Learning PHP, Part 2: Upload files and use XML or JSON to store and display file information

Nicholas Chase
(ibmquestions@nicholaschase.com)
Founder
NoTooMi

Skill Level: Intermediate

Date: 21 Jun 2005
(Updated 03 Jan 2013)

This tutorial is Part 2 of a three-part "Learning PHP" series teaching you how to use PHP through building a simple 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 or JSON.
03 Jan 2013 - *Nicholas Chase updated content throughout this tutorial to reflect current PHP, XML, and JSON technology.*

View more content in this series

Section 1. Before you start

In this tutorial you will learn how to use sessions with PHP, how to manipulate XML data with the DOM, and how to create, use, and read JSON data in PHP.

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 will enable users to upload files to the system by using their browsers, and you will use first XML, then JSON to store and display information about each file.

Part 3 looks 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 JavaScript Object Notation (JSON) data.
- Reading and using JSON data.

Who should take this tutorial?

This tutorial is Part 2 in a three-part series designed to teach you various aspects of working with PHP as you build 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 or JSON.

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 form submissions and databases. Familiarity with XML is helpful, but not required. (You can find more information about these topics in 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 then 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 do 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 need a database 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 commonly used with PHP. If you choose to go this route, you can download and install the Community Server.

Section 2. Getting started: creating a session

Let's begin by creating a login page so you can start, populate, and end 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.

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.

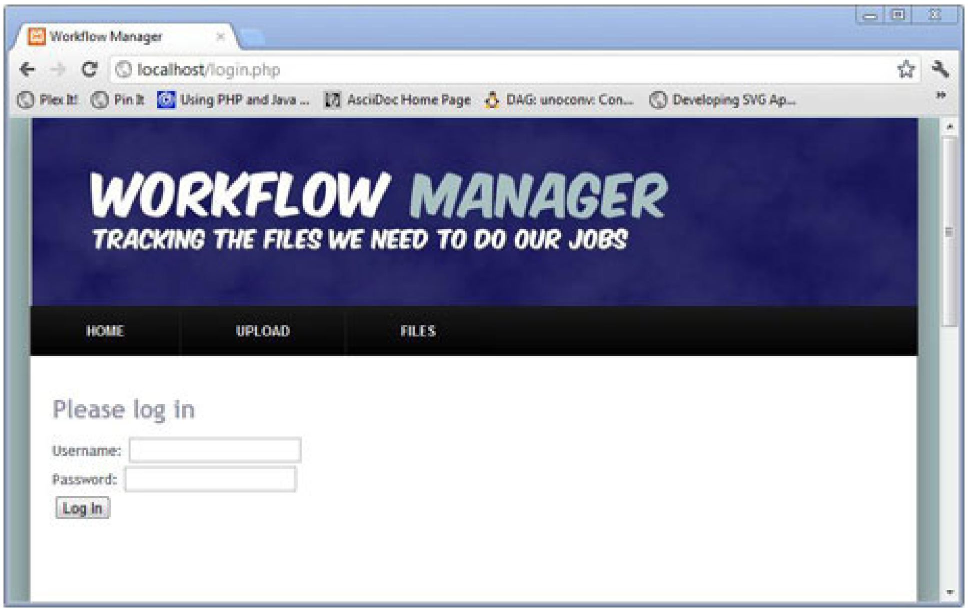
First, create a new blank file, and save it as login.php in the same directory as registration.php. Add the code in Listing 1.

Listing 1. Creating the login form

```
<?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 creates 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, as they're referenced in top.txt and bottom.txt (see Figure 1).

Figure 1. The login.php page with interface elements



The login process, Part 2

Create another new page and save it as login_action.php. Add the code in Listing 2.

Listing 2. Processing login information

```
<?php

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

  $dbh = new PDO('mysql:host=localhost;dbname=workflow', 'wfuser', 'wfpass');

  $stmt = $dbh->prepare("select * from users \
                                where username = :user and password = :pword");

  $stmt->bindParam("user", $name);
  $stmt->bindParam("pword", $pword);

  $name = $_POST["username"];
  $pword = $_POST["password"];

  $stmt->execute();

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

  $dbh = null;

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

After including the interface elements for the top of the page, open a connection to the database and create the PDO object that you'll use to search it. From there, prepare a statement to search for a user account with the username and password submitted by the user.

You might notice a slight difference between this version and the one you used in Part 1. Before, you specified variables by position. Now, you specifically name the parameters by using the colon (:) syntax. Otherwise, it works precisely the same way. After you execute the statement, if the username and password match an existing account, PHP can `fetch()` a row. If not, the fetch fails, and returns `false`.

Starting a session

You ask the user to log in so the site knows which files an individual user can see. As things stand right now, when the user leaves this page, that information is gone. You must do more than just log them in.

