CHAPTER 8

Adding the Ability to Upload Images

Up to this point, you've been dealing exclusively with text on your blog. However, you can make your blog much more interesting by incorporating images.

I'll cover a lot of information in this chapter. For example, I'll explain how to use PHP's GD library (in the section on resizing images), as well as functions that manipulate the file system. I'll even begin to cover the relatively advanced topic of your object-oriented programming.

By the end of this chapter, your blog should be able to:

- Accept image uploads
- Save uploaded images in a folder of your choosing
- Save the uploaded image's path in the database
- Display images to your blog's users
- Resize images to fit into your layout
- Delete images when necessary

Adding a File Input to the Admin Form

Before you can start processing images with PHP, you must first add the ability to upload images to your administrative form on `admin.php`. To do this, you' need to add a file upload input to your administrative form.

When using file inputs, you also have to change the *enctype*, or content type, of the form. By default, HTML forms are set to `application/x-www-form-urlencoded`. However, this won't work when you're uploading files; instead, you need to set the `enctype` of the form to multipart/form-data, which can accept files *and* standard form values.

Modify the form in `admin.php` to include the code in bold:

```
<form method="post"
    action="/simple_blog/inc/update.inc.php"
    enctype="multipart/form-data">
    <fieldset>
        <legend><?php echo $legend ?></legend>
        <label>Title
            <input type="text" name="title" maxlength="150"
                value="<?php echo $title ?>" />
        </label>
        <label>Image
            <input type="file" name="image" />
        </label>
        <label>Entry
            <textarea name="entry" cols="45"
                rows="10"><?php echo $entry ?></textarea>
        </label>
        <input type="hidden" name="id"
            value="<?php echo $id ?>" />
        <input type="hidden" name="page"
            value="<?php echo $page ?>" />
        <input type="submit" name="submit" value="Save Entry" />
        <input type="submit" name="submit" value="Cancel" />
    </fieldset>
</form>
```

Load `admin.php` in a browser to see the added file input.

Accessing the Uploaded File

File uploads work differently than standard inputs, which means you must handle them a little differently. In this case, you must modify `update.inc.php` to deal with uploaded images.

You don't want to force your users to upload an image to post a new entry, so you need to check whether a file was uploaded *after* you verify that the rest of the entry form was filled out. To check whether a file was uploaded, you look in the `$_FILES` superglobal array.

A Quick Refresher on the $_FILES Superglobal Array

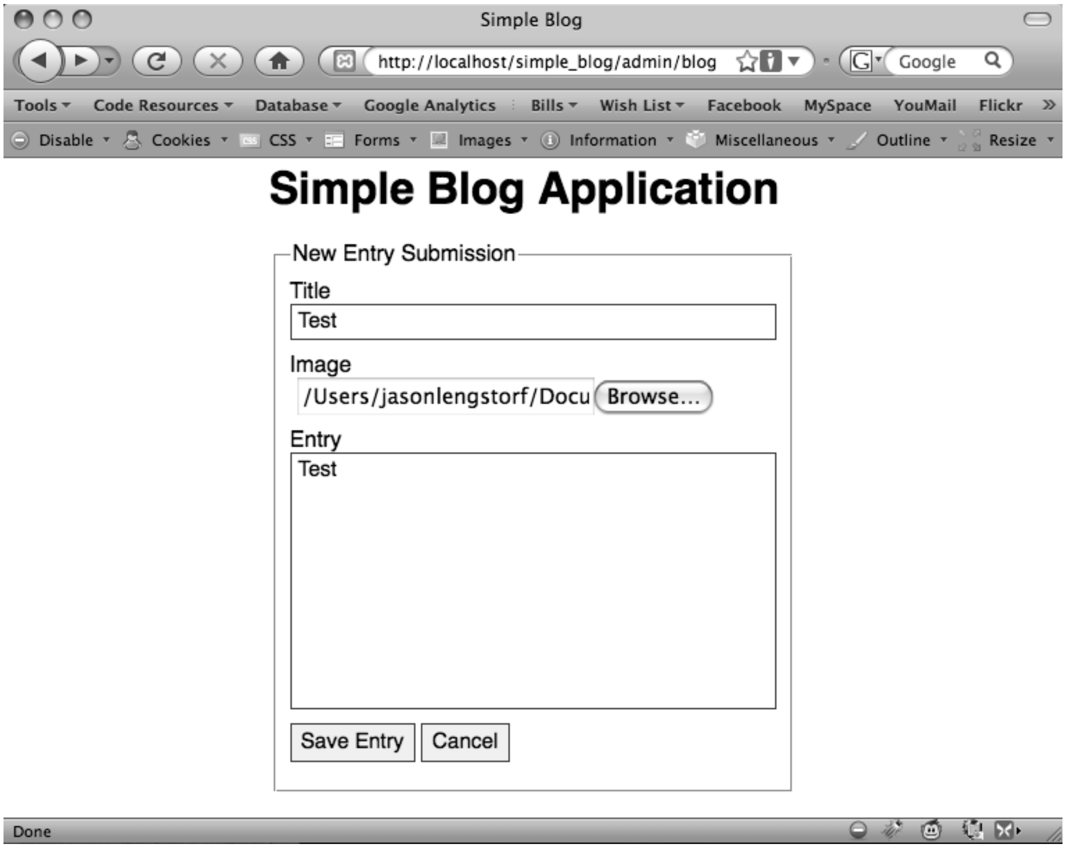
You learned about the `$_FILES` superglobal in Chapter 3, but it might be helpful to review what it does before moving on.

Whenever a file is uploaded via an HTML form, that file is stored in temporary memory and information about the file is passed in the `$_FILES` superglobal. You can see this in action by taking a look at what's being passed to your `update.inc.php` script. To do this, add the code in bold to the top of the file:

```
<?php

// Include the functions so you can create a URL
include_once 'functions.inc.php';

if($_SERVER['REQUEST_METHOD']=='POST'
    && $_POST['submit']=='Save Entry'
    && !empty($_POST['page'])
    && !empty($_POST['title'])
    && !empty($_POST['entry']))
{
    // Create a URL to save in the database
    $url = makeUrl($_POST['title']);

    // Output the contents of $_FILES
    echo "<pre>"; // <pre> tags make the output easy to read
    print_r($_FILES);
    echo "</pre>";

    /*
     * You don't want to save this test, so you exit the script to
     * prevent further execution.
     */
    exit;
```

After you save this file, you can navigate to create a new entry and try uploading an image by pointing your browser to `http://localhost/simple_blog/admin/blog`.

Click the Browse... button on the file input and select any image off your computer. You can fill in the Title and Entry fields with any information you want because this information won't be saved (see Figure 8-1).



Simple Blog Application

New Entry Submission

Title: Test

Image: /Users/jasonlengstorf/Docu Browse...

Entry: Test

Save Entry | Cancel

Figure 8-1. *An image selected in the new file input in **`admin.php`***

Click the Save Entry buttonto send this information to `update.inc.php`, which will display the contents of `$_FILES` and stop execution. The output will look similar to this:

```
Array
(
   [image] => Array
      (
          [name] => IMG001.jpg
          [type] => image/jpeg
          [tmp_name] => /Applications/XAMPP/xamppfiles/temp/phpHQLHjt
          [error] => 0
          [size] => 226452
      )

)
```

Examining this output enables you to determine that your image is stored in the array located at `$_FILES['image']` (named for the `name` attribute of the file input) and that five pieces of information were stored:

- `$_FILES['image']['name']` : The name of the uploaded file
- `$_FILES['image']['type']` : The MIME type of the uploaded file
- `$_FILES['image']['tmp_name']` : The temporary location of the uploaded file
- `$_FILES['image']['error']` : An error code for the upload (0 means no error)
- `$_FILES['image']['size']` : The size of the uploaded file in bytes

You're ready to start processing the uploaded image now that you know what information is being sent from your form to `update.inc.php`.

Object-Oriented Programming

To handle your uploaded images, you need to leverage *object-oriented programming* (OOP) to write a class that handles your image processing. Classes and objects are extremely useful in programming; both have been well supported in PHP scripts since the release of PHP 5.

Drill Down on Objects

An object is a collection of information in a program. An object behaves similarly to an array, except that the information in an object provides a little more flexibility in how the information can be accessed and processed. To use objects in PHP, you must define a *class* to provide a structure for the object.

Classes vs. Objects

It's not uncommon to hear the terms "class" and "object" used interchangeably when talking about programming, but it's important to recognize that classes and objects are *not* the same thing, albeit they are bound tightly together.

A class is a blueprint for information. It provides a map of properties and methods that give a collection of information meaning. However, a class by itself doesn't really mean anything.

Objects contain a mass of information that conforms to the blueprint laid out by the class: an object is to a class as a house is to a blueprint. Each object is an instantiation of a class—just as each house built from a blueprint is an instantiation of the blueprint.

Here's another example that might help you understand the difference between a class and an object. Imagine that you have a toy robot that stores one *property*, or class-specific variable (its name) and one *method*, or class-specific function (the ability to write its name). The methods and properties that describe and define your robot are the class. However, before your toy robot can actually perform any of its abilities, you must first supply it with its required information—its name, in this case. The robot's name represents an object.

Why Objects Are Useful

One neat feature about objects: they enable you to have multiple instances of a given class that run simultaneously, but use different information.

Let's continue with the toy robot example. If you have two toy robots that are exactly the same, you can store the name "Tom" in the first robot, and the name "Jim" in the second, and they will both perform their duties in the same way, but with different information. If you set them loose at the same time, they will both start writing their names—Tom and Jim, respectively.

The advantage of using classes and objects in programming over procedural code is that an object can save its state. This means that you can set a property in an object once, then reference it for as long as the object exists.

In procedural code, you would need to pass that same property to every function to take advantage of it. This makes function calls cumbersome, and this process can prove confusing if the same functions must handle two or more sets of data.

Imagining Your Robot as a PHP Class

Let's explore the benefits of OOP by building your toy robots with PHP.

Begin by defining the class, which you should call `ToyRobot`. Do this using the reserved `class` keyword, followed by the class name, and then a pair of curly braces to enclose the class's properties and methods.

Next, create a new test file called `ToyRobot.php` in your `simple_blog` project, then define the class by adding the following code:

```
<?php

class ToyRobot
{

}

?>
```

Class Properties

A property in a PHP class behaves similarly to a regular variable. The difference is that a class property is specific to the instance of your class and available to the methods without being passed as an argument.

In your `ToyRobot` class, you have only one property: the robot's name. You want to control how the robot's name is accessed and manipulated, so set this property as *private*, which ensures that only the `ToyRobot` class's methods can access the property's value.

