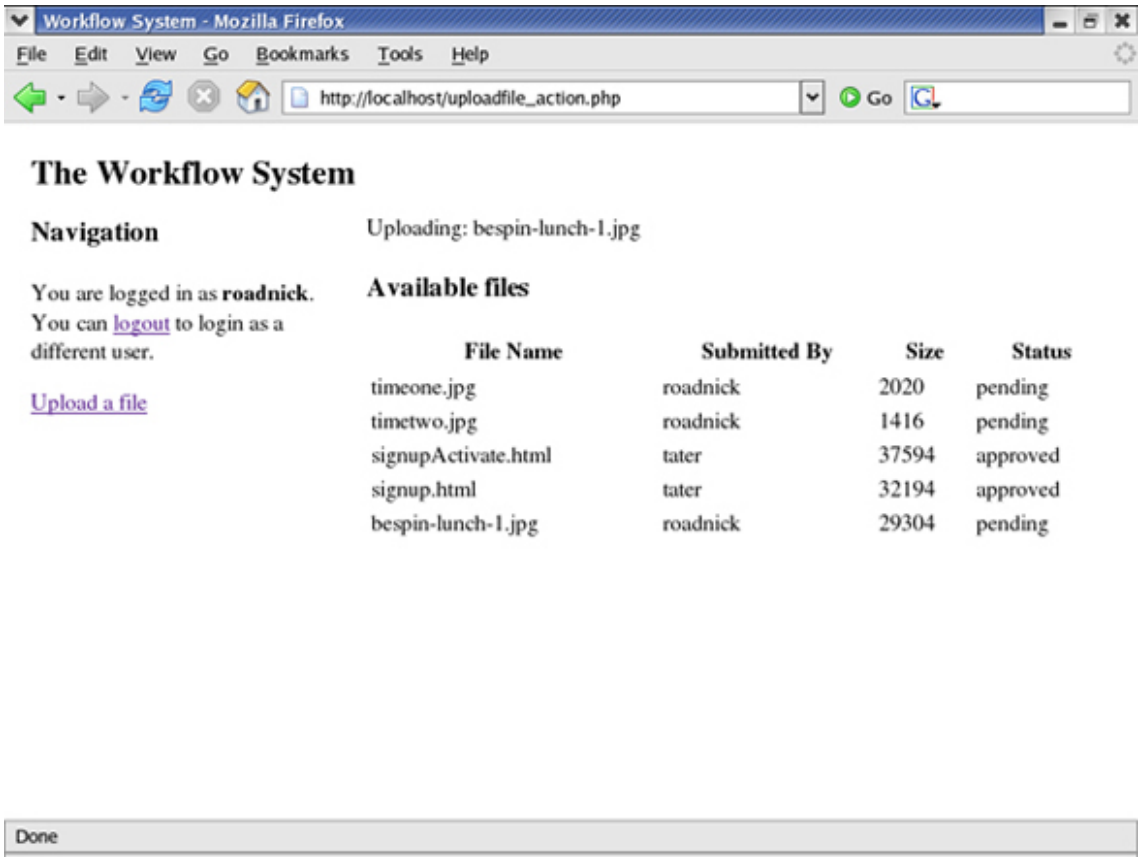
```
End element: fileInfo
End element: workflow
```

These results are informational, of course, but not exactly pretty. Now let's look at creating a page that actually displays information in a useful way.

The overall display

At this point, you have output and you can see how the handler is getting called, but you need to massage this information into some kind of reasonable output. Ultimately, you want a table of information that looks something like Figure 6.

Figure 6. Readable table of information



This table translates into an HTML page of:

```
...
Uploading: bespin-lunch-1.jpg<br />
<!-- At the start of the document -->
<h3>Available files</h3>
<table width='100%'>
<!-- At the start of the document -->
<!-- For each fileInfo element -->
   <tr><th>File Name</th>
```

```
    <th>Submitted By</th><th>Size</th>
        <th>Status</th></tr>
<!-- For each fileInfo element -->
    <tr><td>timeone.jpg</td>
    <td>roadnick</td><td>2020</td>
        <td>pending</td></tr>
    <tr><td>timetwo.jpg</td>
<td>roadnick</td><td>1416</td>
        <td>pending</td></tr>
    <tr><td>signupActivate.html</td><td>tater</td>
        <td>37594</td><td>approved</td></tr>
    <tr><td>signup.html</td><td>tater</td>
<td>32194</td>
        <td>approved</td></tr>
    <tr><td>bespin-lunch-1.jpg</td><td>roadnick</td>
        <td>29304</td><td>pending</td></tr>
<!-- At the end of the document -->
</table>
<!-- At the end of the document -->
</td>
```

Looking at the comments, you can see how you need to break this up for the various events. Let's go ahead and do that.