You want to create a *session* so the server knows which requests to group together as belonging to a single user. You can also associate information—such as the username—with the session, which is convenient.

To start, create the session using the `session_start()` function. This function first checks to see if a session exists, and if not, it starts one.

Remember one crucial item: The use of `session_start()` or, in fact, anything done to set session information has a major restriction. It is part of the HTTP header, so it must happen before any content, including interface elements, goes to the browser (see Listing 3).

First, add the session and restructure login_action.php:

Listing 3. Adding session information

```
<?php
   session_start();

   require("scripts.txt");

   $dbh = new PDO('mysql:host=localhost;dbname=workflow', 'wfuser', 'wfpass');

   $stmt = $dbh->prepare("select * from users
                                where username = :user and password = :pword");
   $stmt->bindParam("user", $name);
   $stmt->bindParam("pword", $pword);

   $name = $_POST["username"];
   $pword = $_POST["password"];
   $stmt->execute();

   $loginOK = false;
   if ($stmt->fetch()) {
      $loginOK = true;
   }

   $dbh = null;
   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

Unless you have information to pass around from request to request, there's no point in creating a session. In this case, you want to pass the username and the email address, so you add them to the session (see Listing 4).

Listing 4. Saving session information

```
...
if ($row = $stmt->fetch()) {
    $loginOK = true;
    $_SESSION["username"] = $row["username"];
    $_SESSION["email"] = $row["email"];
}
...
```

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.

Using an existing session

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, as in Listing 5.

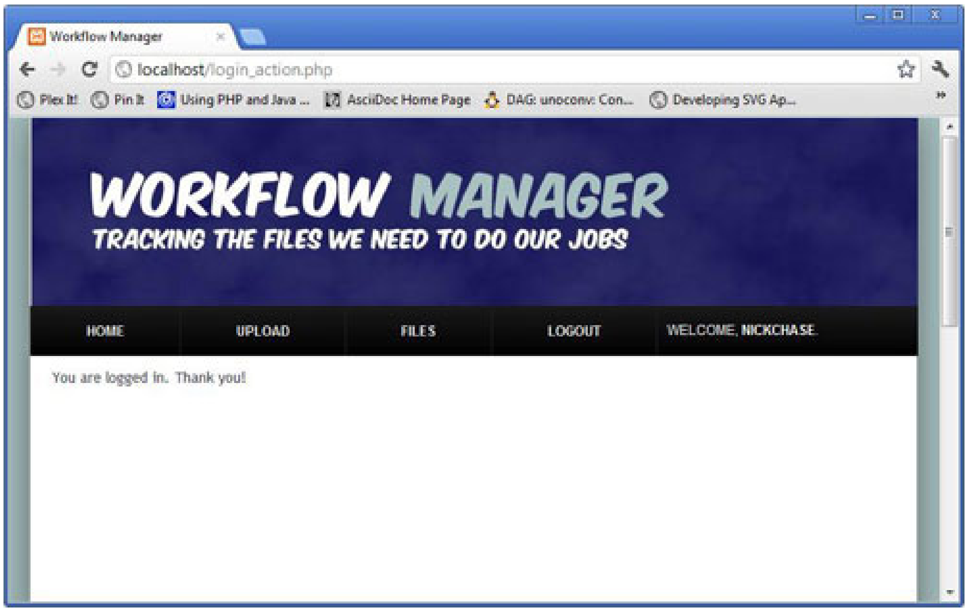
Listing 5. Using session information

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"<
  <head<
    <title<Workflow Manager</title<
    <link rel="stylesheet" type="text/css" href="style.css" /<
  </head<
  <body<
    <div id="wrapper"<<div id="bg"<<div id="header"<</div<
      <div id="page"<<div id="container"<<div id="banner"<</div<
        <div id="nav1"<
          <ul style='float: left'<
            <li<<a href="#" shape="rect"<Home</a<</li<
            <li<<a href="#" shape="rect"<Upload</a<</li<
            <li<<a href="#" shape="rect"<Files</a<</li<
             <?php
```

```
               if (isset($_SESSION["username"]) || isset($username)){
                  if (isset($_SESSION["username"])){
                     $usernameToDisplay =  $_SESSION["username"];
                  } else {
                     $usernameToDisplay = $username;
                  }
            ?<
                  <li shape="rect"<</li<
                  <li<<p style='color:white;'<&nbsp;&nbsp;&nbsp;
                                    Welcome, <?=$usernameToDisplay?<.
</p<</li<
            <?php
               } else {
            ?<
                  <li<<a href="registration.php"
shape="rect"<Register</a<</li<
                  <li<<a href="login.php"
shape="rect"<Login</a<</li<
            <?php
               }
            ?<
          </ul<
        </div<
        <div id="content"<
          <div id="center"<
```

You might wonder 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()` is called. 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


Workflow Manager
localhost/login_action.php
Plex It! Pin It Using PHP and Java ... AsciiDoc Home Page DAG: unoconv: Con... Developing SVG Ap...
WORKFLOW MANAGER
TRACKING THE FILES WE NEED TO DO OUR JOBS
HOME
UPLOAD
FILES
LOGOUT
WELCOME, NICKCHASE

You are logged in. Thank you!

