CHAPTER 11

A Dynamic Image Gallery

So far, the web application we have developed restricts users to only publishing text-based information in their blogs. While we have allowed users a degree of control by permitting a limited subset of HTML to be used (including the use of the `<img>` tag), users are still unable to upload their own images. In this chapter, we will extend the functionality of our blogging system to allow users to upload one or more photos to each of their blog posts.

While this may sound like a fairly trivial process, there are a number of different issues to consider, such as these:

- **Storage of images.** We must store the images on the server and link them to blog posts.
- **Sending images to browsers.** When a user views posts with images in them, we must send the images. This includes dealing with correct MIME headers as well as caching images in the user's browser.
- **Dynamic image sizing.** Since users will upload different sizes and types of images, we must manipulate the images for a consistent layout.

We will simplify the process of image publishing by predetermining the layout of images within a blog post, although users will also have the ability to link to their images via the WYSIWYG editor we implemented in Chapter 7.

One extra feature we will add will allow users to change the order in which their photos appear on a page. We will use Scriptaculous to provide a simple interface for reordering images, and we will use Ajax to save the order of the images. This will be similar to the example in Chapter 5.

The steps we will cover in this chapter are as follows:

1. Adding an image-upload form to the blog post preview page.
2. Adding a new controller action to output uploaded images.
3. Displaying images on blog posts.
4. Displaying a thumbnail on the blog index for each post with images.
5. Allowing users to reorder and delete images from each blog post.

■ **Note** While this chapter deals specifically with images, many of the principles we will look at also apply to general file uploads (after all, an image is a file). The only things that don't apply to non-image files are resizing the images and displaying them in HTML using the `<img>` tag.

Storing Uploaded Files

The first thing we must decide is how we will store files uploaded by users: in the database or on the filesystem. Each method has its own advantages and disadvantages.

Here are some of the reasons you might prefer to store files in the database:

- Doing so provides easy access to all the image information. When using the filesystem, some of the data will still be stored in the database, meaning there is a slight redundancy. Additionally, deleting the image from the database is simply a matter of removing the database record, while on the filesystem you must also delete the image file. It is easier to roll back a failed transaction if you are only using a database.
- Keeping backups of your web application is simpler, since you only need to back up the database and no separate uploaded files.

Now let's take a look at why you may prefer to store images using the filesystem:

- Cross-platform compatibility is easier to achieve. Since most database servers will use different methods for storing binary data, a separate implementation may be required for each type of database server your application is used on.
- It is much simpler to perform filesystem operations on files that are already on the filesystem. For example, if you were to use ImageMagick (a suite of image manipulation tools) to create thumbnails of images, you would find it much simpler to work with files already stored on the filesystem.

■ **Note** We will be using the GD image functions that are built with PHP instead of ImageMagick—I simply used this as an example of filesystem operations that may take place on uploaded files.

There are some other considerations we must take into account. For example, we need to store metadata for each of the images. As mentioned earlier, we want to allow users to change the order of images belonging to each blog post (since a blog post may have several images). As such, not only do we need to track which images belong to which blog posts, but we must also track the order of the images.

My preferred method is to store all uploaded images on the filesystem, and to also use a database table to store information about the images and to link each image to its blog post.

Note If you prefer to store your images in the database, you shouldn't have too much trouble extending the SQL we will create here to do so. However, to produce and save the thumbnail images as the way I describe in this chapter, it is likely that you will still need to store some files on the filesystem.

Creating the Database Table for Image Data

We will first create a table in the database to store information about each uploaded image. This table will hold the filename of the original image as well as a foreign key that links this table to the `blog_posts` table. The final `ranking` column will be used to record the order of the images in a blog post.

Note The name of the `ranking` column isn't too important; however, `order` is a reserved word in SQL, so we cannot use it.

The schema for this table, which we call `blog_posts_images`, is shown in Listing 11-1. This SQL code can be found in the `schema-mysql.sql` file, and the corresponding PostgreSQL code can be found in `schema-pgsql.sql`.

Listing 11-1. *Creating the Database Table Used to Store Image Information (schema-mysql.sql)*

```
create table blog_posts_images (
    image_id        serial          not null,

    filename        varchar(255)    not null,

    post_id         bigint unsigned not null,
    ranking         int unsigned    not null,

    primary key (image_id),
    foreign key (post_id) references blog_posts (post_id)
) type = InnoDB;
```

Controlling Uploaded Images with DatabaseObject

Next, we will create a child class of `DatabaseObject` that we will use to manage both database records for uploaded files and the stored files on the filesystem. As noted previously, we will use a database record to store image data and store the file on the filesystem, as this allows us to easily link the image to the correct blog post. It also allows us to store other data with each image if required (such as an original filename or a caption).

This child class, called `DatabaseObject_BlogPostImage`, will write the file to the filesystem upon successful upload, and it will delete the file from the filesystem and the database record from the table if the user chooses to delete the image.

For now, we will just create the basic skeleton of the `DatabaseObject_BlogPostImage` class, as shown in Listing 11-2; we will add more advanced functionality to this class as we continue on in this chapter. This code should be stored in `BlogPostImage.php`, which resides in the `./include/DatabaseObject` class.

Listing 11-2. *Beginning the Blog Post Image-Management Class (BlogPostImage.php)*

```php
<?php
    class DatabaseObject_BlogPostImage extends DatabaseObject
    {
        public function __construct($db)
        {
            parent::__construct($db, 'blog_posts_images', 'image_id');

            $this->add('filename');
            $this->add('post_id');
            $this->add('ranking');
        }
    }
?>
```

At this stage, the key functionality we need to add to this class involves writing the image file to the filesystem and deleting the file when the record is removed. Before we add this functionality, however, we will look at how to upload files via HTTP in PHP.

Uploading Files

Traditionally speaking, HTTP hasn't been a very good method for uploading files over the Internet. There are several reasons for this:

- **Unreliable.** If a file upload doesn't complete, it is not possible to resume the upload, meaning large files may never be uploaded. Additionally, some browsers may decide after a prolonged period of time that an error has occurred, typically resulting in an error message being displayed to the user.
- **Restrictive.** While a file is being uploaded, the user cannot navigate away from the page they are uploading to without interrupting the upload.
- **Cumbersome.** Due to security concerns, the capabilities of file-upload forms are somewhat restricted. For instance, very few styles can usually be applied to file inputs. Additionally, file inputs allow only single selections, meaning a user cannot choose multiple files at once—if the form allows multiple file inputs, the files must be chosen one at a time.

- **Uninformative.** There is no built-in way in HTTP to notify the user of the status of their upload. This means there is no easy way to know how much of the upload is complete, or how much longer it will take.

Thankfully the increased speeds of Internet connections over recent years have alleviated some of these problems; however, since HTTP hasn't changed, these issues still exist.

In our web application, we will only be uploading images (not other file types, such as PDF files). Compared to other types of files, images are small. For instance, using the Photoshop "Save for Web" tool to save a 1024 × 768 pixel JPEG photo will typically result in a file under 100KB.

In this section, we will create an image-upload form in our web application, as well as a new form processor to deal with this upload.