MEMBER VISIBILITY IN OOP

Depending on how you use your classes, you need to assign the appropriate visibility declarations to their methods and properties. The available visibility declarations include:

- `public`: Public properties and methods can be accessed anywhere in a script after the object has been instantiated
- `protected`: Protected properties and methods can be accessed only within the class that defines them, parent classes, or inherited classes (you can learn more about inheritance in the PHP manual at `http://us.php.net/manual/en/language.oop5.basic.php#language.oop5.basic.extends`)
- `private`: Private properties and methods can only be accessed by the class that defines them

You must assign all properties with a visibility declaration. Methods declared without a declaration default to `public` visibility. In PHP4, developers used the `var` keyword to define object properties because visibility declarations weren't supported yet. For the sake of backward compatibility, PHP5 recognizes `var` as an alias for `public`, although it does raise an `E_STRICT` warning. For more information on visibility declarations, read the entry on it in the PHP manual, which you can find at `http://us.php.net/manual/en/language.oop5.visibility.php`.

To declare a private property in your `ToyRobot` class, you need to add the lines in bold to `test.php`:

```
<?php

class ToyRobot
{
    // Stores the name of this instance of the robot
    private $_name;
}

?>
```

■ **Note** Using an underscore before a private property name (`$_name`) is a widely accepted way of denoting private properties.

Each instance of the `ToyRobot` class will likely have a different name, so you don't specify a value for the `$_name` property; instead, you accomplish this using the object's *constructor*.

Class Constructors

PHP5 introduced *magic methods*, which are essentially methods that execute when a certain action occurs in a script. There are several available, but this book won't delve into them because they don't apply to your blogging application. You can learn more about magic methods in its PHP manual entry at `http://us.php.net/manual/en/language.oop5.magic.php`.

Constructors are one of the most commonly used magic methods available in PHP5. A constructor is called when an object is *instantiated*, or first created. This allows you to initialize certain parameters or other settings for an object when you create the object. You can leverage a constructor in your `ToyRobot` class by declaring it in the `__construct()` magic method. You use this constructor to define the `$_name` property.

Test this by adding the following code in bold to `test.php`:

```
<?php

class ToyRobot
{
    // Stores the name of this instance of the robot
    private $_name;

    // Sets the name property upon class instantiation
    public function __construct($name)
    {
        $this->_name = $name;
    }
}

?>
```

Note the use of `$this` in your class. In object-oriented PHP, `$this` is a reserved variable that refers to the current object.

`$this` enables an object to refer to itself without needing to know what it's called. This enables multiple instances of a class to exist without conflict.

Next, consider the use of an arrow (`->`) after the `$this` keyword. This arrow indicates that you're accessing a property or method contained within the object. Again, this enables you to have multiple instances of a class without conflict.

Finally, the preceding script lets you add the name of the property or method you want to access (the `$_name` property, in this case). When accessing properties, you leave off the dollar sign because you already use it with `$this`.

Your declared constructor will now allow you to define the name of each `ToyRobot` instance right as you create it. The next step is to create the method that enables your `ToyRobot` class to write its name.

Class Methods

You define a method in your `ToyRobot` class much as you declare a function in a procedural script. The only difference is the visibility declaration and the ability to access class properties without needing to accept them as function arguments.

You en able your ToyRobot to write its name by defining a public method called writeName(). Add the code in bold to test.php:

```
<?php

class ToyRobot
{
    // Stores the name of this instance of the robot
    private $_name;

    // Sets the name property upon class instantiation
    public function __construct($name)
    {
        $this->_name = $name;
    }

    // Writes the robot's name
    public function writeName()
    {
        echo 'My name is ', $this->_name, '.<br />';
    }
}

?>
```

This method is straightforward: when called, it causes the robot to introduce itself. All that's left is to learn how to use objects in your scripts.

Using Classes in Your Scripts

The toy robot example initially discussed two robots: Tom and Jim. You're now ready to use your new PHP skills to build this pair of robots.

You use the new keyword to create a new instance of an object, followed by the name of the class. You use the name of the class like a function call, which allows you to pass arguments (assuming any are necessary) to your constructor.

Begin building Tom by adding the lines in bold to test.php:

```
<?php

// Create an instance of ToyRobot with the name "Tom"
$tom = new ToyRobot("Tom");

class ToyRobot
{
    // Stores the name of this instance of the robot
    private $_name;
    // Sets the name property upon class instantiation
    public function __construct($name)
    {
        $this->_name = $name;
    }

    // Writes the robot's name
    public function writeName()
    {
        echo 'My name is ', $this->_name, '.<br />';
    }
}

?>
```

At this point, you have created and instantiated your first PHP class. Passing Tom as an argument to the object enables you to set the private $_name property to Tom, effectively giving your robot its name.

Now you can have Tom write his name by calling his writeName() method. Add the lines in bold to test.php:

```
<?php

// Create an instance of ToyRobot with the name "Tom"
$tom = new ToyRobot("Tom");

// Have Tom introduce himself
$tom->writeName();

class ToyRobot
{
    // Stores the name of this instance of the robot
    private $_name;

    // Sets the name property upon class instantiation
    public function __construct($name)
    {
        $this->_name = $name;
    }

    // Writes the robot's name
    public function writeName()
    {
        echo 'My name is ', $this->_name, '.<br />';
    }
}

?>
```

Now you have some actual output to look at. Load `test.php` in your browser to see what Tom had to say:

```
My name is Tom.
```

You're now ready to build Jim and have him introduce himself; add the bold to `test.php`:

```
<?php

// Create an instance of ToyRobot with the name "Tom"
$tom = new ToyRobot("Tom");

// Have Tom introduce himself
$tom->writeName();

// Build Jim and have him introduce himself
$jim = new ToyRobot("Jim");
$jim->writeName();

class ToyRobot
{
    // Stores the name of this instance of the robot
    private $_name;

    // Sets the name property upon class instantiation
    public function __construct($name)
    {
        $this->_name = $name;
    }

    // Writes the robot's name
    public function writeName()
    {
        echo 'My name is ', $this->_name, '.<br />';
    }
}

?>
```

You should see the following when you load `test.php` in a browser:

```
My name is Tom.
My name is Jim.
```

Your ToyRobot class is admittedly simple, but it exemplifies the power of using objects with PHP. You can have two instances in use at the same time without any conflicts and without needing to keep track of a bunch of variables.

Even if your objects are mixed in together, you still get the proper result. For instance, you can replace the class instantiations in test.php with the following:

```
<?php

$tom = new ToyRobot("Tom");
$jim = new ToyRobot("Jim");

$tom->writeName();
$jim->writeName();

?>
```

This still outputs the following:

```
My name is Tom.
My name is Jim.
```

Armed with your new understanding of OOP, you're ready to take on a new task: writing a class to handle images.

Writing the Image Handling Class

Your first step, of course, is to define the ImageHandler class. You want this class to be portable, so you should create a separate file for it called images.inc.php. You save this file in the inc folder (full path: /xampp/htdocs/simple_blog/inc/images.inc.php). After you create images.inc.php, insert the following code to define the class:

```
<?php

class ImageHandler
{

}

?>
```

Saving the Image

So far you've defined your class is defined; next, you need to determine what methods and properties you need to define.

You need a public method that saves the uploaded image in a folder and returns its path. You also need a public property that stores the folder you want your images saved in. This property is vital, so set it in your constructor.

To be as descriptive as possible, call your method for uploading an image `processUploadedImage`.. Call the property that holds the folder's location `$save_dir`.

In `images.inc.php`, define your properties and constructor first by adding the lines in bold:

```
<?php

class ImageHandler
{
    // The folder in which to save images
    public $save_dir;

    // Sets the $save_dir on instantiation
    public function __construct($save_dir)
    {
        $this->save_dir = $save_dir;
    }
}

?>
```

At this point, instantiating the `ImageHandler` class gives you an object that sets a folder where you can save your images. At last you're ready to save an image. Do this by creating your public `processUploadedImage()` method.

Next, define the method by adding the lines in bold to `images.inc.php`:

```
<?php

class ImageHandler
{
    // The folder in which to save images
    public $save_dir;

    // Sets the $save_dir on instantiation
    public function __construct($save_dir)
    {
        $this->save_dir = $save_dir;
    }
    /**
     * Resizes/resamples an image uploaded via a web form
     *
     * @param array $upload the array contained in $_FILES
     * @return string the path to the resized uploaded file
     */
    public function processUploadedImage($file)
    {
        // Process the image
    }
}

?>
```

■Note The block comment immediately before `processUploadedImage()` is a special comment known as a PHP DocBlock. This is a special comment that provides information about a class, property, or method. DocBlocks are indicated by opening a comment using `/**`, then providing a short description of the class, method, or property. This process also lists a method's list of parameters and its return value. All classes you write from now on will take advantage of DocBlocks to help you keep track of the code you're writing. An additional point of interest: Some IDEs and SDKs (including Eclipse) use DocBlocks to provide descriptions of methods as they're used, including a list of parameters (defined using `@param [datatype] [var_name] [description]`) and the return value (defined using `@return [datatype] [description]`).

You accept the file as an argument, which is the array you find in `$_FILES`. To process this file, you need to break the array apart into individual values. You do this using the `list()` function, which allows you to create named variables for each array index as a comma-separated list.

What that means becomes clear if you look at how you previously defined an array:

```
$array = array(
    'First value',
    'Second value'
);
```

The array values are separated by `list()` as follows:

```
list($first, $second) = $array;
echo $first, "<br />", $second;
```

This produces the following output:

```
First value
Second value
```

In processUploadedImage(), you need to pull out the five pieces of information supplied about your uploaded file. You can accomplish this by adding the following code in bold to images.inc.php:

```
<?php

class ImageHandler
{
    // The folder in which to save images
    public $save_dir;

    // Sets the $save_dir on instantiation
    public function __construct($save_dir)
    {
        $this->save_dir = $save_dir;
    }

    /**
     * Resizes/resamples an image uploaded via a web form
     *
     * @param array $upload the array contained in $_FILES
     * @return string the path to the resized uploaded file
     */
    public function processUploadedImage($file)
    {
        // Separate the uploaded file array
        list($name, $type, $tmp, $err, $size) = array_values($file);

        // Finish processing
    }
}

?>
```

Checking for Errors Using Exceptions

Next, you need to check whether there an error occurred during the upload. When you're dealing with files uploaded through an HTML form, you have access to a special constant called UPLOAD_ERR_OK that tells you whether a file uploaded successfully.

Before you try to process the file, you need to make sure that your $err value is equivalent to UPLOAD_ERR_OK. Add the following code in bold to images.inc.php:

```
<?php

class ImageHandler
{
    // The folder in which to save images
    public $save_dir;

    // Sets the $save_dir on instantiation
    public function __construct($save_dir)
    {
        $this->save_dir = $save_dir;
    }

    /**
     * Resizes/resamples an image uploaded via a web form
     *
     * @param array $upload the array contained in $_FILES
     * @return string the path to the resized uploaded file
     */
    public function processUploadedImage($file)
    {
        // Separate the uploaded file array
        list($name, $type, $tmp, $err, $size) = array_values($file);

        // If an error occurred, throw an exception
        if($err != UPLOAD_ERR_OK) {
            throw new Exception('An error occurred with the upload!');
            return;
        }

        // Finish processing
    }
}

?>
```

Note the use of the `throw new Exception()` line. This is a special form of error handling available in object-oriented scripts. Exceptions give you the ability to catch errors in your scripts without displaying ugly error messages to your end user.