Clearing data

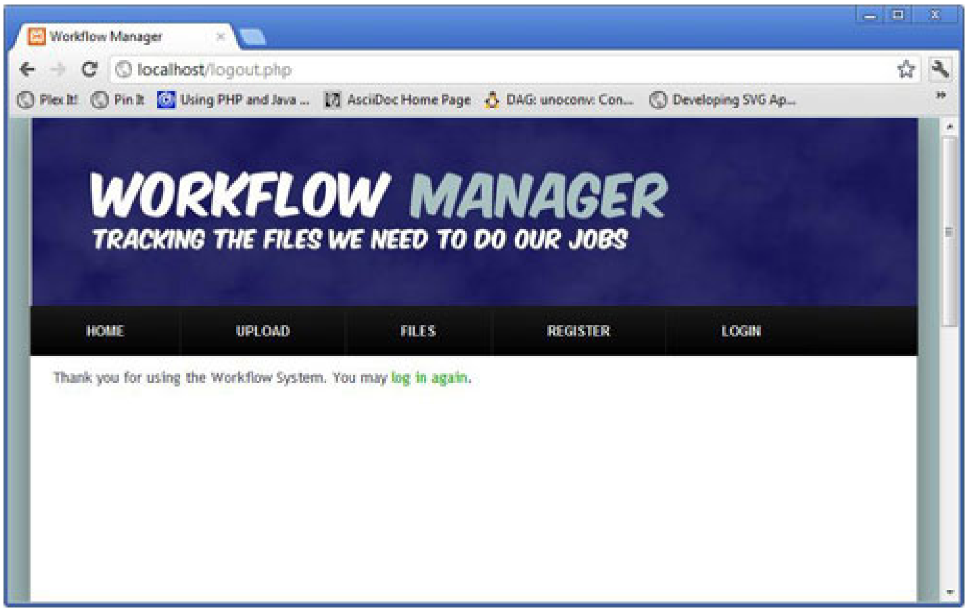
When you finish your session, you can clear out its data or end the session altogether. For example, you might want to log a user out without ending the session. Create a new file called logout.php and add the code in Listing 6.

Listing 6. Clearing session information

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

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

To view this page, point the browser to `http://localhost/logout.php` and show the values were cleared, as in Figure 3.

Figure 3. Page with values cleared



Thank you for using the Workflow System. You may log in again.

A simpler way to clear out data is to end the session (see Listing 7).

Listing 7. Ending a session

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

   include("top.txt");
   echo "Thank you for using the Workflow System.";
   echo "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

In this section you create an upload form for users to upload and save files in the sample workflow application.

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 to upload files is similar to those used for the registration and login pages, with two important exceptions (see Listing 8).

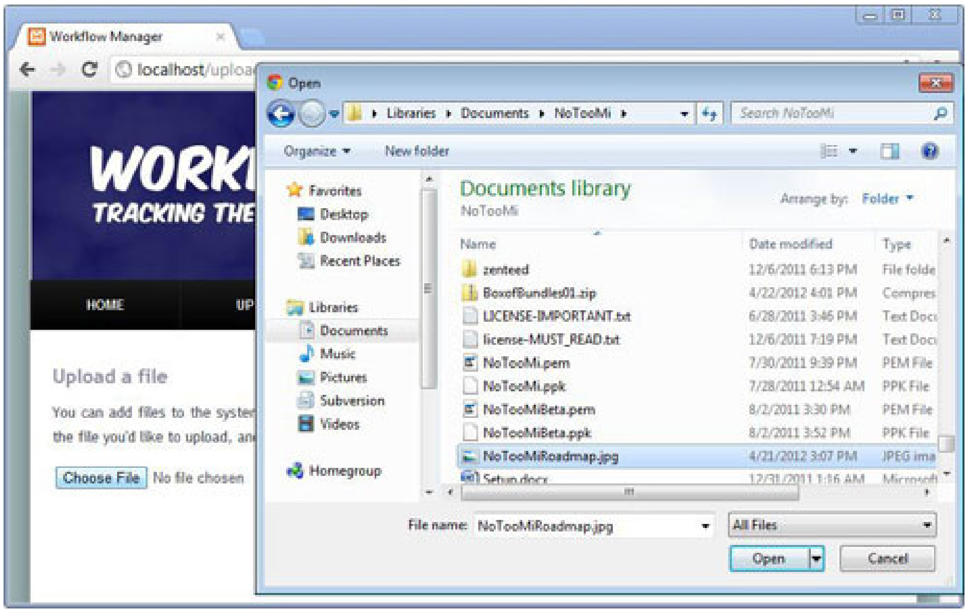
Listing 8. Creating the upload form

```
<?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 in Figure 4.

Figure 4. Choose a file to upload



Upload a file

You can add files to the syster
the file you'd like to upload, an

Name	Date modified	Type
zenteed	12/6/2011 6:13 PM	File folde
BoxofBundles01.zip	4/22/2012 4:01 PM	Compres
LICENSE-IMPORTANT.txt	6/28/2011 3:46 PM	Text Docu
license-MUST_READ.txt	12/6/2011 7:19 PM	Text Docu
NoTooMi.pem	7/30/2011 9:39 PM	PEM File
NoTooMi.ppk	7/28/2011 12:54 AM	PPK File
NoTooMiBeta.pem	8/2/2011 3:30 PM	PEM File
NoTooMiBeta.ppk	8/2/2011 3:52 PM	PPK File
NoTooMiRoadmap.jpg	4/21/2012 3:07 PM	JPEG ima
Setup.docx	12/31/2011 1:16 AM	Microsoft

Add a link to the file in top.txt (see Listing 9).

Listing 9. Adding the upload form to the interface