Setting the Form Encoding

To upload files over HTTP, a traditional HTML form is used (that is, using `<form>` tags), but you must add one extra attribute to this tag: the `enctype` attribute. This notifies the web server what kind of data the web browser is trying to send.

Normally you don't need to specify this attribute. If it is not specified, the default value of `enctype` is `application/x-www-form-urlencoded`. In other words, the following two lines of HTML are equivalent:

```
<form method="post" action="...">
<form method="post" action="..." enctype="application/x-www-form-urlencoded">
```

This indicates to the web server that the browser is sending URL-encoded form data using the HTTP POST method.

In order to have the web server recognize uploaded image files, we must specify the `enctype` as `multipart/form-data`. In other words, the form will probably be sending multiple types of data: normal URL-encoded form data as well as binary data (such as an image).

Adding the Form

Let's now add a new form to the web application that will allow users to upload images to their blog posts once they have been created. We will add the form shown in Listing 11-3 to the `preview.tpl` file in `./templates/blogmanager`.

■ **Note** We could include the image-upload form on the blog post editing page, but uploading files with normal form data can pose new challenges, since if a form error occurs, the user may need to upload the file again.

Listing 11-3. *Creating a File-Upload Form Specifying the Form Encoding Type (preview.tpl)*

```
<!-- // ... other code -->

<fieldset id="preview-tags">
    <!-- // ... other code -->
</fieldset>

<fieldset id="preview-images">
    <legend>Images</legend>

    <form method="post"
          action="{geturl action='images'}"
          enctype="multipart/form-data">

        <div>
            <input type="hidden" name="id" value="{$post->getId()}" />
            <input type="file" name="image" />
            <input type="submit" value="Upload Image" name="upload" />
        </div>
    </form>
</fieldset>

<div class="preview-date">
    <!-- // ... other code -->
</div>

<!-- // ... other code -->
```

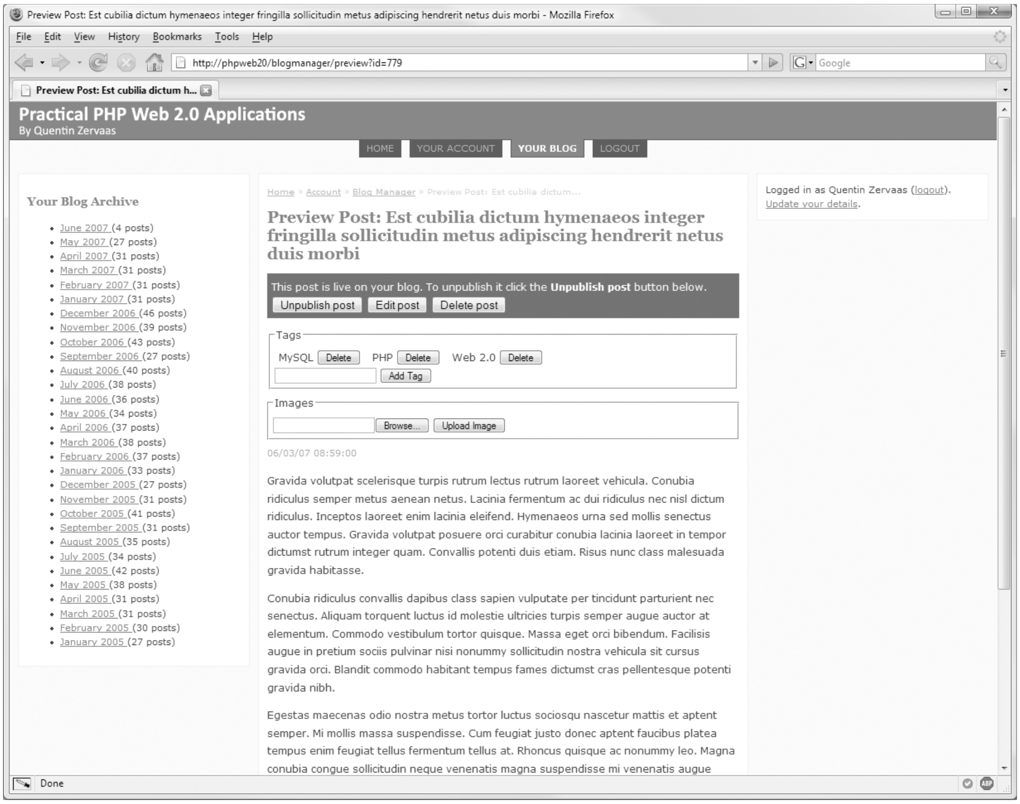
The target script for this form is a new action handler called `images` in the `blogmanager` controller. We will create this handler later. We also include the ID of the blog post the image is being uploaded for, so it can be linked to the post.

In addition to handling uploads, we will use the `images` action handler to save changes to the ordering of the images and to delete images. The submit button is named `upload` so we know that we are handling a file upload when processing this form.

By adding some new styles to the site style sheet (in `./htdocs/css/styles.css`), we can make this block look like the tag management area that is also on this page. Listing 11-4 shows the CSS we need to add to `styles.css`, while Figure 11-1 shows how the form looks on the blog post preview page.

Listing 11-4. *Styling the Image-Management Area of the Blog Post Preview (styles.css)*

```
#preview-images {
    margin     : 5px 0;
    padding    : 5px;
}

#preview-images input {
    font-size  : 0.95em;
}
```


Preview Post: Est cubilia dictum hymenaeos integer fringilla sollicitudin metus adipiscing hendrerit netus duis morbi - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://phpweb20/blogmanager/preview?id=779
Google
Preview Post: Est cubilia dictum h...
Practical PHP Web 2.0 Applications
By Quentin Zervaas
HOME
YOUR ACCOUNT
YOUR BLOG
LOGOUT
Home » Account » Blog Manager » Preview Post: Est cubilia dictum...
Unpublish post
Edit post
Delete post
Tags
MySQL Delete
PHP Delete
Web 2.0 Delete
Add Tag
Images
Browse...
Upload Image
Done

Your Blog Archive

- June 2007 (4 posts)
- May 2007 (27 posts)
- April 2007 (31 posts)
- March 2007 (31 posts)
- February 2007 (31 posts)
- January 2007 (31 posts)
- December 2006 (46 posts)
- November 2006 (39 posts)
- October 2006 (43 posts)
- September 2006 (27 posts)
- August 2006 (40 posts)
- July 2006 (38 posts)
- June 2006 (36 posts)
- May 2006 (34 posts)
- April 2006 (37 posts)
- March 2006 (38 posts)
- February 2006 (37 posts)
- January 2006 (33 posts)
- December 2005 (27 posts)
- November 2005 (31 posts)
- October 2005 (41 posts)
- September 2005 (31 posts)
- August 2005 (35 posts)
- July 2005 (34 posts)
- June 2005 (42 posts)
- May 2005 (38 posts)
- April 2005 (31 posts)
- March 2005 (31 posts)
- February 2005 (30 posts)
- January 2005 (27 posts)

Preview Post: Est cubilia dictum hymenaeos integer fringilla sollicitudin metus adipiscing hendrerit netus duis morbi

This post is live on your blog. To unpublish it click the **Unpublish post** button below.