In its simplest form, an exception can return a custom error message to your user in the event of an error. The preceding script takes this approach. Passing a string as an argument to the exception lets you define a custom error message; I'll cover how you handle Exceptions in a moment.

Saving the File

So far you can determine whether your file was uploaded without error; the next step is to save an uploaded file to your file system. Begin by assigning a path and filename to save. For now, you can use the original name of the file, which you can grab from `$save_dir` of the instantiated object. This is the path that displays the image.

However, you still don't have enough information to save your file. Specifically, your site isn't stored at the web root, so you need to get the root path of your site to generate an absolute path, which you can use to save the file. Fortunately, the `$_SERVER` superglobal stores the document root path for you, so all you need to do is include its value to save the image.

You can establish your paths by adding the code in bold to `images.inc.php`:

```
<?php

class ImageHandler
{
    // The folder in which to save images
    public $save_dir;

    // Sets the $save_dir on instantiation
    public function __construct($save_dir)
    {
        $this->save_dir = $save_dir;
    }

    /**
     * Resizes/resamples an image uploaded via a web form
     *
     * @param array $upload the array contained in $_FILES
     * @return string the path to the resized uploaded file
     */
    public function processUploadedImage($file)
    {
        // Separate the uploaded file array
        list($name, $type, $tmp, $err, $size) = array_values($file);

        // If an error occurred, throw an exception
        if($err != UPLOAD_ERR_OK) {
            throw new Exception('An error occurred with the upload!');
            exit;
        }

        // Create the full path to the image for saving
        $filepath = $this->save_dir . $name;
        // Store the absolute path to move the image
        $absolute = $_SERVER['DOCUMENT_ROOT'] . $filepath;

        // Finish processing
    }
}

?>
```

For example, if your `$filepath` is `simple_blog/images/IMG001.jpg`, your `$absolute` value might be `/Applications/XAMPP/xamppfiles/htdocs/simple_blog/images/IMG001.jpg`.

Now that you have your paths on hand, you can save your image to the file system. Do this using the `move_uploaded file()` function, which accepts two arguments: the temporary location of an uploaded file and the location where that file should be saved permanently.

You can save your image by adding the following code in bold to `images.inc.php`:

```
<?php

class ImageHandler
{
    // The folder in which to save images
    public $save_dir;

    // Sets the $save_dir on instantiation
    public function __construct($save_dir)
    {
        $this->save_dir = $save_dir;
    }

    /**
     * Resizes/resamples an image uploaded via a web form
     *
     * @param array $upload the array contained in $_FILES
     * @return string the path to the resized uploaded file
     */
    public function processUploadedImage($file)
    {
        // Separate the uploaded file array
        list($name, $type, $tmp, $err, $size) = array_values($file);

        // If an error occurred, throw an exception
        if($err != UPLOAD_ERR_OK) {
            throw new Exception('An error occurred with the upload!');
            exit;
        }
        // Create the full path to the image for saving
        $filepath = $this->save_dir . '/' . $name;

        // Store the absolute path to move the image
        $absolute = $_SERVER['DOCUMENT_ROOT'] . $filepath;

        // Save the image
        if(!move_uploaded_file($tmp, $absolute))
        {
            throw new Exception("Couldn't save the uploaded file!");
        }

        return $filepath;
    }
}

?>
```

At this point, you're ready to try out your class!

Modifying update.inc.php to Save Images

You've put your class together; next, you need to instantiate it in `update.inc.php` and feed it your uploaded image.

You can do this by opening `update.inc.php` and modifying it so it contains the lines in bold:

```
<?php

// Include the functions so you can create a URL
include_once 'functions.inc.php';

// Include the image handling class
include_once 'images.inc.php';

if($_SERVER['REQUEST_METHOD']=='POST'
    && $_POST['submit']=='Save Entry'
    && !empty($_POST['page'])
    && !empty($_POST['title'])
    && !empty($_POST['entry']))
{
    // Create a URL to save in the database
    $url = makeUrl($_POST['title']);
```

```
if(isset($_FILES['image']['tmp_name']))
{
    try
    {
        // Instantiate the class and set a save path
        $img = new ImageHandler("/simple_blog");

        // Process the file and store the returned path
        $img_path = $img->processUploadedImage($_FILES['image']);

        // Output the uploaded image as it was saved
        echo '<img src="', $img_path, '" /><br />';
    }
    catch(Exception $e)
    {
        // If an error occurred, output your custom error message
        die($e->getMessage());
    }
}
else
{
    // Avoids a notice if no image was uploaded
    $img_path = NULL;
}

// Outputs the saved image path
echo "Image Path: ", $img_path, "<br />";
exit; // Stops execution before saving the entry

// Include database credentials and connect to the database
include_once 'db.inc.php';
$db = new PDO(DB_INFO, DB_USER, DB_PASS);
```

The most important task this code accomplishes is to make your `ImageHandler` class available by including `images.inc.php`. Next, you need to check whether an image was uploaded, which you accomplish by making sure the temporary file exists.

Using try...catch with Exceptions

Your next task introduces a new construct: the `try...catch` statement. You use this construct with exceptions to handle errors gracefully. Essentially it says: “Run this snippet of code. If an exception is thrown, catch it and perform the following snippet.”

In your image handling snippet, you place your object instantiation and processing within a `try` block. Doing so ensures that you can output the custom error message by grabbing the message within the catch block if any of the custom errors you define within the class are thrown.

Inside your `try` block, you instantiate the ImageHandler object and pass "/simple_blog" as its argument, which sets the `$save_dir` property. Next, you call `processUploadedImage()` and pass the uploaded file as its argument. `processUploadedImage()` returns the file's path, so you store that in a variable called `$img_path`, which you use (for now) to output the image for viewing via an `<img>` HTML tag.

Finally, you output the image path as plain text and exit the script to prevent your test entries from being saved to the database.

You can test your class by uploading an image. Navigate to your admin form in a browser and fill out the form with test data (you don't save this), then select a file to upload. When you press the Save Entry button, you should see your image displayed, along with its path (see Figure 8-2).

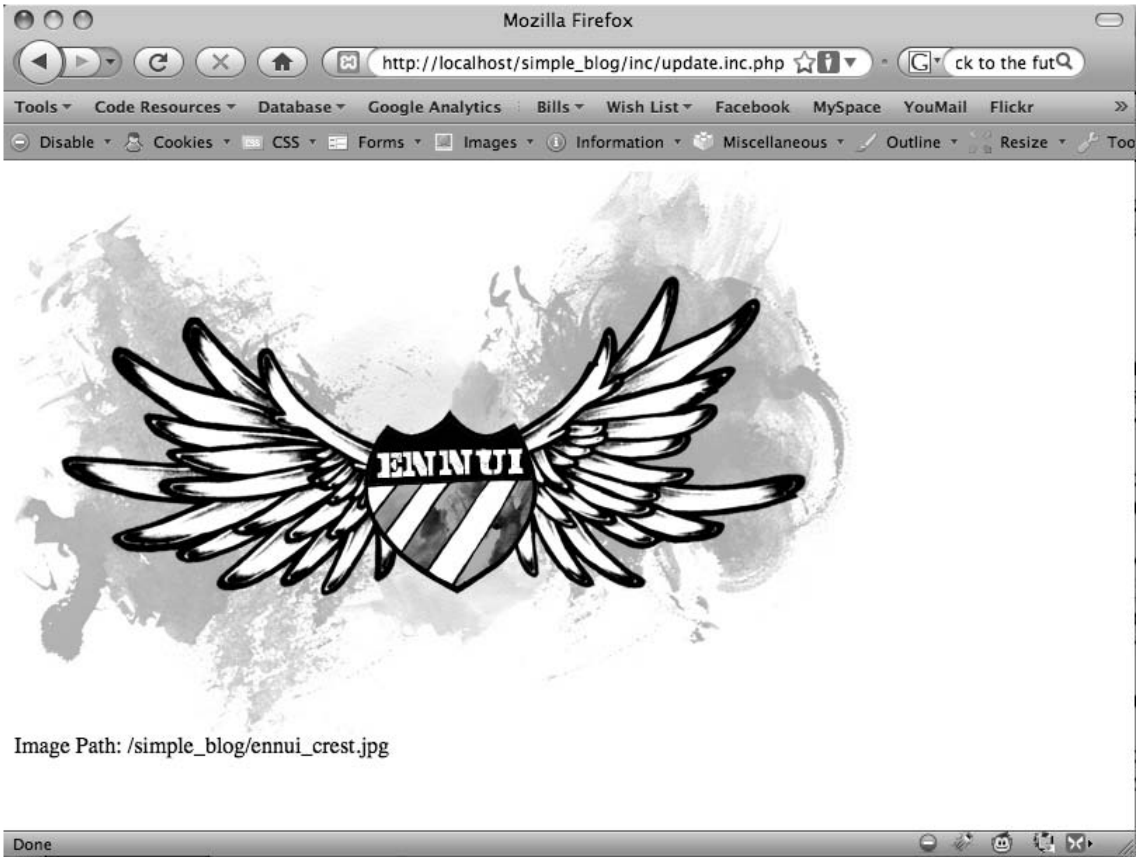


Image Path: /simple_blog/ennui_crest.jpg

Figure 8-2. *An image uploaded by your* ***`ImageHandler`*** *class*

If you look at your `simple_blog` folder in the file system, you'll see that the image has been saved (see Figure 8-3).

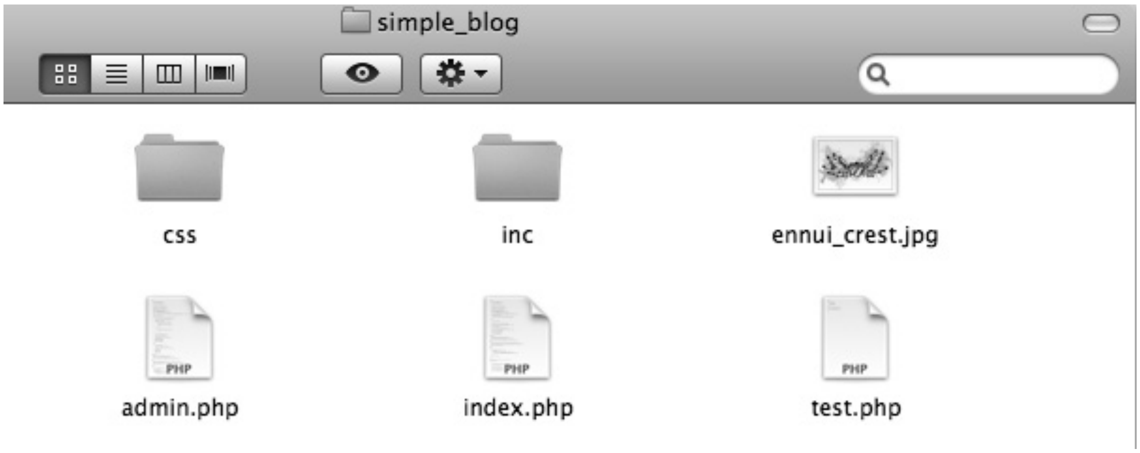


***Figure 8-3.** The uploaded image saved in the `simple_blog` folder*

This isn't necessarily bad, but it does cause clutter in your folder. You can clear this up by creating a new folder to store images in.