```
...
   <li><a href="#" shape="rect">Home</a></li>
   <li><a href="uploadfile.php" shape="rect">Upload</a></li>
   <li><a href="#" shape="rect">Files</a></li>
                    ...
```

Now you're ready to take a look at the information that gets uploaded.

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

Since you know what information should be present, verify 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 actually was uploaded. Create the action page for this form, uploadfile_action.php, and add the code in Listing 10.

Listing 10. Checking for an uploaded file

```
<?php

   session_start();

   include("top.txt");

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

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

If the user hasn't specified a file to upload, `$_FILES['ufile']['name']` won't be passed by the browser. (Note that `isset()` also returns false if the value of a variable is `null`.

Next you'll save the file.

Save the file

Before you start to save the uploaded file, decide where to put it. Until the file's been approved, you don't want it accessible from the website, 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. A constant is like a variable, except that once you set it, you can't change its value. Open scripts.txt and add the following definition (see Listing 11).

Listing 11. Creating a constant

```
...
}
   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 (see Listing 12).

Listing 12. Saving the uploaded file

```
<?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");
?>
```

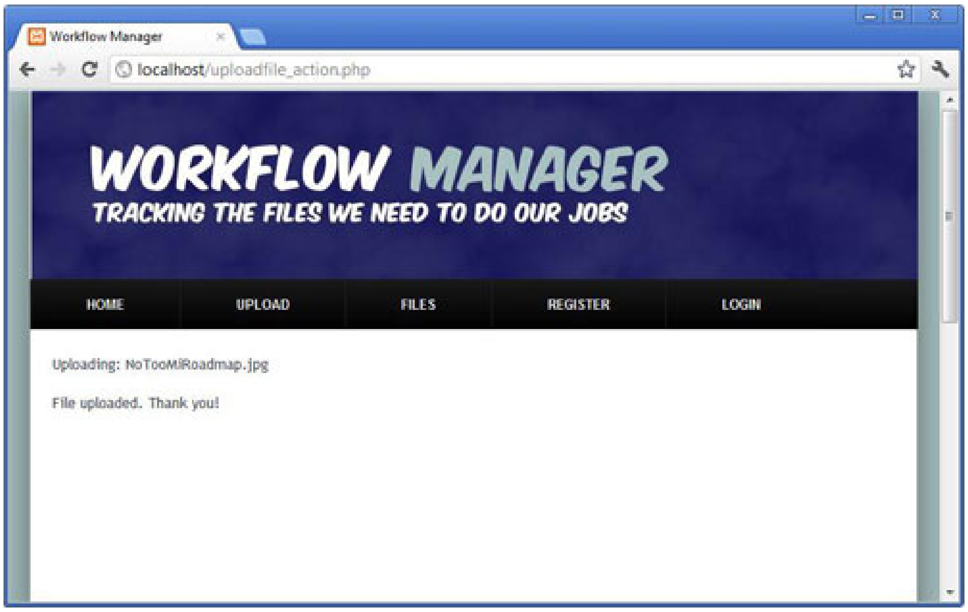
First, get the current location of the file and its temporary name (`tmp_name`), and determine where you want it to go using the constant you defined. (Notice that constants don't start with `$`.)

Next, you do two things in the if-then statement. Check to make sure that the file you're trying to move is actually a file that was uploaded to the server, rather than a file such as /etc/passwd that the user tricked you 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 `is_uploaded_file()` returns `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 the 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 display an error message. Otherwise, you display the success message (see Figure 5).

Figure 5. The success message


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

Uploading: NoTooMiRoadmap.jpg

File uploaded. Thank you!

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

Section 4. Using XML: DOM

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

What is XML?

These days, you can't do much programming without running into some form of XML. Fortunately, XML is an easy subject to understand. You probably already deal with a relative of XML: HTML. Consider this HTML V4.01 page (see Listing 13).

Listing 13. An example of HTML

```
<!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, 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 a `</P>` end tag. (Alternatively, you can write it as the shortcut *empty* `<P />` element.) Also, XML is case-sensitive, so you must close `<P>` with `</P>`, and not `</p>`.