06/03/07 08:59:00

Gravida volutpat scelerisque turpis rutrum lectus rutrum laoreet vehicula. Conubia ridiculus semper metus aenean netus. Lacinia fermentum ac dui ridiculus nec nisl dictum ridiculus. Inceptos laoreet enim lacinia eleifend. Hymenaeos urna sed mollis senectus auctor tempus. Gravida volutpat posuere orci curabitur conubia lacinia laoreet in tempor dictumst rutrum integer quam. Convallis potenti duis etiam. Risus nunc class malesuada gravida habitasse.

Conubia ridiculus convallis dapibus class sapien vulputate per tincidunt parturient nec senectus. Aliquam torquent luctus id molestie ultricies turpis semper augue auctor at elementum. Commodo vestibulum tortor quisque. Massa eget orci bibendum. Facilisis augue in pretium sociis pulvinar nisi nonummy sollicitudin nostra vehicula sit cursus gravida orci. Blandit commodo habitant tempus fames dictumst cras pellentesque potenti gravida nibh.

Egestas maecenas odio nostra metus tortor luctus sociosqu nascetur mattis et aptent semper. Mi mollis massa suspendisse. Cum feugiat justo donec aptent faucibus platea tempus enim feugiat tellus fermentum tellus at. Rhoncus quisque ac nonummy leo. Magna conubia congue sollicitudin neque venenatis magna suspendisse mi venenatis augue

Logged in as Quentin Zervaas (logout).
Update your details.

Figure 11-1. *The image management area for blog posts*

Specifying the File Input Type

The other element in Listing 11-3 that we have not yet discussed is the file input. This is a special type of form element that allows the user to select a file from their computer for upload. It is typically made up of a text input on the left and a button on the right (used to open the file-selection dialog box).

Browsers typically give developers less control over the look and feel of file inputs than for other inputs, as there would be security implications if they did not do this. Here are some of the things you can and can't do with file inputs (although different browsers will behave slightly differently):

- You can observe the `onchange` event, so you can detect when the user has chosen a file (or removed their selection).
- You can retrieve the value of the form element. This does not mean you can read the contents of the selected file—you can simply read the path and/or filename of the file as it is stored on the user's computer.
- You can change the font size and color of the file input element, but you cannot change the text (most browsers will use "`Browse…`").

- You cannot use a custom image as the browse button, nor can you hide the text input that shows the file path. However, you can manipulate the input by changing its position in CSS or making it fully transparent (allowing you to add styled buttons behind it).

Some web developers have been quite creative in how they style this input in an effort to customize their own site layout fully. We will be using the plain-vanilla version of the file input control, as shown in Listing 11-3.

Setting the Maximum File Size

The next step is to look at how maximum file upload sizes are specified. You will want to impose some kind of restriction on the maximum size of uploaded files to prevent abuse from users. There are several ways to achieve this, both within the HTML form as well as on the server:

- `MAX_FILE_SIZE`: By including a hidden form element called `MAX_FILE_SIZE`, you can set the maximum number of bytes in an uploaded file by specifying that value as the form element value. Using this feature is somewhat pointless, since it can easily be fooled by somebody manually manipulating the form data.
- `post_max_size`: This is a `php.ini` setting that specifies the maximum size POST request data can be. Note that if there are several files being uploaded within a single form, this value applies to the total amount of data being included. In a default PHP installation, this value is set to 8MB (using the value `8M`).
- `upload_max_filesize`: This `php.ini` setting specifies the maximum size for a single file that is uploaded. By default, this value is set to 2MB. In combination with the `post_max_size` value (of `8M`), you could upload three 1.5MB files (since each is below 2MB and the total is approximately 4.5MB), but uploading 10 1MB files would fail since the total would be about 10MB.

You should use the `post_max_size` and `upload_max_filesize` settings to specify upload limits and ignore the `MAX_FILE_SIZE` form directive. In addition to these limits, you may also want to impose other artificial limits on users, such as the maximum number of photos per blog post or a total quota for their account.

Realistically, you won't need to make any changes to your configuration to deal with maximum file sizes. However, if you wanted to allow users to upload other types of files (such as PDF or MP3 files), you would want to increase these configuration settings.

■**Note** We won't be implementing any such restrictions in this chapter; however, as an exercise, you may want to add this functionality. Note that if you choose to restrict the number of photos per blog post, the user can get around this limit by simply creating more posts. As such, using an absolute number as a restriction may be a better solution (such as 20MB per user).

Handling Uploaded Files

As mentioned previously, we must create a new action handler in the BlogmanagerController class to deal with image-handling operations. The different operations that can take place include uploading, reordering, and deleting. At this stage, we will only look at the upload operation.

In addition to creating the new action handler, we must also create a new form processor to save the uploaded files as well as to report on any errors that may have occurred.

Creating the Blog Manager Action Handler

We can use the tagsAction() function we created in Chapter 10 as a basis for the new imagesAction() function. The functionality of these two functions is almost identical: in both we must first load a blog post; next, we must determine the action to take (in this case, it's whether to upload, reorder, or delete an image; in tagsAction() it was whether to add or delete a tag); finally, we will redirect the browser back to the blog post preview.

Listing 11-5 shows the code we will add to the BlogmanagerController.php class (in ./include/Controllers) in order to manage images. For now we will simply include placeholders for the other image operations.

Listing 11-5. *The Action Handler for Image Management (BlogmanagerController.php)*

```
<?php
    class BlogmanagerController extends CustomControllerAction
    {
        // ... other code

        public function imagesAction()
        {
            $request = $this->getRequest();

            $post_id = (int) $request->getPost('id');

            $post = new DatabaseObject_BlogPost($this->db);
            if (!$post->loadForUser($this->identity->user_id, $post_id))
                $this->_redirect($this->getUrl());

            if ($request->getPost('upload')) {
                $fp = new FormProcessor_BlogPostImage($post);
                if ($fp->process($request))
                    $this->messenger->addMessage('Image uploaded');
                else {
                    foreach ($fp->getErrors() as $error)
                        $this->messenger->addMessage($error);
                }
            }
            else if ($request->getPost('reorder')) {
                // todo
            }
            else if ($request->getPost('delete')) {
                // todo
            }

            $url = $this->getUrl('preview') . '?id=' . $post->getid();
            $this->_redirect($url);
        }
    }
?>
```

One key aspect of this code is that we now use the flash messenger to hold any error messages that occur in the upload form. This means that if any errors occur (such as a file being too large or a file type being invalid), the error messages will be shown in a message area at the top of the right column of our web application.

Showing errors in this manner is slightly different from what we have done so far in this book (previously errors have been shown below the form input related to the error). I have simply done this to show you an alternative way of displaying error messages.