Creating a New Folder

You could simply create the folder manually, but it's better to use PHP to check whether a folder exists, then create it if it doesn't. This way, you have to change only the path if the folder you wish to save images in changes in the future; the alternative is to go in and manipulate the file system directly.

You can make a new folder by creating a new method in your `ImageHandler` class called `checkSaveDir()` that creates a directory if it doesn't exist already.

Begin by declaring your method in `ImageHandler`. This method is private, so you want to make sure you control its use. In `images.inc.php`, after `processUploadedImage()`, define your new method by adding the code in bold:

```
<?php

class ImageHandler
{
    public $save_dir;

    public function __construct($save_dir)
    {
        $this->save_dir = $save_dir;
    }
```

```
/**
 * Resizes/resamples an image uploaded via a web form
 *
 * @param array $upload the array contained in $_FILES
 * @return string the path to the resized uploaded file
 */
public function processUploadedImage($file, $rename=TRUE)
{
    // Separate the uploaded file array
    list($name, $type, $tmp, $err, $size) = array_values($file);

    // If an error occurred, throw an exception
    if($err != UPLOAD_ERR_OK) {
        throw new Exception('An error occurred with the upload!');
        exit;
    }

    // Create the full path to the image for saving
    $filepath = $this->save_dir . $name;

    // Store the absolute path to move the image
    $absolute = $_SERVER['DOCUMENT_ROOT'] . $filepath;

    // Save the image
    if(!move_uploaded_file($tmp, $absolute))
    {
        throw new Exception("Couldn't save the uploaded file!");
    }

    return $filepath;
}

/**
 * Ensures that the save directory exists
 *
 * Checks for the existence of the supplied save directory,
 * and creates the directory if it doesn't exist. Creation is
 * recursive.
 *
 * @param void
 * @return void
 */
```

```
    private function checkSaveDir()
    {
        // Check for the dir
    }

}

?>
```

You've declared your method. Next you need to figure out which path to check. As with `processUploadedImage()`, you use the `$_SERVER` superglobal and your `$save_dir` property to create your path to check. The only difference is that you don't attach a file name this time.

Add the lines in bold to `checkSaveDir()` to store your path to check:

```
    /**
     * Ensures that the save directory exists
     *
     * Checks for the existence of the supplied save directory,
     * and creates the directory if it doesn't exist. Creation is
     * recursive.
     *
     * @param void
     * @return void
     */
    private function checkSaveDir()
    {
        // Determines the path to check
        $path = $_SERVER['DOCUMENT_ROOT'] . $this->save_dir;

        // Check for the dir
    }
```

Next, you need to see whether the `$path` you've stored exists. PHP provides a function to do exactly this in `is_dir()`. If the path exists, `is_dir()` returns `TRUE`; otherwise, it returns `FALSE`. You want to continue processing only if the directory doesn't exist, so add a check to see whether `is_dir()` returns `FALSE` before continuing. Insert the lines in bold into `checkSaveDir()`:

```
    /**
     * Ensures that the save directory exists
     *
     * Checks for the existence of the supplied save directory,
     * and creates the directory if it doesn't exist. Creation is
     * recursive.
     *
     * @param void
     * @return void
     */
    private function checkSaveDir()
    {
        // Determines the path to check
        $path = $_SERVER['DOCUMENT_ROOT'] . $this->save_dir;

        // Checks if the directory exists
        if(!is_dir($path))
        {
            // Create the directory
        }
    }
```

If the directory doesn't exist, you need to create it. You accomplished in PHP with the `mkdir()` function, which translates in plain English to *make directory*. You need to pass three arguments to `mkdir()` for it to work properly: the path, the mode, and a value that indicates whether directories should be created recursively.

The first argument, the path, is what you just created and stored in the `$path` variable.

The second argument, the mode, describes how to set the folder permissions. The default mode is `0777`, which provides the widest possible access. Your image files are not sensitive, so you display them to any user viewing your page.

■ **Note** For more information on file permissions, check the PHP manual entry on `chmod()`at `http://php.net/chmod`. Basically, you set folder permissions using an octal number, where each number represents who can access the file (owner, owner's group, and everyone else). Each number represents a level of permission, with `7` being the highest (`read`, `write`, and `execute`).

Your third argument is a boolean value that tells the function whether directories should be created recursively. Only one directory at a time can be created when set to `FALSE` (the default). This means you need to call `mkdir()` twice if you want to add two subdirectories to the `simple_blog` folder with the path, `simple_blog/images/uploads/`. If you set the third argument to `TRUE`, however, you can create both directories with a single function call.

You need to control access to this method, so you allow the function to create directories recursively.

You can create the directory in the simple_blog folder by adding the lines in bold to checkSaveDir():

```
    /**
     * Ensures that the save directory exists
     *
     * Checks for the existence of the supplied save directory,
     * and creates the directory if it doesn't exist. Creation is
     * recursive.
     *
     * @param void
     * @return void
     */
    private function checkSaveDir()
    {
        // Determines the path to check
        $path = $_SERVER['DOCUMENT_ROOT'] . $this->save_dir;

        // Checks if the directory exists
        if(!is_dir($path))
        {
            // Creates the directory
            if(!mkdir($path, 0777, TRUE))
            {
                // On failure, throws an error
                throw new Exception("Can't create the directory!");
            }
        }
    }
```

This code includes a provision to throw an exception if mkdir() returns FALSE, which means it failed. Your method is finally ready. You want to call it from the processUploadedImage() method before you attempt to move the uploaded file. Implement this by adding the lines in bold lines to processUploadedImage():

```
    /**
     * Resizes/resamples an image uploaded via a web form
     *
     * @param array $upload the array contained in $_FILES
     * @return string the path to the resized uploaded file
     */
    public function processUploadedImage($file)
    {
        // Separate the uploaded file array
        list($name, $type, $tmp, $err, $size) = array_values($file);
        // If an error occurred, throw an exception
        if($err != UPLOAD_ERR_OK) {
            throw new Exception('An error occurred with the upload!');
            exit;
        }

        // Check that the directory exists
        $this->checkSaveDir();

        // Create the full path to the image for saving
        $filepath = $this->save_dir . $name;

        // Store the absolute path to move the image
        $absolute = $_SERVER['DOCUMENT_ROOT'] . $filepath;

        // Save the image
        if(!move_uploaded_file($tmp, $absolute))
        {
            throw new Exception("Couldn't save the uploaded file!");
        }

        return $filepath;
    }
```

It's time to test your new function. In `update.inc.php`, modify your object instantiation to use this path as the `$save_dir`: `/simple_blog/images/`.

Your code should look like this:

```
<?php

// Include the functions so you can create a URL
include_once 'functions.inc.php';

// Include the image handling class
include_once 'images.inc.php';

if($_SERVER['REQUEST_METHOD']=='POST'
    && $_POST['submit']=='Save Entry'
    && !empty($_POST['page'])
    && !empty($_POST['title'])
    && !empty($_POST['entry']))
{
    // Create a URL to save in the database
    $url = makeUrl($_POST['title']);
    if(isset($_FILES['image']['tmp_name']))
    {
        try
        {
            // Instantiate the class and set a save path
            $img = new ImageHandler("/simple_blog/images/");

            // Process the file and store the returned path
            $img_path = $img->processUploadedImage($_FILES['image']);

            // Output the uploaded image as it was saved
            echo '<img src="', $img_path, '" /><br />';
        }
        catch(Exception $e)
        {
            // If an error occurred, output your custom error message
            die($e->getMessage());
        }
    }
    else
    {
        // Avoids a notice if no image was uploaded
        $img_path = NULL;
    }

    // Outputs the saved image path
    echo "Image Path: ", $img_path, "<br />";
    exit; // Stops execution before saving the entry

    // Include database credentials and connect to the database
    include_once 'db.inc.php';
    $db = new PDO(DB_INFO, DB_USER, DB_PASS);
```

Save `update.inc.php` and navigate to your admin form in a browser, then fill out the form and submit an image. After you click the Save Entry button, you should see the image you uploaded previously, as well as its path; your script places the image in the newly created images folder (see Figure 8-4).