The start and end of the document

In some languages, you can set a specific function to handle the start and end of a document, but PHP doesn't make it quite that simple. You could handle these events separately, but in this case, you also have the option to simply treat the start and end of the `workflow` element as the start and end of the document. For example, add the following to the `Content_Handler` class definition:

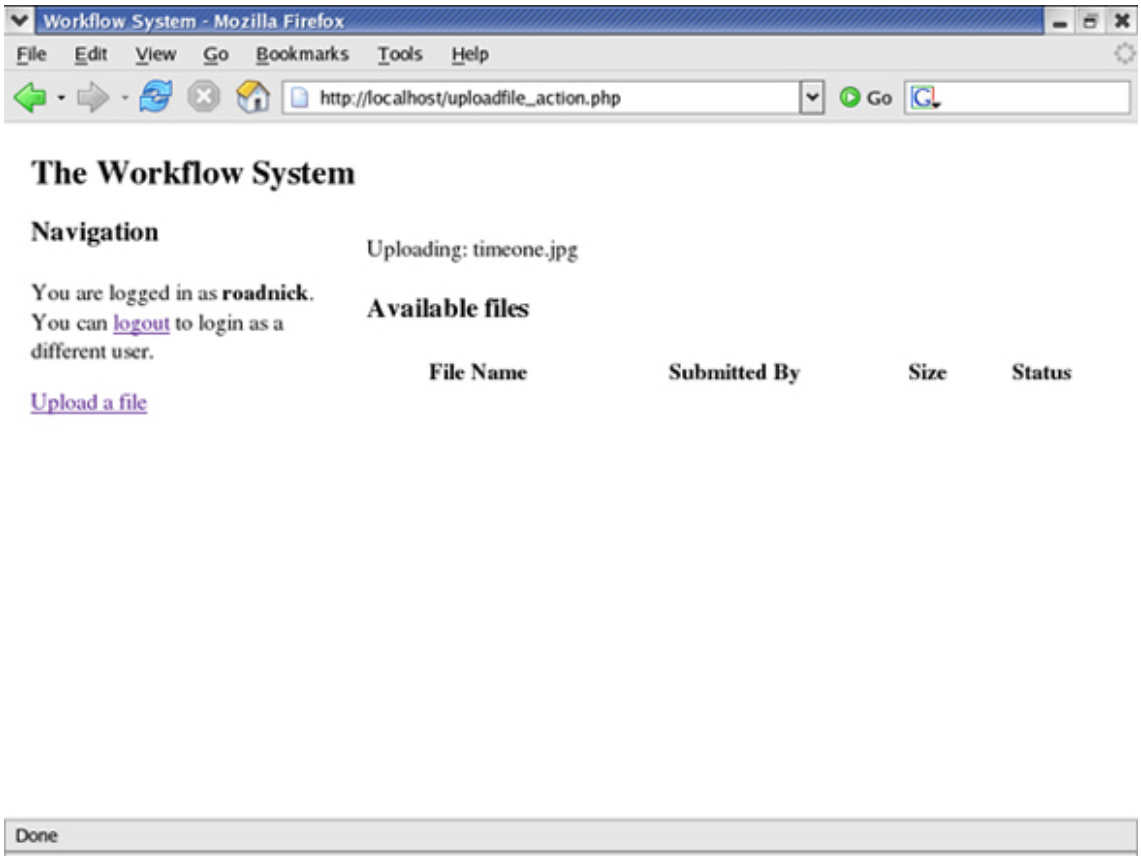
```
...
class Content_Handler{

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

     if ($name == "workflow"){
        echo "<h3>Available files</h3>";
        echo "<table width='100%'><tr>".
            "<th>File Name</th><th>Submitted By</th>".
            "<th>Size</th><th>Status</th></tr>";
     }

  }

  function end_element($parser, $name){

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

  }
```

```
    function chars($parser, $chars){
       //echo "Text: ".$chars."<br />";
    }
}
...
```

When the parser sends the `start element` event, it calls the `start_element()` function and passes the parser, the name of the element, and any attributes to that function. You can use that output to decide whether to display the information.