Creating the Image-Upload Form Processor

In Listing 11-5, we used a class called `FormProcessor_BlogPostImage`. We will now create this class, which we will use to process the uploaded image. This class has several responsibilities:

- Ensuring the upload completed correctly
- Checking the type of file that was uploaded and ensuring it is an image
- Writing the file to the filesystem and creating the database record using the `DatabaseObject_BlogPostImage` class we created earlier in this chapter

Listing 11-6 shows the constructor for this class, which we will store in a file called `BlogPostImage.php` in the `./include/FormProcessor` directory. The constructor instantiates `DatabaseObject_BlogPost` and sets the ID of the blog post the image is being uploaded for.

Listing 11-6. *The Constructor of the Image-Upload Processing Form (BlogPostImage.php)*

```
<?php
    class FormProcessor_BlogPostImage extends FormProcessor
    {
        protected $post;
        public $image;

        public function __construct(DatabaseObject_BlogPost $post)
        {
            parent::__construct();

            $this->post = $post;

            // set up the initial values for the new image
            $this->image = new DatabaseObject_BlogPostImage($post->getDb());
            $this->image->post_id = $this->post->getId();
        }
```

Next, we will implement the `process()` method of this class, which will process any uploaded images. As we saw earlier in this chapter, we must specify the `multipart/form-data` encoding type to upload files using HTML forms.

When we set this attribute, PHP will know to create the superglobal array called `$_FILES`, which stores information about uploaded files (even though the form is submitted using HTTP POST, the image-upload information is stored in `$_FILES`, not in `$_POST`). There is one entry in `$_FILES` for each file that is uploaded, with the array key being the value of the `name` attribute in the form input (in our case, we used `image`).

Each element in `$_FILES` is an array consisting of the following elements:

- `name`: The original filename of the uploaded file as it was stored on the client computer (typically not including the path). We will store this value in the database for each uploaded image.
- `type`: The mime type of the uploaded file. For example, if the uploaded file was a PNG image, this would have a value of `image/png`. Since this is set by the browser, we should not trust this value. In the `process()` method we will not use this value and instead will verify the type of data manually.
- `size`: The size of the uploaded file in bytes. If you want to impose a restriction on the size of uploaded files (in addition to the PHP configuration settings), you can use this value.
- `tmp_name`: The full path on the server where the uploaded file is stored. This is a temporary location, so you must move or copy the file from this location in order to keep it (we will do this shortly using the `move_uploaded_file()` function).
- `error`: The error code associated with the uploaded file. There are several different codes that can be set (which we will look at in the following code). We must check this value using the built-in PHP constants and generate an appropriate error message. If the file upload is successful, the value of `error` will be `0` (which we can check using the constant `UPLOAD_ERR_OK`).

To begin implementing the `process()` method, the first thing we must do is check for the presence of the uploaded file in the `$_FILES` superglobal. This is shown in Listing 11-7. Once we know it exists, we can assign it to the `$file` variable.

Listing 11-7. *Ensuring the $_FILES Array Is Set Correctly (BlogPostImage.php)*

```
        public function process(Zend_Controller_Request_Abstract $request)
        {
            if (!isset($_FILES['image']) || !is_array($_FILES['image'])) {
                $this->addError('image', 'Invalid upload data');
                return false;
            }

            $file = $_FILES['image'];
```

Next, we will check the error code, as set in the `error` element of `$file`. If this value is not equal to `UPLOAD_ERR_OK`, an error has occurred. We will check for each error code explicitly, as this allows us to create a more informative error message for the user. (These codes are documented at `http://www.php.net/manual/en/features.file-upload.errors.php`.) Listing 11-8 shows the `switch()` statement in `BlogPostImage.php` that will check each of the different error codes.

Listing 11-8. *Checking the Error Code for the Uploaded Image (BlogPostImage.php)*

```
        switch ($file['error']) {
            case UPLOAD_ERR_OK:
                // success
                break;

            case UPLOAD_ERR_FORM_SIZE:
                // only used if MAX_FILE_SIZE specified in form
            case UPLOAD_ERR_INI_SIZE:
                $this->addError('image', 'The uploaded file was too large');
                break;

            case UPLOAD_ERR_PARTIAL:
                $this->addError('image', 'File was only partially uploaded');
                break;

            case UPLOAD_ERR_NO_FILE:
                $this->addError('image', 'No file was uploaded');
                break;

            case UPLOAD_ERR_NO_TMP_DIR:
                $this->addError('image', 'Temporary folder not found');
                break;

            case UPLOAD_ERR_CANT_WRITE:
                $this->addError('image', 'Unable to write file');
                break;

            case UPLOAD_ERR_EXTENSION:
                $this->addError('image', 'Invalid file extension');
                break;

            default:
                $this->addError('image', 'Unknown error code');
        }

        if ($this->hasError())
            return false;
```

In this code, note that if an error has occurred, we return from the process() function immediately, since there's nothing else to do. The remainder of the code relies on a file being successfully uploaded.

Next, we must ensure that the uploaded file is in fact an image. Since we cannot rely on the mime type specified by the user's web browser, we must check the data manually. This is fairly straightforward for images, since PHP has the getImageSize() function, which returns an array of information about image files. This function will return false if the file is not an image.

The getImageSize() function supports a wide range of image types, but since we only want to allow JPEG, GIF, and PNG files (since these are the three types of files commonly supported in web browsers), we must first check the type of image. The getImageSize() function returns an array: the first and second elements are the width and height of the image, and the third element (index of 2) specifies the image type.

Listing 11-9 shows the code we will add to fetch the image information and check its type. We will use built-in constants to check for JPEG, GIF, and PNG images.

Listing 11-9. *Ensuring the Uploaded File Is a JPEG, GIF, or PNG Image (BlogPostImage.php)*

```
            $info = getImageSize($file['tmp_name']);
            if (!$info) {
                $this->addError('type', 'Uploaded file was not an image');
                return false;
            }

            switch ($info[2]) {
                case IMAGETYPE_PNG:
                case IMAGETYPE_GIF:
                case IMAGETYPE_JPEG:
                    break;

                default:
                    $this->addError('type', 'Invalid image type uploaded');
                    return false;
            }
```

At this point in the code, we can assume a valid file was uploaded and that it is a JPEG, GIF, or PNG image (it doesn't matter to us which one). Now we must write the file to the filesystem (it is currently stored in a temporary area) and save the database record.

To move the file from the temporary area, we will call the uploadFile() method, which we will implement shortly. Additionally, we will set the filename of the uploaded file and save the database record, as shown in Listing 11-10.

Listing 11-10. *Saving the Image File and the Database Record (BlogPostImage.php)*

```
            // if no errors have occurred, save the image
            if (!$this->hasError()) {
                $this->image->uploadFile($file['tmp_name']);
                $this->image->filename = basename($file['name']);
                $this->image->save();
            }

            return !$this->hasError();
        }
    }
?>
```

■**Note** Be sure to use `basename()` on the value in `$file['name']`, since this value is supplied by the browser. The `basename()` method is used to strip out the path from a full filesystem path (so `/path/to/foo.jpg` becomes `foo.jpg`). As mentioned earlier, most browsers will not include the full path, but you should still call `basename()` just in case.