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 Listing 14.

Listing 14. XML describing files

```
<?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>
```

I added spacing here to make things clearer, 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 (see Listing 15).

Listing 15. XML describing a single file

```
<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 white space 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 white space 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. Both the DOM API and the SAX API discussed in the next sections use them,. Before going further, you need 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 Listing 16.

Listing 16. A very simple class

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

When you create an object, you can then reference its functions (see Listing 17).

Listing 17. Creating and using an object

```
$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()`. Start by adding a call to this function in uploadfile_action.php (see Listing 18).

Listing 18. Calling the save_document_info() function

```
  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

Let's continue manipulating the XML file with the help of DOM.

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 code in Listing 19.

Listing 19. Creating the DOM Document

```
...
   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 have a working `Document`, you can use it to create the main, or *root*, element of the document (see Listing 20).

Listing 20. Creating an element

```
...
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 code in Listing 21 to `save_document_info()`.

Listing 21. Saving a DOM Document

```
...
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 Listing 22.

Listing 22. The basic file

```
<?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 the docinfo.xml by virtue of testing the previous step. Until you completely save the first file's information, you can assume that you're still creating the docinfo.xml file.

Start by creating the `statistics` element (see Listing 23).

Listing 23. Setting attributes on an 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 have the `Element` object, `$statistics`, you can use its built-in functions to set two attributes: `total` and `approved`. After you do that, add this element as a child of `$root`, rather than as a child of `$doc`. If you save the file, you can see the difference (see Listing 24).

Listing 24. The resulting file

```
<?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 white space here. The `statistics` element is the first child of `workflow`. (Although you often see XML written as "pretty print," each of those text chunks is a child node.)

Next let's look at adding real information.

Create the file information element

Now you can create the actual file information element. The process uses the techniques you just learned (see Listing 25).

Listing 25. Creating the actual file information

```
...
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 Listing 26.

Listing 26. The resulting document

```
<?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

You now know how to add information to the file and are ready to look at working with the file on subsequent uploads. First check whether the file already exists, then act accordingly (see Listing 27).

Listing 27. Checking to see if the file already exists

```
...
function save_document_info($fileInfo){
   $doc = new DOMDocument('1.0');

   $xmlfile = UPLOADEDFILES."docinfo.xml";

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

}
```

In Listing 27, you create a variable to represent the location of the file, since you'll now refer to it in more than one place. Next, verify whether that file already exists. If it does, call the `load()` function rather than creating a new object.

This static function—which I 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, simply add the new `fileInfo` element, and it will appear after the original. See Listing 28, with space added for clarity.

Listing 28. The file, with added information

```
<?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 (see Listing 29).

Listing 29. Updating the statistics attribute

```
...
   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, you get a reference to the existing `statistics` element the same way you got a reference to the existing `workflow` element. In this case, 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, as shown in Listing 30.

Listing 30. The resulting file

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

While technically correct, it is **more** correct to set the number of items by doing a count after the new element is added. Then you only have to set it once (see Listing 31).

Listing 31. Counting elements

```
...
   if(is_file($xmlfile)){
      $doc->load($xmlfile);
      $workflowElements = $doc->getElementsByTagName("workflow");
      $root = $workflowElements->item(0);
   } else{
...
   $root->appendChild($fileInfo);

   $statistics = $root->getElementsByTagName("statistics")->item(0);
   $total =  $root->getElementsByTagName("fileInfo")->length;
   $statistics->setAttribute("total",$total);

   $doc->save($xmlfile);
}
```

In this case, you moved statistics processing to the bottom, after the `fileInfo` element is added. You can then get a list of all of the `fileInfo` elements, and know how many by referencing the `length` property. You'll learn more about properties in Part 3 of this series. For now you can think of a property as a variable assigned to an object. You can tell it's a property and not a function because it doesn't have parentheses after it.

Now that you know how to use XML to create a file, look at an alternate method for storing hierarchical data: JSON.

Section 6. Using JSON

Take a look at an alternate way to store our data, this time using JSON instead of XML.

What is JSON?

JavaScript Object Notation, or JSON, is a way to specify information that lets you store data much in the same way you store it with XML, but in a much less verbose way. For example, a JSON representation of the data store looks something like Listing 32.