In this case, you get just a single row for the top of the table and the `</table>` tag for the bottom of the table, as you can see in Figure 7.

Figure 7. Display information from the start element



Displaying the individual rows, however, is not nearly that simple.

The `start_element()` method

One important aspect of an event-based API is the fact that at any given time, the parser has information only about that specific event. That means that while you can move up and down the tree when you're dealing with DOM, when you use SAX, you have no way of knowing what's come before unless you've

explicitly saved it. In other words, when you get to the `fileName` element, you don't know whether that `fileInfo` element is available for display or not, unless you've explicitly noted it:

```
...
class Content_Handler{

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

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

  if ($name == "workflow"){
     echo "<h3>Available files</h3>";
     echo "<table width='100%'><tr>".
          "<th>File Name</th><th>Submitted By</th>".
          "<th>Size</th><th>Status</th></tr>";
     }

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

  }

  function end_element($parser, $name){
...
```

When you get to the beginning of a `fileInfo` element, you want to check the `status` attribute and the `submittedBy` element to see if either of them is a reason to show the document. You can do that by using the `$attrs` array.

If the `status` is approved or the file was submitted by the current user, you want to note that any data that's coming within this element is available. To do that, you've created a variable called `available` within the class itself. (Don't worry about that `private` stuff; we will discuss it in Part 3.) Because the variable is part of this object, you can reference it using `$this->available`, and this value will remain set even when you move on to the next event.

If the information *is* available, you want to save it so it can be displayed when the time comes. To that end, you save it in the `submittedBy` and `status` variables.

You need to do the same thing in dealing with certain text.

The **`characters()`** method

Some of the information you want to display is stored as attributes of the `fileInfo` element, but some is stored as text nodes of other elements. To save the information, however, you need to know which element you're processing:

```
class Content_Handler{

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

  private $currentElement = "";

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

  function start_element($parser, $name, $attrs){
...
     $this->currentElement = $name;

  }

  function end_element($parser, $name){

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

     $this->currentElement = "";

  }

  function chars($parser, $chars){

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

  }
}
```

Let's start at the top. First of all, you create a variable to store the name of the current element and store it each time you execute the `start_element()` function. You clear it out every time you hit the `end_element()` function, but it's important to note that just clearing it doesn't revert it to its previous value. It's not like walking the XML tree.

The important part is that you know which element you're dealing with when you get to the `chars()` function, which is executed one or more times for each text

node. Each time it's executed, if the file is available for display, you're checking the current element and adding the current characters to the appropriate variables if necessary.