Writing Files to the Filesystem

Now that we have completed the action handler and the form processor, we must make the necessary changes to the `DatabaseObject_BlogPostImage` class to save the uploaded image. There are a number of functions we must write, including the `uploadFile()` function we briefly looked at in Listing 11-10.

The first function we will write is one that returns the path on the filesystem where we will be storing the uploaded images. In Chapter 1 we created a directory called `uploaded-files` in the `./data` directory—this is where the uploaded images will be stored. Listing 11-11 shows `GetUploadPath()`, a static function we will call to determine where files will be stored.

Listing 11-11. *Determining the Base Location for Uploaded Files (BlogPostImage.php)*

```
<?php
    class DatabaseObject_BlogPostImage extends DatabaseObject
    {
        // ... other code

        public static function GetUploadPath()
        {
            $config = Zend_Registry::get('config');

            return sprintf('%s/uploaded-files', $config->paths->data);
        }
    }
?>
```

Next, we will write a function to determine the full path where an uploaded file is stored for a particular database record. To simplify this process, rather than storing files with the names they used on the client's computer, we will store them in the uploaded file directory using their database ID. If we need to refer back to their original filenames, we can get this information from the database record.

Listing 11-12 shows the `getFullpath()` function, which returns the full path to the uploaded file. This basically just combines the `GetUploadPath()` function with the record ID.

Listing 11-12. *Retrieving the Full Filesystem Path of an Uploaded File (BlogPostImage.php)*

```
<?php
    class DatabaseObject_BlogPostImage extends DatabaseObject
    {
        // ... other code

        public function getFullPath()
        {
            return sprintf('%s/%d', self::GetUploadPath(), $this->getId());
        }

        // ... other code
    }
?>
```

Next, we will implement the `uploadFile()` function. All this function does is store the temporary path of the uploaded file in anticipation of the `save()` method being called. When `save()` is called on a new record of `DatabaseObject_BlogPostImage`, the `preInsert()` and `postInsert()` callbacks will be executed. The copying of the file from its temporary location to its new location will occur on `postInsert()`.

Listing 11-13 shows the code for `uploadFile()`, which writes the temporary path to an object property for later use. Note that it also does some basic error checking to ensure the temporary file exists and is readable.

Listing 11-13. *Setting the Location of the Uploaded File so It Can Be Copied Across (BlogPostImage.php)*

```
<?php
    class DatabaseObject_BlogPostImage extends DatabaseObject
    {
        protected $_uploadedFile;

        // ... other code

        public function uploadFile($path)
        {
            if (!file_exists($path) || !is_file($path))
                throw new Exception('Unable to find uploaded file');

            if (!is_readable($path))
                throw new Exception('Unable to read uploaded file');

            $this->_uploadedFile = $path;
        }
        // ... other code
    }
?>
```

Next, we will implement the preInsert() callback, which is called before the database record is inserted into the database. This function first ensures that the upload location exists and is writable, which will help us solve any permissions errors if the upload area hasn't been created properly. Then the ranking value for the image is determined, based on the other images that have been uploaded for the blog post. The ranking system simply uses numbers from 1 to *N*, where *N* is the number of images for a single post.

Since the new image is considered to be the last image for the blog, we can use the SQL max() function to determine its ranking. The only problem with this is that if no images exist for the given blog post, a value of null is returned. To avoid this problem, we will use the coalesce() function, which returns the first non-null value from its arguments.

The code for preInsert() is shown in Listing 11-14.

Listing 11-14. *Ensuring the File Can Be Written, and Determining Its Ranking (BlogPostImage.php)*

```
<?php
    class DatabaseObject_BlogPostImage extends DatabaseObject
    {
        // ... other code

        public function preInsert()
        {
            // first check that we can write the upload directory
            $path = self::GetUploadPath();
            if (!file_exists($path) || !is_dir($path))
                throw new Exception('Upload path ' . $path . ' not found');

            if (!is_writable($path))
                throw new Exception('Unable to write to upload path ' . $path);

            // now determine the ranking of the new image
            $query = sprintf(
                'select coalesce(max(ranking), 0) + 1 from %s where post_id = %d',
                $this->_table,
                $this->post_id
            );

            $this->ranking = $this->_db->fetchOne($query);
            return true;
        }

        // ... other code
?>
```

Finally, we will implement the postInsert() callback. This is the function responsible for copying the image file from its temporary upload location to the uploaded files area of our web application. We will do this in postInsert() because if any SQL errors occurred before this point, the whole transaction could be easily rolled back, preventing the file from being incorrectly moved into the web application.

To move the file, we will use the PHP move_uploaded_file() function. This function is used for security reasons, as it will automatically ensure that the file being moved was in fact uploaded via PHP. This function will return true if the file was successfully moved and false if not. Thus we can use the return value as the postInsert() return value. Remember that returning false from this callback will roll back the database transaction. In other words, if the file could not be copied for some reason, the database record would not be saved.

Listing 11-15 shows the postInsert() method, which completes the image-upload functionality for the web application.

Listing 11-15. *Moving the Uploaded File to the Application File Storage Area (BlogPostImage.php)*

```
<?php
    class DatabaseObject_BlogPostImage extends DatabaseObject
    {
        // ... other code

        public function postInsert()
        {
            if (strlen($this->_uploadedFile) > 0)
                return move_uploaded_file($this->_uploadedFile,
                                          $this->getFullPath());

            return false;
        }

        // ... other code
    }
?>
```

Once you have added this code, you will be able to upload images to blog posts via the form we added to the post preview page. Currently, though, we haven't implemented code to display these uploaded images, so to verify that your code is working, you should check that there are records present in the database table by using the following query:

```
mysql> select * from blog_posts_images;
```

You should also check that the file you uploaded is in /var/www/phpweb20/data/uploaded-files.

Sending Images

Now that users can upload photos to their blog posts, we must display their images both on the blog post preview page and on the actual blog page. Before we do this, however, we must write the code to send the images. We could use the built-in file serving from the web server, but since the original images as well as generated thumbnails will be stored in the application data directory, we will serve these files using PHP code.

To begin, we will simply output uploaded images in full, just as they were uploaded. We will build on this functionality later by adding the ability to resize images.

■ **Note** The image resizing we will implement will generate thumbnails on demand. In other words, the first time a thumbnail of a particular size is requested, it will be generated and saved for later reuse. The advantage of doing this over creating thumbnails when the image is uploaded is that we can easily choose what size thumbnails we want in the template rather than deciding in the PHP code at upload time.

To send blog post images, we will create a new action handler in the `UtilityController` class we created earlier in this book. Currently this controller is used only for outputting CAPTCHA images, but we will now make it also send blog post images.

Listing 11-16 shows the start of the `imageAction()` method we will add to `UtilityController.php`. This file can be found in `./include/Controllers`.

Listing 11-16. *Initial Setup of the Image-Output Function (UtilityController.php)*

```
<?php
    class UtilityController extends CustomControllerAction
    {
        // ... other code

        public function imageAction()
        {
            $request  = $this->getRequest();
            $response = $this->getResponse();

            $id = (int) $request->getQuery('id');

            // disable autorendering since we're outputting an image
            $this->_helper->viewRenderer->setNoRender();
```

As in many other action handlers in this book, we begin by retrieving the request object. In this function, we also retrieve the response object because we are going to send some additional HTTP headers. Namely, we are going to send the `content-type` header (to specify the type of data) and `content-length` header (to specify the amount of data in bytes).