Listing 32. A sample JSON file

```
{
   "statistics" : {
                  "total" : 2, "approved": 0
                  },
   "fileInfo" : [
       {
         "status" : "pending",
         "submittedBy" : "nickChase",
         "approvedBy" : "",
         "fileName" : "NoTooMiRoadmap.jpg",
         "location" : "\/var\/www\/hidden\/",
         "fileType" : "image\/jpeg",
         "size" : 12813
       },
       {
         "status" : "pending",
         "submittedBy" : "stanley",
         "approvedBy" : "",
         "fileName" : "NoTooMiBeta.pem",
         "location" : "\/var\/www\/hidden\/",
         "fileType" : "application\/octet-stream",
         "size" : 1692
       }
   ]
}
```

Look at a couple of different notations. Rather than having a `workflow` root element, the data is part of a single object, and objects are delimited by curly braces (`{}`). So the overall object is shown within the outer braces.

An object has one or more properties. In this case, the properties are `statistics` and `fileInfo`.

The value of the `statistics` property [shown after the colon (`:`)] is an object, which you can tell because it's wrapped in curly braces `{}`. That object itself has two properties, `total` and `approved`, both of which have integer values. The properties are delimited by commas. Notice also that all strings—including the property names—are wrapped in quotes.

The value of the `fileInfo` property is a little different; it's not an object, it's an array of objects, and you can tell that because it's wrapped in straight brackets (`[]`). In this case, it's an array of objects, each of which is wrapped in curly braces, which brings you right back to where you started.

To summarize:

1. Objects are wrapped in curly braces.
2. Objects are collections of properties, name-value pairs separated by a colon and delimited by commas.
3. Arrays are collections of objects or properties, separated by commas and wrapped in brackets.

JSON objects are also capable of having properties that are functions, but that's beyond the scope of this tutorial. (See Resources for more information.)

Now you're ready to look at actually using this data.

How to use JSON data in PHP

The process of using JSON data in PHP generally has three steps:

1. Create a PHP object that holds the data in a structure that matches the object you want to create. In some cases, you'll create that structure from scratch; in others you'll get it from an existing JSON data store.
2. Manipulate the PHP object. This might consist of changing values, or it might consist of adding or removing data.
3. Convert the PHP object to JSON and save it.

You will take these three steps using the workflow data.

Arrays versus Objects

What you see above is a text-based serialization of an object. To work with it in PHP, you need to turn it into an actual PHP object, which you can do in two ways.

- The first is to turn it into an actual object, which uses PHP's object notation, such as:
  ```
  $fileInfo->size
  ```
- The second is to use array notation, which looks at the data as an associative array, which you saw earlier, as in:
  ```
  $fileInfo["size"]
  ```

Which one you use depends on how you create the variable within PHP.

For two reasons you will use the array notation for this example. The first, and most obvious, is that you're familiar with arrays, but you haven't covered objects yet. The second is that certain operations, such as adding to a JSON array, are difficult or impossible using the object notation.

Preparing the JSON object

The first step is to create the basic PHP array to hold your data. To make things simple, create the equivalent of one `fileInfo` element. In the scripts.txt file, create the function in Listing 33.

Listing 33. Preparing the basic JSON object

```
function save_document_info_json($file){

   $filename = $file["name"];
   $filetype = $file["type"];
   $filesize = $file["size"];

   $fileInfo["status"] = "pending";
   $fileInfo["submittedBy"] = $_SESSION["username"];
   $fileInfo["approvedBy"] = "";
   $fileInfo["fileName"] = $filename;
   $fileInfo["location"] = UPLOADEDFILES;
   $fileInfo["fileType"] = $filetype;
   $fileInfo["size"] = $filesize;

}
```

Although it's been re-arranged a bit, nothing here is really new. First you extract the values from the `$file` array. Then you assign them to the newly (and dynamically) created `$fileInfo` array.

This `$fileInfo` array represents a single object; if you saved it out as JSON (which you'll do shortly) it looks something like Listing 34.

Listing 34. What the $fileInfo object looks like

```
{
   "status" : "pending",
   "submittedBy" : "stanley",
   "approvedBy" : "",
   "fileName" : "NoTooMiBeta.pem",
   "location" : "\/var\/www\/hidden\/",
   "fileType" : "application\/octet-stream",
   "size" : 1692
}
```

Next you'll see how to deal with nested properties.

Creating and saving JSON

So far you've dealt with a simple object, with simple values. Now you will see what happens when you delve a little deeper.

In the previous example, you built a single `fileInfo` object. Ultimately, however, you'll build the overall `workflow` object, which includes the `statistics` property. But the `statistics` property is an object that itself has properties (see Listing 35).

Listing 35. Properties of properties

```
function save_document_info_json($file){

   $workflow["statistics"]["total"] = 1;
   $workflow["statistics"]["approved"] = 0;

   $filename = $file['name'];
...
```

As you can see, you create the `$workflow` variable, then the `statistics` property. Once you have that, you can then specify the `total` and `approved` properties.