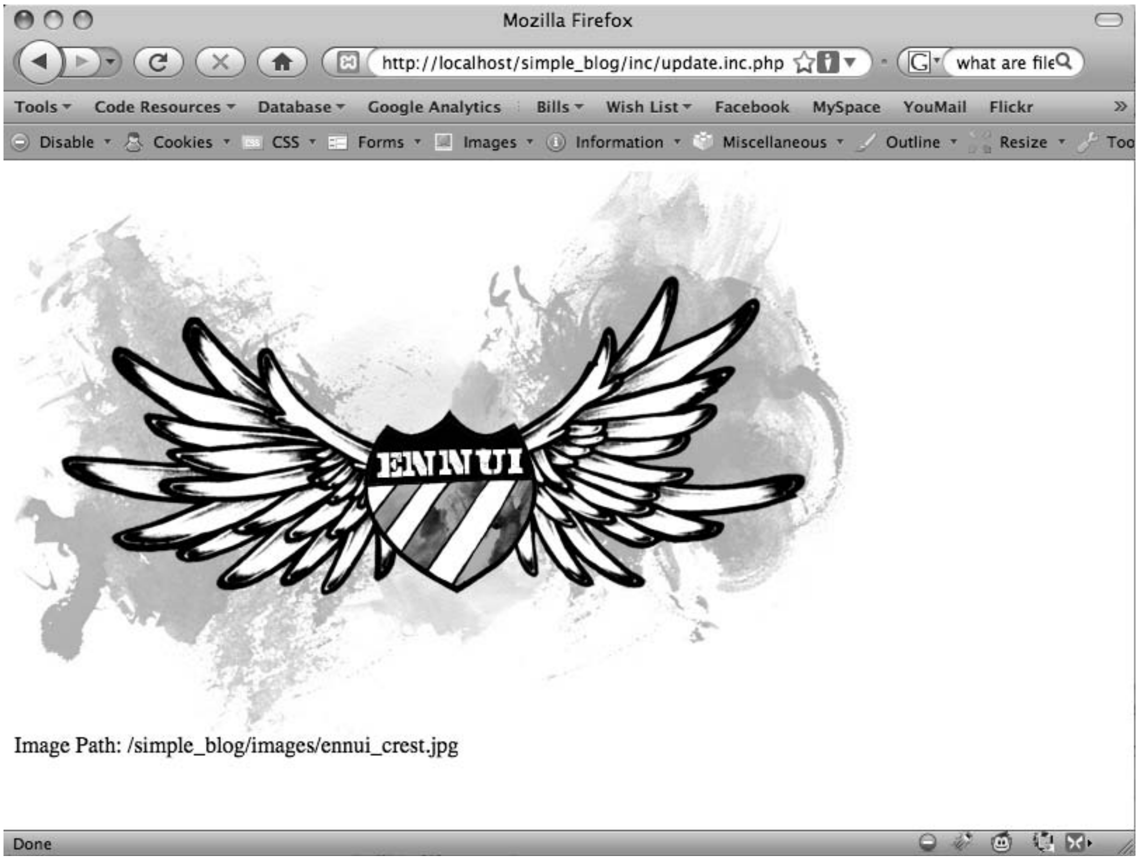
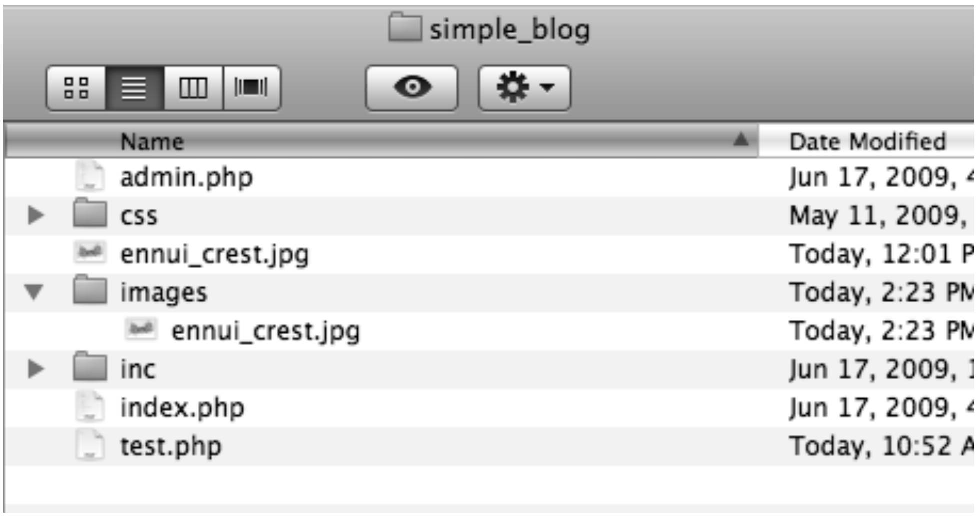


Image Path: /simple_blog/images/ennui_crest.jpg

Figure 8-4. *The image uploaded shows that it's been stored in the new images folder.*

You can check the file system manually to see your new folder and the saved, uploaded image (see Figure 8-5).



```
Name                          Date Modified
admin.php                     Jun 17, 2009,
css                           May 11, 2009,
ennui_crest.jpg               Today, 12:01 P
images                        Today, 2:23 PM
    ennui_crest.jpg           Today, 2:23 PM
inc                           Jun 17, 2009,
index.php                     Jun 17, 2009,
test.php                      Today, 10:52 A
```

Figure 8-5. *The images folder has been created, and the image has been saved in it.*

You're almost ready to start working with the database. First, however, you need to make sure that your images have unique names, so you don't accidentally overwrite older uploads with new ones.

Renaming the Image

You can't trust that every file uploaded to your blog will be uniquely named, so you need to rename any image that is uploaded to your blog. Otherwise, it's possible for a user who uploads an image named to overwrite that image later if he submits a future image with the same name. In this case, the new image would suddenly appear for the older entry, as well as the new one, and you would lose the old image.

You can avoid this by creating a new private method in `ImageHandler` that generates a new, unique name for any uploaded image. You can make sure this name is unique by using the current timestamp and a random four-digit number between 1000 and 9999. This way, even images uploaded during the same second will receive unique names.

■ **Note** This method is not 100% effective, but the likelihood of two images being uploaded at the exact same second and generating the exact same random number is so slim that it will most likely never be a problem.

This method is a one-liner that accepts one argument: the file extension you want to use (we'll get to how you know what file extension to send in just a moment). The method returns the current timestamp, an underscore, a random number, and a file extension.

It's time to add your method to `ImageHandler` by inserting the code in boldafter `processUploadedImage()` in `images.inc.php`:

```
    /**
     * Generates a unique name for a file
     *
     * Uses the current timestamp and a randomly generated number
     * to create a unique name to be used for an uploaded file.
     * This helps prevent a new file upload from overwriting an
     * existing file with the same name.
     *
     * @param string $ext the file extension for the upload
     * @return string the new filename
     */
    private function renameFile($ext)
    {
        /*
         * Returns the current timestamp and a random number
         * to avoid duplicate filenames
         */
        return time() . '_' . mt_rand(1000,9999) . $ext;
    }
```

Determining the File Extension

Before `renameFile()` can work, you need to figure out the uploaded image's extension. Do this by accessing the image's type with the value stored in `$type` in `processUploadedImage()`.

All uploaded files have a content type, which you can use to determine the proper file extension to use. You need to make sure you're processing only images, so you use a switch and match the known content types you want to accept, then set a default action to throw an error if an unexpected content type is passed.

The content types you want to accept are:

- `image/gif`: A GIF image (`.gif`)
- `image/jpeg`: A JPEG image (`.jpg`)
- `image/pjpeg`: A JPEG image as it is recognized by certain browsers, which uses the same file extension as a "normal" JPEG (`.jpg`)
- `image/png`: A PNG image (`.png`)

To check for content types, you create a new private method called `getImageExtension()`. This method contains the switch just discussed and returns the proper file extension. Add the following code in bold to `images.inc.php` just after `renameFile()`:

```
    /**
     * Determines the filetype and extension of an image
     *
     * @param string $type the MIME type of the image
     * @return string the extension to be used with the file
     */
    private function getImageExtension($type)
    {
        switch($type) {
            case 'image/gif':
                return '.gif';

            case 'image/jpeg':
            case 'image/pjpeg':
                return '.jpg';

            case 'image/png':
                return '.png';

            default:
                throw new Exception('File type is not recognized!');
        }
    }
```

Now you can retrieve the file extension to pass to `renameFile()`, which means you're ready to implement the methods in `processUploadedImage()`.

There might be a point at which you no longer wish to rename files, so you should add an argument to `processUploadedImage()` that holds a boolean value. If set to `TRUE` (the default), the image is renamed; if set to `FALSE`, the original filename is used.

You can add file renaming to your method by adding the following code in bold to `processUploadedImage()`:

```
    /**
     * Resizes/resamples an image uploaded via a web form
     *
     * @param array $upload the array contained in $_FILES
     * @param bool $rename whether or not the image should be renamed
     * @return string the path to the resized uploaded file
     */
    public function processUploadedImage($file, $rename=TRUE)
    {
        // Separate the uploaded file array
        list($name, $type, $tmp, $err, $size) = array_values($file);
        // If an error occurred, throw an exception
        if($err != UPLOAD_ERR_OK) {
            throw new Exception('An error occurred with the upload!');
            exit;
        }

        // Check that the directory exists
        $this->checkSaveDir();

        // Rename the file if the flag is set to TRUE
        if($rename===TRUE) {
            // Retrieve information about the image
            $img_ext = $this->getImageExtension($type);

            $name = $this->renameFile($img_ext);
        }

        // Create the full path to the image for saving
        $filepath = $this->save_dir . $name;

        // Store the absolute path to move the image
        $absolute = $_SERVER['DOCUMENT_ROOT'] . $filepath;

        // Save the image
        if(!move_uploaded_file($tmp, $absolute))
        {
            throw new Exception("Couldn't save the uploaded file!");
        }

        return $filepath;
    }
```

Save images.inc.php and try uploading another image through the admin form. You should see a renamed file stored in the file system, as well as on your screen (see Figure 8-6).

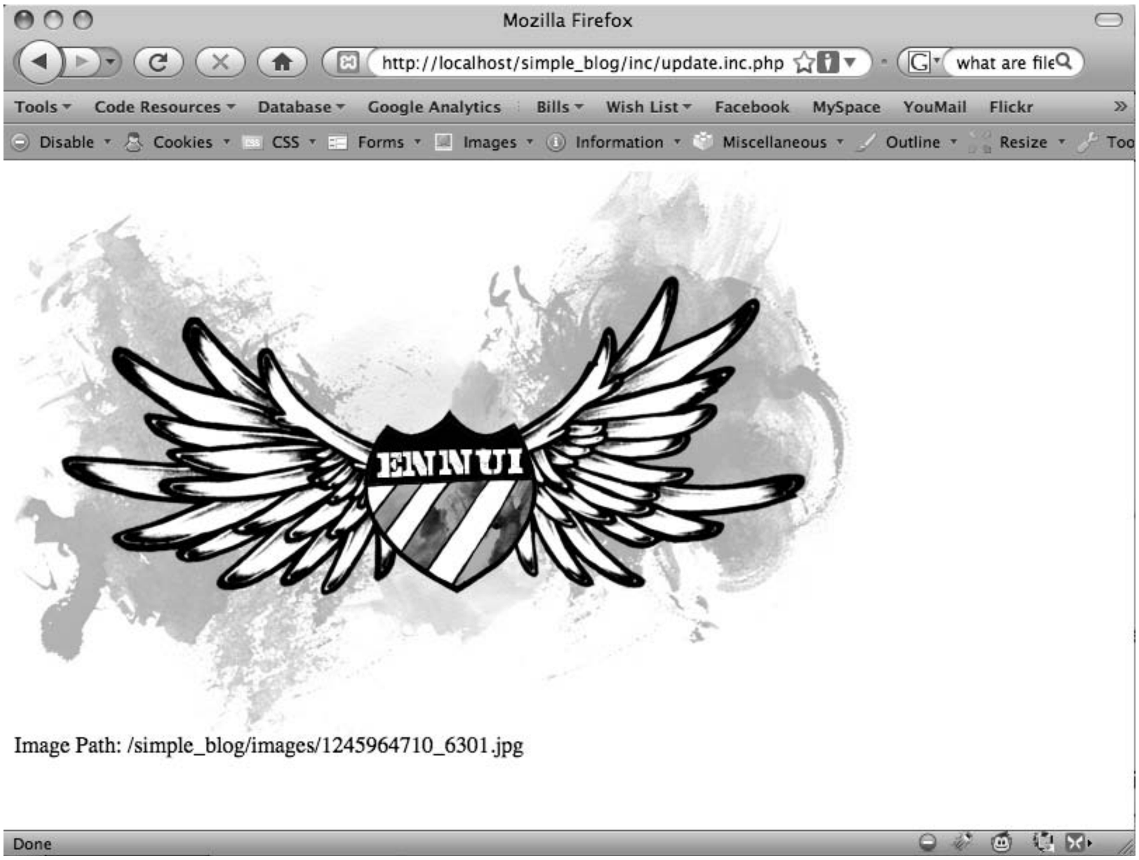


Image Path: /simple_blog/images/1245964710_6301.jpg

Figure 8-6. *An image renamed by your script*

Your ImageHandler class now accepts, renames, and stores images in your file system, which means you're ready to start working with the database.