Next, we retrieve the requested image ID from the URL. This means that to request the image with an ID of 123, the URL `http://phpweb20/utility/image?id=123` would be used.

The next step is to disable the automatic view renderer, since we are outputting an image and not an HTML template.

At this point, we will try to load the `DatabaseObject_BlogPostImage` record specified by the `$id` variable, as shown in Listing 11-17. If the image cannot be found, we use the `$response` object to send a 404 header and return. If the image does load, we simply proceed in the function.

Listing 11-17. *Loading the DatabaseObject_BlogPostImage Record (UtilityController.php)*

```
            $image = new DatabaseObject_BlogPostImage($this->db);
            if (!$image->load($id)) {
                // image not found
                $response->setHttpResponseCode(404);
                return;
            }
```

At this point in the function, we can assume that a blog post image has been successfully loaded. As such, we must now determine what type of image it is and send the appropriate `content-type` header. The `getImageSize()` function we looked at earlier in this chapter also includes an appropriate header in the `mime` index of the returned array.

In addition to sending this header, we will also send the `content-length` header. This tells the browser how much data to expect. We can use the PHP `filesize()` function to determine the value for this (specified in bytes).

■ **Note** Why is the `content-length` header important? Perhaps you have downloaded a large file in your browser, and the browser was unable to give you an estimate of remaining time. This is because the `content-length` header was not sent. The browser simply receives data until no more is available—without the header, it is not able to determine how much data is still to come. This is usually more of an issue for larger files.

Listing 11-18 shows the remainder of the `imageAction()` function. This code begins by retrieving the full filesystem path using `getFullPath()`. It then determines which type of image the file is and sends headers for the type and the size of the image using the `setHeader()` function. Finally, the actual image data is sent.

Listing 11-18. *Sending the Image Headers and Then the Image Itself (UtilityController.php)*

```
            $fullpath = $image->getFullPath();
            $info = getImageSize($fullpath);

            $response->setHeader('content-type', $info['mime']);
            $response->setHeader('content-length', filesize($fullpath));
            echo file_get_contents($fullpath);
        }
    }
?>
```

This completes the minimum code required to output uploaded blog post images. In order to test it, you can upload an image using the form we created earlier in this chapter, and then manually enter the image ID into the following URL: `http://phpweb20/utility/image?id=`*`ImageID`*.

Resizing Images

Depending on the context, you will often want to display uploaded images at different resolutions in different areas of your site. For example, on the blog post index page in our application, you might want small thumbnails (perhaps around about 100 pixels by 75 pixels) while on the blog post detail page you might want to show somewhat larger images (such as about 200 pixels by 150 pixels). In addition, you may want to allow the user to click on an image to show the image at full size.

In this section, we will build a simple mechanism to generate resized versions (that is, thumbnails) of uploaded images. We will build this system such that the desired dimensions can be specified in the URL and an image the appropriate size will be returned.

In order to do this, we will use the GD functions that are included with PHP. A popular alternative to GD is ImageMagick, but it requires that ImageMagick also be installed on the server, while GD is typically included in most PHP installations.

■ **Note** The techniques we use here can be achieved using ImageMagick's `convert` tool. You can use this tool either by calling `convert` directly within your script, or by using the `imagick` PECL package. After lying dormant for several years, this package has recently gained new life and provides a simple interface to ImageMagick. The biggest drawback to using this package is that it needs to be built into the PHP server in addition to ImageMagick being installed on the server. More information about `convert` can be found at `http://www.imagemagick.org/script/convert.php`.

Creating Thumbnails

The image thumbnailer we will now create is fairly straightforward when you look at the individual pieces. We will create a new method in the `DatabaseObject_BlogPostImage` class called `createThumbnail()`, which generates a thumbnail for the loaded record based on the width and height arguments specified. This method will return the full filesystem path to the created thumbnail, which allows us to easily link the thumbnailer into the existing code to load and display images. Additionally, it allows us to simply return the path of the original file if the requested thumbnail is bigger than the original image. This saves unnecessary duplication of the image on the filesystem.

The other thing `createThumbnail()` will do is cache the thumbnails. Since creating thumbnails can be processor-intensive (depending on the size of the input and output images), we want to make this process as efficient as possible. Fortunately, it is very straightforward to cache the created thumbnails, as we will soon see.

Before we write `createThumbnail()`, we will add in another method, which we will call `GetThumbnailPath()`. This method will return the filesystem path to where created thumbnails should be stored. We will use the `./data/tmp` directory as the base directory for this, and use a subdirectory within it called `thumbnails`, as shown in Listing 11-19.

Listing 11-19. *Retrieving the Thumbnail Storage Path (BlogPostImage.php)*

```
<?php
    class DatabaseObject_BlogPostImage extends DatabaseObject
    {
        // ... other code

        public static function GetThumbnailPath()
        {
            $config = Zend_Registry::get('config');

            return sprintf('%s/tmp/thumbnails', $config->paths->data);
        }
    }
?>
```

Next, we can look at `createThumbnail()`, in which we begin by retrieving the path of the original file and some basic information about this file, which we will use later in the function. Listing 11-20 shows beginning of `createThumbnail()`.

Listing 11-20. *Determining the Image Attributes for Later Use (BlogPostImage.php)*

```
<?php
    class DatabaseObject_BlogPostImage extends DatabaseObject
    {
        // ... other code

        public function createThumbnail($maxW, $maxH)
        {
            $fullpath = $this->getFullpath();

            $ts = (int) filemtime($fullpath);
            $info = getImageSize($fullpath);
```

Determining the Width and Height of the Thumbnail

The first (and probably the most complicated) step of creating a thumbnail image is to determine the dimensions of the thumbnail. The `createThumbnail()` function accepts the maximum width of a thumbnail as its first argument, and the maximum height as the second argument. Note that the proportions remain the same as the original regardless of the specified width and height; we simply use these values to determine the maximum size.

We will allow for either of these arguments (but not both) to be set to 0. This means the image will be constrained only by the specified value (so if a maximum width of 100 is specified with a maximum height of 0, the image can be any height as long as it is no wider than 100 pixels).

We use the width and height values returned from `getImageSize()` in combination with the specified maximum width and height (`$maxW` and `$maxH`) to determine the width and height of the thumbnail (`$newW` and `$newH`). This code is shown in Listing 11-21, and is explained in the comments.