You can proceed down this path to any level (within reason).

Adding an item to an array

Now that you have the `$workflow` variable, you need to add the `$fileinfo` object to its `fileInfo` property. Remember, `fileInfo` is an array, so you can use the `array_push()` function to do that (see Listing 36).

Listing 36. Adding an item to an array

```
function save_document_info_json($file){

   $workflow["fileInfo"] = array();
   $workflow["statistics"]["total"] = 1;
......
   $fileInfo["fileType"] = $filetype;
   $fileInfo["size"] = $filesize;

   array_push($workflow["fileInfo"], $fileInfo);
}
```

The `array_push()` function adds one or more items to the target array. In this case, that array is `$workflow["fileInfo"]`, which you created as an empty array at the top of the script. This is a typical (non-associative) array, so the first time you add an item, you can then go back and refer to it as:

```
$workflow["fileInfo"][0]
```

Now you're ready to look at saving the data.

Serializing and saving JSON

At this point, you created a variable that represents the data structure you want to save as JSON, so it's time to look at serializing and saving that data.

PHP actually makes it incredibly easy to turn a variable such as `$workflow` into JSON text by providing the `json_encode()` (see Listing 37).

Listing 37. Adding an item to an array

```
function save_document_info_json($file){

   $jsonFile = UPLOADEDFILES."docinfo.json";

   $workflow["fileInfo"] = array();
   $workflow["statistics"]["total"] = 1;
...
   $fileInfo["fileType"] = $filetype;
   $fileInfo["size"] = $filesize;

   array_push($workflow["fileInfo"], $fileInfo);

   $jsonText = json_encode($workflow);
   file_put_contents($jsonFile, $jsonText);

}
```

The `json_encode()` function converts the `$workflow` object to JSON text, and then `file_put_contents()` saves that text to the `docinfo.json` file.

You just tell the upload_action.php page to call this new function instead of the XML-based function (see Listing 38).

Listing 38. Calling the new function

```
...
       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_json($_FILES['ufile']);

       }
...
```

Now you can upload a new document and check the contents of the `docinfo.json` file. It should look something like Listing 39 (spaces added for clarity).

Listing 39. The resulting JSON file

```
{
 "statistics":{"total":1,"approved":0},
 "fileInfo":[
       {"status":"pending",
        "submittedBy":"stanley",
        "approvedBy":"",
        "fileName":"NoTooMiBeta.pem",
        "location":"c:\/sw\/temp\/",
        "fileType":"application\/octet-stream",
        "size":1692}
       ]
}
```

So you didn't have to do any work to create the JSON notation; the `json_encode()` function did it for you. Getting a variable back out is just as easy, and you'll see how to do that next.

Reading JSON from a file

Now that you can write JSON, how about reading it? Just as when dealing with XML, you need to add new `fileInfo` objects to an existing `docinfo.json` file. To do that, you can use relatives of functions that you've already seen (see Listing 40).

Listing 40. The resulting JSON file

```
function save_document_info_json($file){

   $jsonFile = UPLOADEDFILES."docinfo.json";

  if (is_file($jsonFile)){
      $jsonText = file_get_contents($jsonfile);
      $workflow = json_decode($jsonText, true);
   } else{
      $jsonText = '{"statistics": {"total": 0, "approved": 0}, "fileInfo":[]}';
      $workflow = json_decode($jsonText, true);
   }

   $filename = $file['name'];
...
   array_push($workflow["fileInfo"], $fileInfo);

   $total =  count($workflow["fileInfo"]);
   $workflow["statistics"]["total"] = $total;
   $jsonText = json_encode($workflow);
   file_put_contents($jsonFile, $jsonText);

}
```

Just as before, you first check to see if the file exists. If it does, rather than create the `$workflow` variable from scratch, you first extract the existing JSON text using the `file_get_contents()` function, then use the `json_decode()` function to turn that into a PHP variable.

Note the second parameters in the `json_decode()` function, which tells PHP whether to return an array or not. The default value (in other words, what the function will use if you don't provide a value) is `false`, which means that PHP will **not** return an array, and will return an object instead. In this case, you wanted an array, so you provide a value of `true`.

Just for the sake of an example, if the file doesn't exist, and you're creating things from scratch, this example shows that you can just provide plain JSON text—after all, that's what you pull out of the `docinfo.json` file anyway!

Finally, before you save the file, you can find out the number of `fileInfo` objects by doing a `count()` on that array. You can then set that value as the `total`.

Before moving on to actually using all this data, you need to know one very important thing.

A crucial note about security

The `json_decode()` function doesn't execute any code; it simply loads data. That doesn't mean it's immune to security problems, because like other string-handling functions that are vulnerable to buffer overflows and other techniques, it can be used for evil. That's an extremely rare situation.