Storing and Retrieving Images from the Database

If you need to, you can store the image itself in the database as a BLOB column. However, it's much more efficient to save the path to the image in the database instead. This means you need to do three things to save an image:

- Add an image column to the entries table
- Modify update.inc.php to save the image path along with the rest of the entry
- Modify retrieveEntries() to select the new image column

Modifying the entries Table

Your next step is to add the `image` column to your `entries` table. You do this the same way that you added all the other columns to your table.

Navigate to `http://localhost/phpmyadmin`, open the `simple_blog` database, select the `entries` table, and open the SQL tab. Insert the following command to add the `image` column:

```
ALTER TABLE entries
ADD image VARCHAR(150) DEFAULT NULL
AFTER title
```

This creates an `image` column after the `title` column, which stores a 150-character string and defaults to `NULL` if no value is supplied.

Modifying update.inc.php to Save Images

Now that your entries table can store the image path, it's time to modify `update.inc.php` to save the image path.

You've already done everything necessary to make the image path available. All you need to do is remove the sections of code that output the image and exit the script.

After the code no longer outputs image data, you need to modify your queries to include the image path. You do this for both new entries and updated entries.

You can save the image path in the database by modifying `update.inc.php` to reflect the changes shown in bold:

```
<?php

// Include the functions so you can create a URL
include_once 'functions.inc.php';

// Include the image handling class
include_once 'images.inc.php';

if($_SERVER['REQUEST_METHOD']=='POST'
    && $_POST['submit']=='Save Entry'
    && !empty($_POST['page'])
    && !empty($_POST['title'])
    && !empty($_POST['entry']))
```

```
{
    // Create a URL to save in the database
    $url = makeUrl($_POST['title']);

    if(isset($_FILES['image']['tmp_name']))
    {
        try
        {
            // Instantiate the class and set a save dir
            $img = new ImageHandler("/simple_blog/images/");

            // Process the uploaded image and save the returned path
            $img_path = $img->processUploadedImage($_FILES['image']);
        }
        catch(Exception $e)
        {
            // If an error occurred, output your custom error message
            die($e->getMessage());
        }
    }
    else
    {
        // Avoids a notice if no image was uploaded
        $img_path = NULL;
    }

    // Include database credentials and connect to the database
    include_once 'db.inc.php';
    $db = new PDO(DB_INFO, DB_USER, DB_PASS);

    // Edit an existing entry
    if(!empty($_POST['id']))
    {
        $sql = "UPDATE entries
                SET title=?, image=?, entry=?, url=?
                WHERE id=?
                LIMIT 1";
        $stmt = $db->prepare($sql);
        $stmt->execute(
            array(
                $_POST['title'],
                $img_path,
                $_POST['entry'],
                $url,
                $_POST['id']
            )
        );
        $stmt->closeCursor();
    }

    // Create a new entry
    else
    {
        // Save the entry into the database
        $sql = "INSERT INTO entries (page, title, image, entry, url)
                VALUES (?, ?, ?, ?, ?)";
        $stmt = $db->prepare($sql);
        $stmt->execute(
            array(
                $_POST['page'],
                $_POST['title'],
                $img_path,
                $_POST['entry'],
                $url
            )
        );
        $stmt->closeCursor();
    }

    // Sanitize the page information for use in the success URL
    $page = htmlentities(strip_tags($_POST['page']));

    // Send the user to the new entry
    header('Location: /simple_blog/'.$page.'/'.$url);
    exit;
}
```

```
else
{
    header('Location: ../');
    exit;
}

?>
```

At this point, you're ready to create a new entry with an image. Navigate to your admin form in a browser and create an entry with the following information:

- *Title*: Entry with an image
- *Body*: This entry is created with an accompanying image

Add an image, then click Save Entry. If you look in the database, you should see that an image path has been saved in the `image` column.

Now you need to retrieve the path from the database to display uploaded images with the entry.

Modifying retrieveEntries() to Retrieve Images

Your first step in displaying saved images is to add the `image` column to the array returned from `retrieveEntries()`. This is easy to do; it requires only that you add the column name to the SQL query.

Modify `retrieveEntries()` in `functions.inc.php` to reflect the changes shown in bold:

```
function retrieveEntries($db, $page, $url=NULL)
{
    /*
     * If an entry URL was supplied, load the associated entry
     */
    if(isset($url))
    {
        $sql = "SELECT id, page, title, image, entry
                FROM entries
                WHERE url=?
                LIMIT 1";
        $stmt = $db->prepare($sql);
        $stmt->execute(array($url));

        // Save the returned entry array
        $e = $stmt->fetch();

        // Set the fulldisp flag for a single entry
        $fulldisp = 1;
    }
```

```
/*
 * If no entry ID was supplied, load all entry titles for the page
 */
else
{
    $sql = "SELECT id, page, title, image, entry, url
            FROM entries
            WHERE page=?
            ORDER BY created DESC";
    $stmt = $db->prepare($sql);
    $stmt->execute(array($page));

    $e = NULL; // Declare the variable to avoid errors

    // Loop through returned results and store as an array
    while($row = $stmt->fetch()) {
        if($page=='blog')
        {
            $e[] = $row;
            $fulldisp = 0;
        }
        else
        {
            $e = $row;
            $fulldisp = 1;
        }
    }

    /*
     * If no entries were returned, display a default
     * message and set the fulldisp flag to display a
     * single entry
     */
    if(!is_array($e))
    {
        $fulldisp = 1;
        $e = array(
            'title' => 'No Entries Yet',
            'entry' => 'This page does not have an entry yet!'
        );
    }
}
```

```
    // Add the $fulldisp flag to the end of the array
    array_push($e, $fulldisp);

    return $e;
}
```

Now you can access the value stored in the image column, just as you access all the other information in an entry. Next, you need to use this information to display the image for the world to see.

Modifying index.php to Display Images

Unlike text entries, images require a little bit of special treatment when you retrieve them from the database. You can't simply check whether the value is set, then output it; instead, you need to create some extra HTML markup to display images properly.

For the sake of keeping your code clean, you need to write a function that checks whether an image exists and return the appropriate HTML markup if it does.

Adding a Function to Format Images for Output

You add this function to functions.inc.php. The function accepts two arguments: the path to the image and the title of the entry (you use this as an alt attribute).

Its functionality is simple: if an image path is supplied, return valid HTML markup; if not, return NULL.

Open functions.inc.php and add the new function after the existing functions:

```
function formatImage($img=NULL, $alt=NULL)
    {
    if(isset($img))
    {
        return '<img src="'.$img.'" alt="'.$alt.'" />';
    }
    else
       {
        return NULL;
    }
}
```

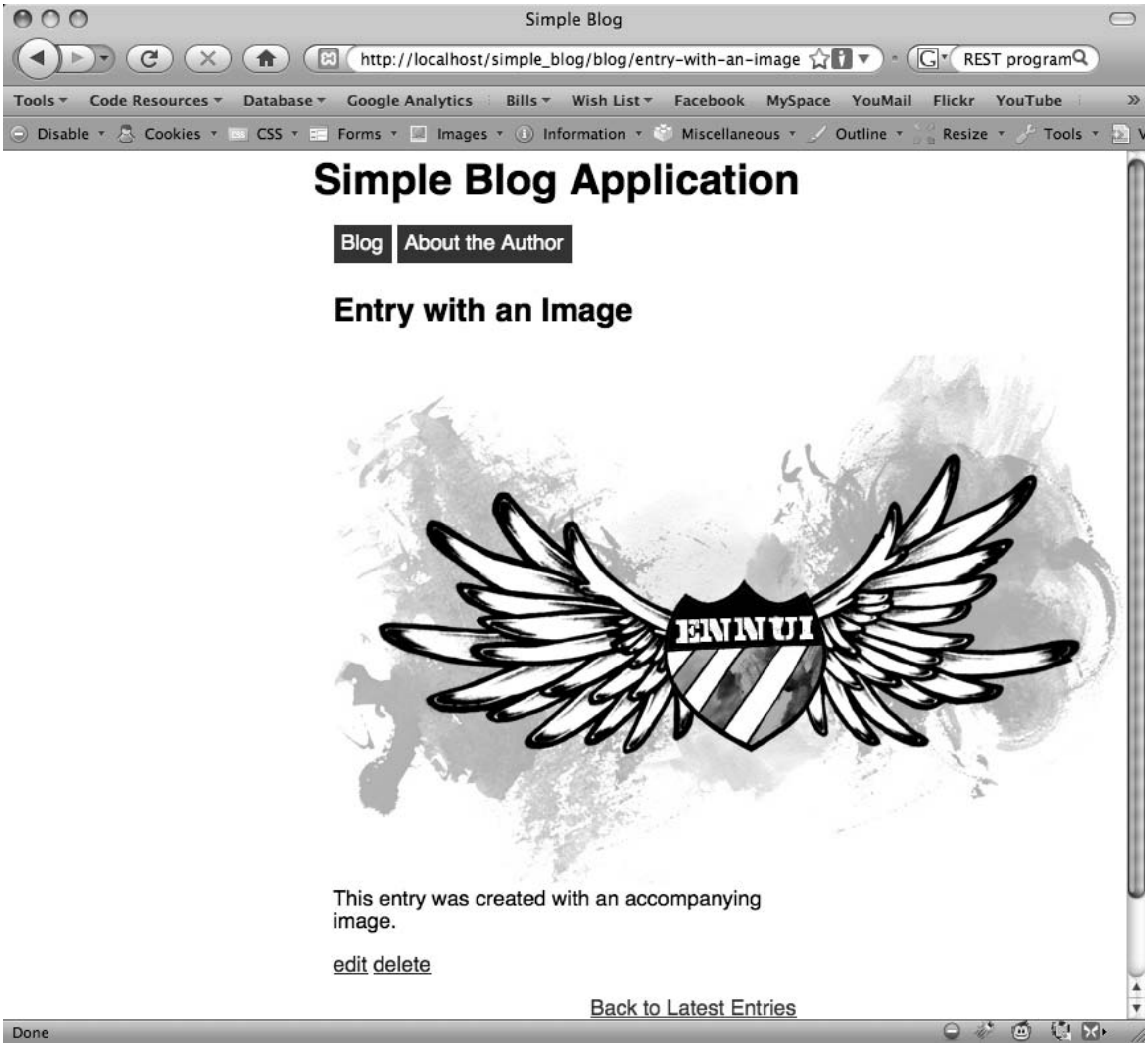
With your new function in place, you can add your image to the displayed entry. For the sake of organization, you display images only with a fully displayed entry.