Listing 11-21. *Calculating the Width and Height of the Thumbnail (BlogPostImage.php)*

```
$w = $info[0];          // original width
$h = $info[1];          // original height

$ratio = $w / $h;       // width:height ratio

$maxW = min($w, $maxW); // new width can't be more than $maxW
if ($maxW == 0)         // check if only max height has been specified
    $maxW = $w;

$maxH = min($h, $maxH); // new height can't be more than $maxH
if ($maxH == 0)         // check if only max width has been specified
    $maxH = $h;

$newW = $maxW;          // first use the max width to determine new
$newH = $newW / $ratio; // height by using original image w:h ratio

if ($newH > $maxH) {        // check if new height is too big, and if
    $newH = $maxH;          // so determine the new width based on the
    $newW = $newH * $ratio; // max height
}

if ($w == $newW && $h == $newH) {
    // no thumbnail required, just return the original path
    return $fullpath;
}
```

Determining the Input and Output Functions

In order to create thumbnails with GD, we must turn the original image into a GD image resource (a special type of PHP variable). There is a different function to do this for each of the image types we support (JPEG, GIF, and PNG).

Once the thumbnail has been created, we need to output the new GD image resource to the filesystem. We must determine which function to use for this, also based on the type of image. While we could simply use the same image type for all thumbnails, we will use the input image type as the output image type.

Just as we did when writing the image uploader (Listing 11-9), we can check the third index of the `getImageSize()` result to determine which functions to use. This is shown in Listing 11-22.

Listing 11-22. *Determining the GD Input and Output Image Functions (BlogPostImage.php)*

```
switch ($info[2]) {
    case IMAGETYPE_GIF:
        $infunc = 'ImageCreateFromGif';
        $outfunc = 'ImageGif';
        break;
```

```
            case IMAGETYPE_JPEG:
                $infunc = 'ImageCreateFromJpeg';
                $outfunc = 'ImageJpeg';
                break;

            case IMAGETYPE_PNG:
                $infunc = 'ImageCreateFromPng';
                $outfunc = 'ImagePng';
                break;

            default;
                throw new Exception('Invalid image type');
        }
```

Generating the Thumbnail Filename

Next, we will generate a filename for the newly created thumbnail. We generate this based on the height and width of the thumbnail, as well as on the image ID and the date the original file was created. By using the creation date, the thumbnail will be regenerated if the file is ever modified.

■**Note** We haven't actually implemented functionality to allow the user to edit an uploaded image, but if you did, this timestamp would ensure new thumbnails would be generated automatically for the new image.

In addition to creating the filename, we must also determine the full path of the thumbnail and ensure that we can write to that directory, as shown in Listing 11-23. If the destination directory doesn't exist, we will create it. Note that this will typically only occur the first time this function is called.

Listing 11-23. *Generating the Thumbnail Filename, and Creating the Target Directory (BlogPostImage.php)*

```
        // create a unique filename based on the specified options
        $filename = sprintf('%d.%dx%d.%d',
                            $this->getId(),
                            $newW,
                            $newH,
                            $ts);

        // autocreate the directory for storing thumbnails
        $path = self::GetThumbnailPath();
        if (!file_exists($path))
            mkdir($path, 0777);

        if (!is_writable($path))
            throw new Exception('Unable to write to thumbnail dir');
```

Creating the Thumbnail

Now that we know the dimensions of the thumbnail, the input and output functions, and the thumbnail destination path, we can create the actual thumbnail. The very first thing we will do, however, is check whether the thumbnail already exists. This simple check (in combination with the previous filename generation) is the caching functionality. If the thumbnail exists, we simply skip the generation part of this code.

If the thumbnail doesn't exist, we read in the image to GD using the input determined in Listing 11-22. So if the original image is a PNG file, `ImageCreateFromPng()` is used. If an error occurs reading the image, we throw an exception and return from the function. The first portion of the thumbnail-creation code is shown in Listing 11-24.

Listing 11-24. *Reading the Image into GD (BlogPostImage.php)*

```
        // determine the full path for the new thumbnail
        $thumbPath = sprintf('%s/%s', $path, $filename);

        if (!file_exists($thumbPath)) {

            // read the image in to GD
            $im = @$infunc($fullpath);
            if (!$im)
                throw new Exception('Unable to read image file');
```

When resizing an image with GD, the original image resource remains unchanged while the resized version is written to a secondary GD image resource (which in this case we will call `$thumb`). We must first create this secondary GD image using `ImageCreateTrueColor()`, with the `$newW` and `$newH` variables specifying the size.

We then use GD's `ImageCopyResampled()` function to copy a portion of the source image (`$im`) to the `$thumb`. The target image resource is the first argument, and the source image resource is the second argument.

The remainder of the arguments indicate the X and Y coordinates of the target and source images respectively, followed by the width and height of both images. This function is fairly powerful, and it also allows you to easily crop or stretch images.

The code to create the target image and resample the original image onto the new image is shown in Listing 11-25.

Listing 11-25. *Resampling the Original Image onto the New Image Resource (BlogPostImage.php)*

```
            // create the output image
            $thumb = ImageCreateTrueColor($newW, $newH);

            // now resample the original image to the new image
            ImageCopyResampled($thumb, $im, 0, 0, 0, 0, $newW, $newH, $w, $h);
```

Finally, we write this new image to the filesystem using the output function we selected in Listing 11-22 (stored in `$outfunc`). So if the original image was a PNG image, we would use `ImagePng()` to write the image to disk.

■**Note** The second argument to the output functions (`ImagePng()`, `ImageJpeg()`, and `ImageGif()`) specifies where on the filesystem the image file should be written. If this isn't specified, the image data is output directly to the browser. You could choose to take advantage of this if you didn't want to write the generated images to the filesystem.

Finally, we ensure that the image was written to the system, and if so we return the path to the newly created thumbnail. Listing 11-26 shows the code that writes the image to the filesystem and returns from `createThumbnail()`.

Listing 11-26. *Writing the Thumbnail and Returning from createThumbnail() (BlogPostImage.php)*

```
                $outfunc($thumb, $thumbPath);
            }

            if (!file_exists($thumbPath))
                throw new Exception('Unknown error occurred creating thumbnail');
            if (!is_readable($thumbPath))
                throw new Exception('Unable to read thumbnail');

            return $thumbPath;
        }

        // ... other code
    }
?>
```

Linking the Thumbnailer to the Image Action Handler

Now that we have the capability to easily create image thumbnails, we must hook this into our web application. We will do this by making some simple modifications to the `imageAction()` function we created in Listing 11-16.

We are going to provide the ability to specify the desired width and height in the URL, so it will be extremely simple to generate thumbnails as required. This means you can decide on the dimensions of the thumbnail in the templates that output the image, rather than having to hard-code these dimensions in your PHP code.

Because users could potentially abuse a system that allows them to generate thumbnails of any size, we will add a mechanism to make it more difficult for this to occur. This mechanism works as follows:

1. When an image is requested, the URL must include a parameter called a `hash` in addition to the image ID, width, and height. This parameter will be generated based on the ID, width, and height.

2. The `imageAction()` method will check the supplied hash against what the hash should be for the combination of ID, width, and height.

3. If the two hash values are different, we will assume the image was requested incorrectly, and a 404 error is sent back.
4. If the hash value is correct, we generate the thumbnail and send it back.

If the user manually changes the width or height in the URL, the hash will not match the request, so the thumbnail won't be generated.

Generating an Image Hash

To implement this system, we first need the ability to generate an image hash based on the given parameters. We will use this method both in the generation of URLs in the template, as well as to generate a hash based on the ID, width, and height supplied in the request URL.