The bigger problem is when you provide JSON data back to a JavaScript implementation. You might be tempted to use JavaScript's `eval()` function to turn it into a JavaScript object. Unless you know **precisely** what's in that data and that you can trust it, ***don't***. Use a JSON library intended for this purpose instead.

The overall display

Now you can put together everything that you've learned so far to build a script that reads the JSON and displays the list of files on the upload_action.php page. Start by creating the function in scripts.txt (see Listing 41).

Listing 41. Reading the data

```
function display_files(){

   echo "<table width='100%'>";
   echo "<tr><th>File Name</th>";
   echo "<th>Submitted By</th><th>Size</th>";
   echo "<th>Status</th></tr>";

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

   $files = $workflow["fileInfo"];

   for ($i = 0; $i < count($workflow["fileInfo"]); $i++){


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

First get the `$workflow` object by using `json_decode()` to analyze the contents of the `docinfo.json` file.

Once you have that, you can get the array that represents all the files by extracting the `fileInfo` property, then loop through each one, displaying its information.

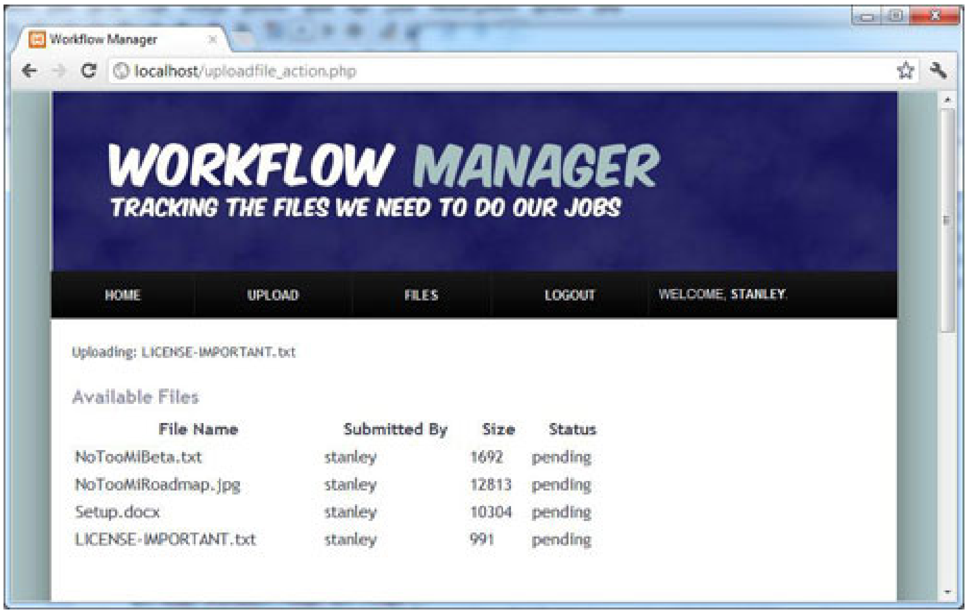
Now you need to add the function to the upload_action.php file (see Listing 42).

Listing 42. Calling the display function

```
...
            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_json($_FILES['ufile']);

           echo "<h3>Available Files</h3>";
           display_files();
       }

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

The end result is an HTML table that should look something like Figure 6.

Figure 6. Readable table of information


Workflow Manager
localhost/uploadfile_action.php
WORKFLOW MANAGER
TRACKING THE FILES WE NEED TO DO OUR JOBS
HOME
UPLOAD
FILES
LOGOUT
WELCOME, STANLEY.

Uploading: LICENSE-IMPORTANT.txt

Available Files

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

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

Section 7. 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 first XML, then JSON 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 JSON data
- Manipulating data using JSON
- Saving and loading text files

In Part 3, you'll complete the application.

Downloads

Description	Name	Size	Download method
Part 2 source code	PHPPart2SampleFiles.zip	65KB	HTTP

Information about download methods

Resources

Learn

- For an excellent primer on objects in PHP, read Using PHP Objects.
- Read Part 1 of this series: "Learning PHP, Part 1" and learn to register for an account, upload files for approval, and view and download approved files.
- Read Part 3 of this series: "Learning PHP, Part 3" to explore authentication, objects, exceptions, and streaming.
- Read through the official PHP Manual.
- Check out PHP Manual: Session Handling Functions.
- Read PHP Manual: DOM Functions.
- Don't miss PHP Manual: XML Parser Functions.
- And see PHP Manual: php.ini directives.
- Lean more about JavaScript Object Notation (JSON).
- Visit IBM developerWorks' PHP project resources to learn more about PHP.
- Stay current with developerWorks technical events and webcasts.
- Check out upcoming conferences, trade shows, webcasts, and other Events around the world that are of interest to IBM open source developers.
- Visit the developerWorks Open source zone for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.
- Listen to developerWorks podcasts for interesting interviews and discussions for software developers.
- 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).