The `end_element()` method

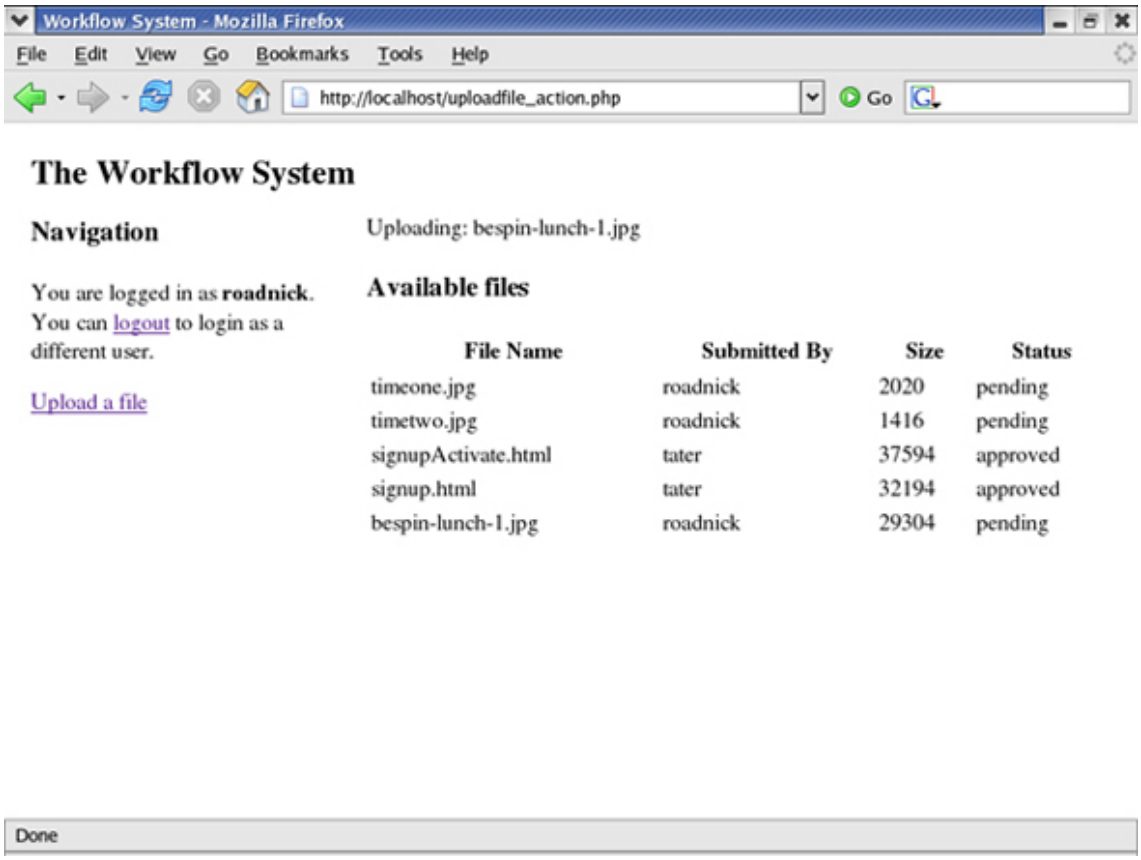
Now that you've stored all this information, you need to display it for each `fileInfo` element. The easiest way to display it is to output when you get to the end of each `fileInfo` element:

```
...
  function end_element($parser, $name){

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

     if ($name == "fileInfo"){
        echo "<tr><td>".$this->fileName."</td>".
                  "<td>".$this->submittedBy."</td>".
                  "<td>".$this->fileSize."</td>".
                  "<td>".$this->status."</td></tr>";

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

        $this->available = false;
     }

     $this->currentElement = "";

  }
...
```

When the `end_element()` function executes, you first display a table row with the appropriate information in it. Once that's done, you need to clear out all the variables, including the `$available` variable, so that you're ready to process the next `fileInfo` element.

The result is a table of information as shown in Figure 8.

Figure 8. Resulting table of information



In Part 3, you'll look at enhancing this table with links and other information and functionality.

Section 7. Summary and resources

Summary

In this tutorial, you started to create the heart of the workflow application: the user's ability to add files. You enabled users to log into the system and create a session that recognizes them and upload a file. You saved the file on the server, and you used XML to save information about it. Along the way, the following topics were covered:

- Creating a session
- Using an existing session
- Uploading a file
- Creating an XML file using DOM
- Loading XML data using DOM
- Manipulating XML data using DOM
- Creating a SAX content handler
- Parsing an XML file using SAX
- Tracking XML data using SAX

In Part 3, you'll complete the application.

Resources

All we can do in a tutorial with a scope like this one is introduce topics with enough information to make it easier for you to delve into them on your own. Here are some good resources for further study:

- For an excellent primer on objects in PHP, read *Using PHP Objects*.
- Access Part 1 of this series *Learning PHP, Part 1* (developerWorks, June 2005).
- Read through *the official PHP Manual* (http://www.php.net/manual/en/) and its various sections listed below.
- *PHP Manual: Session Handling Functions* (http://www.php.net/manual/en/ref.session.php)
- *PHP Manual: DOM Functions* (http://www.php.net/manual/en/ref.dom.php)
- *PHP Manual: XML Parser Functions* (http://www.php.net/manual/en/ref.xml.php)
- *PHP Manual: php.ini directives* (http://www.php.net/manual/en/ini.php)
- Also check out "*SAX-like apps in PHP*" (developerWorks, March 2003).
- Shop for the book *Programming PHP*.
- Shop for the book *Learning PHP*.

- 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's products.
- Innovate your next open source development project with *IBM trial software*, available for download or on DVD.
- Find hundreds of *discounted books on open source topics* in the Open source section of The Developer Bookstore, including many *books on PHP*.
- 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/ .