Your image handling is taken care of in an external function, so you can add your image to the display with just two lines of code. In `index.php`, where the block of code that handles full display is located, modify the script to contain the lines in bold:

```
<?php

// If the full display flag is set, show the entry
if($fulldisp==1)
{

    // Get the URL if one wasn't passed
    $url = (isset($url)) ? $url : $e['url'];

    // Build the admin links
    $admin = adminLinks($page, $url);

    // Format the image if one exists
    $img = formatImage($e['image'], $e['title']);

?>

        <h2> <?php echo $e['title'] ?> </h2>
        <p> <?php echo $img, $e['entry'] ?> </p>
        <p>
            <?php echo $admin['edit'] ?>
            <?php if($page=='blog') echo $admin['delete'] ?>
        </p>
        <?php if($page=='blog'): ?>
        <p class="backlink">
            <a href="./">Back to Latest Entries</a>
        </p>
        <?php endif; ?>
```

If you navigate to `http://localhost/simple_blog/` and select your new entry, "Entry with an Image"; you should see the image displayed with your text (see Figure 8-7).

Unfortunately, the layout doesn't look very good if a large image is displayed (as shown in Figure 8-7). You don't want to resize every image uploaded manually, so you need to add new functionality to your `ImageHandler` class that resizes large images automatically.


Simple Blog
http://localhost/simple_blog/blog/entry-with-an-image
REST program
Tools Code Resources Database Google Analytics Bills Wish List Facebook MySpace YouMail Flickr YouTube
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tools
Done

Simple Blog Application

Blog About the Author

Entry with an Image

This entry was created with an accompanying image.

edit delete

Back to Latest Entries

Figure 8-7. *Your image displayed with the accompanying text*

Resizing Images

Resizing images is a little tricky, but you can break the process down into small sections. Resizing an image requires several steps:

- Determine new dimensions based on the defined maximum dimensions
- Determine which image handling functions are necessary for resampling
- Resample the image at the proper size

Determining the New Image Dimensions

Begin the process of resizing an image by determining what size the image should actually be. Unfortunately, you can't simply tell your script that all images should be 350 pixels by 240 pixels because forcing an image to match dimensions that aren't proportional to an image's original dimensions causes distortion, which doesn't look good.

Instead, you need to determine the proportional size of the image that will fit *within* the maximum dimensions you set. You can accomplish this following several math steps:

- Determine which side of the image is the longer
- Divide the maximum dimension that corresponds to the long side by the long side's size
- Use the resulting decimal point to multiply both sides of the image
- Round the product to keep an integer value
- Use the product of this multiplication as the new width and height of the image

Adding a Property for Maximum Dimensions

Before you can start performing any resizing, you need to define maximum dimensions for uploaded images. This might need to change at some future point, so you define a new property in the `ImageHandler` class, called `$max_dims`. This property stores an array of the maximum width and height allowed for uploaded images.

You set this property in the constructor, but be aware that you need to use a default value so that it doesn't *need* to be set for the class to work.

At the top of the `ImageHandler` class, modify the property declarations and constructor as shown by the code in bold:

```
class ImageHandler
{
    public $save_dir;
    public $max_dims;

    public function __construct($save_dir, $max_dims=array(350, 240))
    {
        $this->save_dir = $save_dir;
        $this->max_dims = $max_dims;
    }
```

The maximum dimensions of 350 pixels by 240 pixels are acceptable for the layout of your current site. These are the default values, so don't worry about adding maximum dimensions to the class instantiation. However, if you needed to change the size of your images, you can change the dimensions using the following instantiation of `ImageHandler`:

```
$obj = new ImageHandler('/images/', array(400, 300));
```

This snippet sets the maximum dimensions allowed to 400 pixels wide by 300 pixels high.

Creating the Method to Determine New Width and Height

You have your maximum dimensions prepared; next, you need to define a new private method in the ImageHandler class. Place this method at the bottom of the class and accept one argument: the image for which you need new dimensions.

In images.inc.php, add the following method declaration:

```
/**
 * Determines new dimensions for an image
 *
 * @param string $img the path to the upload
 * @return array the new and original image dimensions
 */
private function getNewDims($img)
{
    // Get new image dimensions
}
```

You can determine the original dimensions of your image with a function called getimagesize().This function returns the width and height of the image supplied as an argument (as well as other information I'll address in a moment).

You use the list() function to define the first two array elements as the $src_w (source width) and $src_h (source height) variables.

Next, you use the maximum dimensions that you just added to the class. Again, use the list() function to separate their values into $max_w (maximum width) and $max_h (maximum height), then add the following code in bold to getNewDims():

```
/**
 * Determines new dimensions for an image
 *
 * @param string $img the path to the upload
 * @return array the new and original image dimensions
 */
private function getNewDims($img)
{
    // Assemble the necessary variables for processing
    list($src_w, $src_h) = getimagesize($img);
    list($max_w, $max_h) = $this->max_dims;

    // Finish processing
}
```

Before you start resizing your image, you need to check that it is, in fact, larger than your maximum dimensions. Otherwise, small images will be blown up, which can make them look bad. You accomplish this by checking whether either the original image's width or height is greater than the maximum corresponding dimension.

If so, you need to determine the scale to which you should resize the image. You can determine this by dividing the maximum length of both sides by the original length and returning the smaller of the two values using the `min()` function, which compares two or more expressions and returns the lowest value.

After you know what side is longer, you use the corresponding maximum dimension to determine the scale. You determine the scale by dividing the smaller maximum dimension by the original dimension, which gives you a decimal value (for instance, if your maximum size is 60 pixels, and the original size is 100 pixels, the scale is .6).

In the event that your image is smaller than the maximum allowed dimensions, you want to keep it the same size. Do this by setting the scale to 1.

Add the code in bold to `getNewDims()` to determine the scale:

```
/**
 * Determines new dimensions for an image
 *
 * @param string $img the path to the upload
 * @return array the new and original image dimensions
 */
private function getNewDims($img)
{
    // Assemble the necessary variables for processing
    list($src_w, $src_h) = getimagesize($img);
    list($max_w, $max_h) = $this->max_dims;

    // Check that the image is bigger than the maximum dimensions
    if($src_w > $max_w || $src_h > $max_h)
    {
        // Determine the scale to which the image will be resized
        $s = min($max_w/$src_w,$max_h/$src_h);
    }
    else
    {
        /*
         * If the image is smaller than the max dimensions, keep
         * its dimensions by multiplying by 1
         */
        $s = 1;
    }

    // Finish processing
}
```

Finally, you need to multiply the original dimensions by the scale you've just determined, then return the new dimensions (as well as the old dimensions for reasons I'll cover momentarily) as an array.

To accomplish this, insert the lines bold into `getNewDims()`:

```
/**
 * Determines new dimensions for an image
 *
 * @param string $img the path to the upload
 * @return array the new and original image dimensions
 */
private function getNewDims($img)
{
    // Assemble the necessary variables for processing
    list($src_w, $src_h) = getimagesize($img);
    list($max_w, $max_h) = $this->max_dims;

    // Check that the image is bigger than the maximum dimensions
    if($src_w > $max_w || $src_h > $src_h)
    {
        // Determine the scale to which the image will be resized
        $s = min($max_w/$src_w,$max_h/$src_h);
    }
    else
    {
        /*
         * If the image is smaller than the max dimensions, keep
         * its dimensions by multiplying by 1
         */
        $s = 1;
    }

    // Get the new dimensions
    $new_w = round($src_w * $s);
    $new_h = round($src_h * $s);

    // Return the new dimensions
    return array($new_w, $new_h, $src_w, $src_h);
}
```

Determining Which Image Functions to Use

You use two functions when resampling your images: one to create an image resource and one to save your resampled image in a format of your choosing.

Your blog accepts three types of image files (JPEG, GIF, and PNG),, and each of these image types requires an individual set of functions to create and save images. Specifically, you use `imagecreatefromjpeg()`, `imagecreatefromgif()`, or `imagecreatefrompng()` to create the images, depending on the named file type.

Similarly, you save the images using imagejpeg(), imagegif(), or imagepng()—again, depending on the appropriate file type.

You want to eliminate redundant code, so you need to write a method that checks what type of image you're using and returns the names of the functions you should use with the uploaded image.

Call this private method getImageFunctions(); you will use it to return an array containing the names of the functions for creating and saving your images, which you determine with a switch statement.

Your original $type variable is out of the scope for this method, so use the getimagesize() function again. However, this time you need to access the array element that holds the image's MIME type, which you access using the mime array key.

In images.inc.php, declare the getImageFunctions() method and store the output of getimagesize(). Next, pass the MIME type to a switch statement that returns the proper array of function names. You accomplish all of this by adding the following method at the bottom of the ImageHandler class:

```
/**
 * Determines how to process images
 *
 * Uses the MIME type of the provided image to determine
 * what image handling functions should be used. This
 * increases the perfomance of the script versus using
 * imagecreatefromstring().
 *
 * @param string $img the path to the upload
 * @return array the image type-specific functions
 */
private function getImageFunctions($img)
{
    $info = getimagesize($img);

    switch($info['mime'])
    {
        case 'image/jpeg':
        case 'image/pjpeg':
            return array('imagecreatefromjpeg', 'imagejpeg');
            break;
        case 'image/gif':
            return array('imagecreatefromgif', 'imagegif');
            break;
        case 'image/png':
            return array('imagecreatefrompng', 'imagepng');
            break;
        default:
            return FALSE;
            break;
    }
}
```

Now you're able to get functions specific to the image type quickly; I'll cover how you use these in a moment.

Resampling the Image at the Proper Size

Finally, you're ready to resample the image. You do this inside a new private method called `doImageResize()`, which accepts one argument: the image to be resampled.

This method performs its magic in five steps:

- It determines the new dimensions for the image
- It determines the functions needed to resample the image
- It creates image resources to use in the resampling
- It resamples the image at the proper size
- It saves the resampled image

The first two steps are already done; you simply need to call the `getNewDims()` and `getImageFunctions()` methods you defined previously.

Begin by defining your method in `ImageHandler` and calling your methods. Add the following to `images.inc.php` in the `ImageHandler` class:

```
    /**
     * Generates a resampled and resized image
     *
     * Creates and saves a new image based on the new dimensions
     * and image type-specific functions determined by other
     * class methods.
     *
     * @param array $img the path to the upload
     * @return void
     */
    private function doImageResize($img)
    {
        // Determine the new dimensions
        $d = $this->getNewDims($img);

        // Determine what functions to use
        $funcs = $this->getImageFunctions($img);

        // Finish resampling
    }
```

Next you need to create the image resources that PHP uses to deal with images. For your resampling, you need to create two resources. The first is the original image, which you save as a resource using the first of your two functions specific to the image type. The second is a new, blank image resource, which you copy the resampled image into. You create the second image resource with a different function called `imagecreatetruecolor()`; this function accepts two arguments: the width and height of the new image resource you want to create.

To create your image resources, add the lines in bold to `doImageResize()`:

```
/**
 * Generates a resampled and resized image
 *
 * Creates and saves a new image based on the new dimensions
 * and image type-specific functions determined by other
 * class methods.
 *
 * @param array $img the path to the upload
 * @return void
 */
private function doImageResize($img)
{
    // Determine the new dimensions
    $d = $this->getNewDims($tmp);

    // Determine what functions to use
    $funcs = $this->getImageFunctions($img);

    // Create the image resources for resampling
    $src_img = $funcs[0]($img);
    $new_img = imagecreatetruecolor($d[0], $d[1]);

    // Finish resampling
}
```

Note that you're calling your image type-specific function with the snippet, `$funcs[0]($img)`. This is a trick available to developers for instances just like this one, where the function you want to call varies based on the current data being handled.

So far you have your image resources; next, you can copy the original image into the new image resource using the complicated-looking `imagecopyresampled()` function, which accepts a whopping *ten* arguments:

- `$dst_image`: This is the destination image that serves as your new image resource (`$new_img`).
- `$src_image`: This is the source image that you copy your new image from (`$src_img`).
- `$dst_x`: This is the offset from the new image's left-hand side; you use this to start inserting the source image. This value is usually `0`, but if you overlay something like a watermark, you might need to insert the source image in the bottom-right corner, which would require a different value.
- `$dst_y`: The is the offset from the new image's top; you use this to start inserting the source image.

- `$src_x`: This is the offset from the source image's left-hand side; you use this to start copying image data. This value is usually `0`, but this value can vary if you're cropping an image.
- `$src_y`: This is the offset from the top of the source image; you use this to start copying the source image.
- `$dst_w` : This argument specifies thewidth at which you should insert the copied image. You use the new width value stored at `$d[0]`.
- `$dst_h`: This argument specifies the height at which you insert the copied image. You use the new height value stored at `$d[1]`.
- `$src_w`: This argument describes the distance from the starting point to copy horizontally. Usually, this value matches the original size of the image, but this value can shorter than the original width if your image is cropped. You use the original width value stored at `$d[2]`.
- `$src_h`: This argument describes the distance from the starting point to copy vertically. You use the original height value stored at `$d[3]`.

You need to check whether the `imagecopyresampled()` call is successful, then destroy the source image (`$src_img`) to free system resources because you won't need it again. If the function fails for some reason, you throw a new exception to create a custom error message. To do this, add the following code in bold to `doImageResize()`:

```
/**
 * Generates a resampled and resized image
 *
 * Creates and saves a new image based on the new dimensions
 * and image type-specific functions determined by other
 * class methods.
 *
 * @param array $img the path to the upload
 * @return void
 */
private function doImageResize($img)
{

    // Determine the new dimensions
    $d = $this->getNewDims($tmp);

    // Determine what functions to use
    $funcs = $this->getImageFunctions($img);

    // Create the image resources for resampling
    $src_img = $funcs[0]($img);
    $new_img = imagecreatetruecolor($d[0], $d[1]);
```

```
        if(imagecopyresampled(
            $new_img, $src_img, 0, 0, 0, 0, $d[0], $d[1], $d[2], $d[3]
        ))
        {
            imagedestroy($src_img);

            // Finish resampling
        }
        else
        {
            throw new Exception('Could not resample the image!');
        }
    }
```

Now $new_img contains the resized and resampled image. All that's left to do at this point is to save the image. You can accomplish this by using the second image type-specific function that saves the image resource to a location of your choosing. You can save it over the top of the original image because you no longer need it at its original size in your blog application.

You need to make sure that the image type-specific function fires successfully, then destroy the new image resource ($new_img) to free the memory it consumes. If the function fails, you throw an error. Add the following code in bold to doImageResize() to complete your function:

```
    /**
     * Generates a resampled and resized image
     *
     * Creates and saves a new image based on the new dimensions
     * and image type-specific functions determined by other
     * class methods.
     *
     * @param array $img the path to the upload
     * @return void
     */
    private function doImageResize($img)
    {

        // Determine the new dimensions
        $d = $this->getNewDims($tmp);

        // Determine what functions to use
        $funcs = $this->getImageFunctions($img);

        // Create the image resources for resampling
        $src_img = $funcs[0]($img);
        $new_img = imagecreatetruecolor($d[0], $d[1]);
        if(imagecopyresampled(
            $new_img, $src_img, 0, 0, 0, 0, $d[0], $d[1], $d[2], $d[3]
        ))
        {
            imagedestroy($src_img);
            if($new_img && $funcs[1]($new_img, $img))
            {
                imagedestroy($new_img);
            }
            else
            {
                throw new Exception('Failed to save the new image!');
            }
        }
        else
        {
            throw new Exception('Could not resample the image!');
        }
    }
```

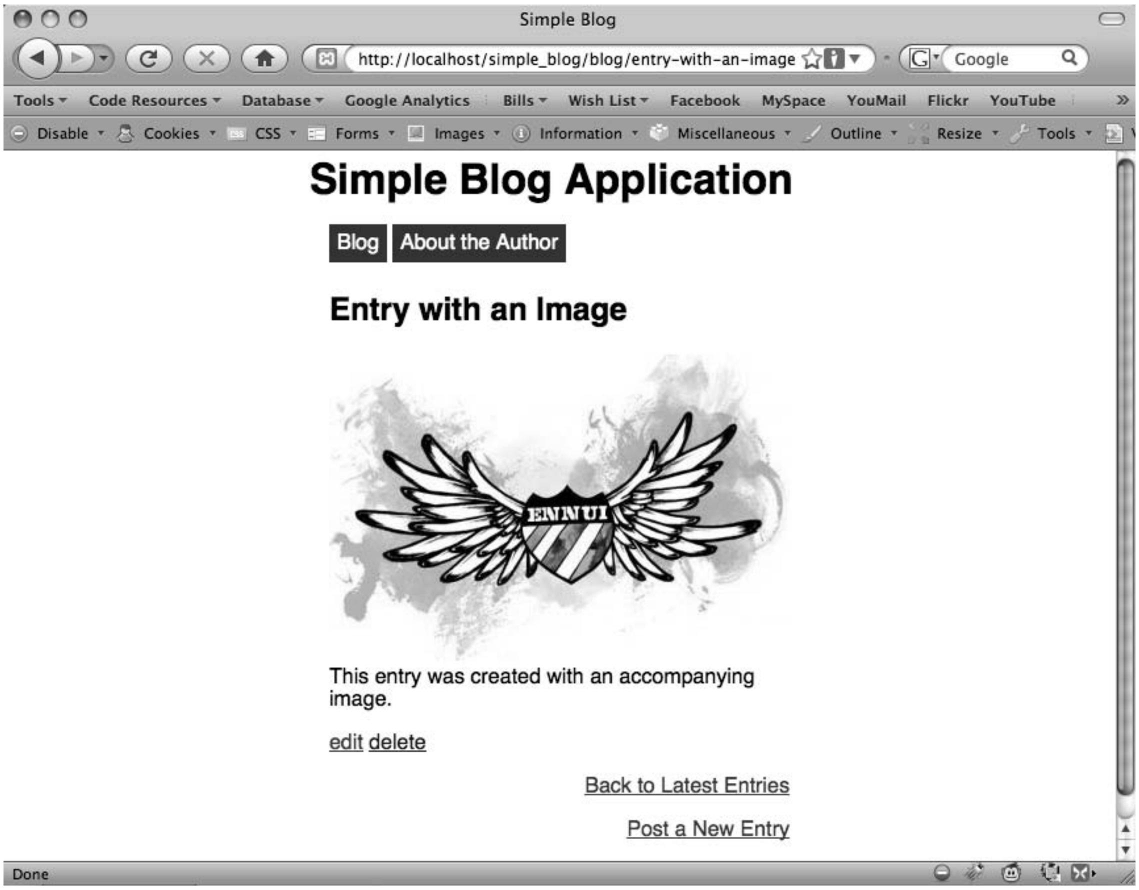
You can now resize any JPEG, GIF, or PNG image to fit within your maximum dimensions. Next, add a call to your new method to processUploadedImage() and try it out!

Adding Your New Method to processUploadedImage()

You need to add only one line of code to the function to resample images processed with processUploadedImage(). In images.inc.php, add the lines in bold to processUploadedImage():

```
    /**
     * Resizes/resamples an image uploaded via a web form
     *
     * @param array $upload the array contained in $_FILES
     * @param bool $rename whether or not the image should be renamed
     * @return string the path to the resized uploaded file
     */
    public function processUploadedImage($file, $rename=TRUE)
    {
        // Separate the uploaded file array
        list($name, $type, $tmp, $err, $size) = array_values($file);

        // If an error occurred, throw an exception
        if($err != UPLOAD_ERR_OK) {
            throw new Exception('An error occurred with the upload!');
            exit;
        }
        // Generate a resized image
        $this->doImageResize($tmp);

        // Rename the file if the flag is set to TRUE
        if($rename===TRUE) {
            // Retrieve information about the image
            $img_ext = $this->getImageExtension($type);

            $name = $this->renameFile($img_ext);
        }

        // Check that the directory exists
        $this->checkSaveDir();

        // Create the full path to the image for saving
        $filepath = $this->save_dir . $name;

        // Store the absolute path to move the image
        $absolute = $_SERVER['DOCUMENT_ROOT'] . $filepath;

        // Save the image
        if(!move_uploaded_file($tmp, $absolute))
        {
            throw new Exception("Couldn't save the uploaded file!");
        }

        return $filepath;
    }
```

You can test this by re-uploading your original image to the "Entry with an Image" entry. Navigate to the entry in a browser at `http://localhost/simple_blog/entry-with-an-image` and click the `edit` link to bring up the admin form. Select the image you uploaded previously and click the Save Entry button. You should now see that the resampled, properly-sized image (see Figure 8-8).


Simple Blog
http://localhost/simple_blog/blog/entry-with-an-image
Google
Tools Code Resources Database Google Analytics Bills Wish List Facebook MySpace YouMail Flickr YouTube
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tools
Done

Simple Blog Application

Blog About the Author

Entry with an Image

This entry was created with an accompanying image.

edit delete

Back to Latest Entries

Post a New Entry

Figure 8-8. *A resized image that is displayed with an entry*

Summary

This chapter has been intense. You've covered several advanced topics, including object-oriented programming and image handling. The next chapter is a little less complex, but the still important: you'll learn how to create an RSS feed for your blog, which is a great feature to add to your blog's functionality.