Listing 11-27 shows the `GetImageHash()` method, which generates a string based on the supplied arguments using `md5()`. This code should be added to the `BlogPostImage.php` file in `./include/DatabaseObject`.

Listing 11-27. *Generating a Hash for the Given Image ID, Width, and Height (BlogPostImage.php)*

```
<?php
    class DatabaseObject_BlogPostImage extends DatabaseObject
    {
        // ... other code

        public static function GetImageHash($id, $w, $h)
        {
            $id = (int) $id;
            $w  = (int) $w;
            $h  = (int) $h;

            return md5(sprintf('%s,%s,%s', $id, $w, $h));
        }

        // ... other code
    }
?>
```

Generating Image Filenames

Next, we will implement a new Smarty plug-in called `imagefilename`, which is used to generate image filenames using the desired image ID, width, and height. This plug-in will allow us to include image thumbnails in our templates very easily.

For example, to include a thumbnail that is 100 pixels by 75 pixels of an image with an ID of 12, the following code would be used in the template:

```
<img src="{imagefilename id=12 w=100 h=75}" alt="" />
```

Based on the arguments in this example, we would want to generate a URL as follows:

```
/utility/image?id=12&w=100&h=75&hash=[hash]
```

Similarly, if you wanted to generate the image path for the full-sized image, you would use the following:

```
<img src="{imagefilename id=12}" alt="" />
```

In order to generate this URL, we would use the `{geturl}` plug-in created earlier, in conjunction with the arguments and the `GetImageHash()` method.

Listing 11-28 shows the code for the `function.imagefilename.php` file, which we will store in `./include/Templater/plugins`.

Listing 11-28. *The imagefilename Plug-In, Used to Generate a Thumbnail Image Path (function.imagefilename.php)*

```
<?php
    function smarty_function_imagefilename($params, $smarty)
    {
        if (!isset($params['id']))
            $params['id'] = 0;

        if (!isset($params['w']))
            $params['w'] = 0;

        if (!isset($params['w']))
            $params['h'] = 0;

        require_once $smarty->_get_plugin_filepath('function', 'geturl');

        $hash = DatabaseObject_BlogPostImage::GetImageHash(
            $params['id'],
            $params['w'],
            $params['h']
        );

        $options = array(
            'controller' => 'utility',
            'action'     => 'image'
        );

        return sprintf(
            '%s?id=%d&w=%d&h=%d&hash=%s',
            smarty_function_geturl($options, $smarty),
            $params['id'],
            $params['w'],
            $params['h'],
            $hash
        );
    }
?>
```

This function begins by initializing the parameters (the image ID, as well as the desired width and height). Next, it loads the geturl plug-in so we can generate the /utility/image part of the URL (the controller and action values are specified in the $options array that we create in this function). Next, we generate the hash for the given ID, width, and height, and then finally combine all of the parameters together into a single string and return this value from the plug-in.

Updating imageAction() to Serve the Thumbnail

We can now update the imageAction() method to look for the w, h, and hash parameters so a thumbnail can be served if required. We simply need to generate a new hash based on the id, w, and h parameters, and then compare it to the hash value in the URL. Once we have determined that the supplied hash is valid and that the image could be loaded, we continue on by generating the thumbnail and sending it.

Instead of calling getFullPath(), we will call createThumbnail(), which returns the full path to the generated thumbnail. Since createThumbnail() throws various exceptions, we will call getFullPath() as a fallback. In other words, if the thumbnail creation fails for some reason, the original image is displayed instead. You may prefer instead to output an error.

The other code in imageAction() operated on the returned path from getFullPath(), so we don't need to change any of it—createThumbnail() also returns a full filesystem path.

Listing 11-29 shows the new version of imageAction(), which belongs in the UtilityController.php file in ./include/Controllers.

Listing 11-29. *Modifying imageAction() to Output Thumbnails on Demand (UtilityController.php)*

```
<?php
    class UtilityController extends CustomControllerAction
    {
        // ... other code

        public function imageAction()
        {
            $request  = $this->getRequest();
            $response = $this->getResponse();

            $id = (int) $request->getQuery('id');
            $w  = (int) $request->getQuery('w');
            $h  = (int) $request->getQuery('h');
            $hash = $request->getQuery('hash');

            $realHash = DatabaseObject_BlogPostImage::GetImageHash($id, $w, $h);

            // disable autorendering since we're outputting an image
            $this->_helper->viewRenderer->setNoRender();

            $image = new DatabaseObject_BlogPostImage($this->db);
            if ($hash != $realHash || !$image->load($id)) {
                // image not found
                $response->setHttpResponseCode(404);
                return;
            }

            try {
                $fullpath = $image->createThumbnail($w, $h);
            }
            catch (Exception $ex) {
                $fullpath = $image->getFullPath();
            }

            $info = getImageSize($fullpath);

            $response->setHeader('content-type', $info['mime']);
            $response->setHeader('content-length', filesize($fullpath));
            echo file_get_contents($fullpath);
        }
    }
?>
```

Managing Blog Post Images

Now that we have the ability to view uploaded images (both at their original size and as thumbnails) we can display the images on the blog post preview page.

In this section, we will modify the blog manager to display uploaded images, thereby allowing the user to easily delete images from their blog posts. Additionally, we will implement Ajax code using Prototype and Scriptaculous that will allow the user to change the order in which the images in a single post are displayed.

Automatically Loading Blog Post Images

Before we can display the images on the blog post preview page, we must modify `DatabaseObject_BlogPost` to automatically load all associated images when the blog post record is loaded. To do this, we will change the `postLoad()` function to automatically load the images.

Currently this function only loads the profile data for the blog post, but we will add a call to load the images, as shown in Listing 11-30. Additionally, we must initialize the `$images` array.

Listing 11-30. *Automatically Loading a Blog Post's Images When the Post Is Loaded (BlogPost.php)*

```
<?php
    class DatabaseObject_BlogPost extends DatabaseObject
    {
        public $images = array();

        // ... other code
        protected function postLoad()
        {
            $this->profile->setPostId($this->getId());
            $this->profile->load();

            $options = array(
                'post_id' => $this->getId()
            );
            $this->images = DatabaseObject_BlogPostImage::GetImages($this->getDb(),
                                                                    $options);

        }

        // ... other code
    }
?>
```

The code in Listing 11-30 calls a method called `GetImages()` in `DatabaseObject_BlogPostImage`, which we must now implement. This function, which we will add to `BlogPostImage.php` in `./include/DatabaseObject`, is shown in Listing 11-31. Note that we use the `ranking` field as the sort field. This ensures the images are returned in the order specified by the user (we will implement the functionality to change this order shortly).

Listing 11-31. *Retrieving Multiple Blog Post Images (BlogPostImage.php)*

```
<?php
    class DatabaseObject_BlogPostImage extends DatabaseObject
    {
        // ... other code

        public static function GetImages($db, $options = array())
        {
            // initialize the options
            $defaults = array('post_id' => array());

            foreach ($defaults as $k => $v) {
                $options[$k] = array_key_exists($k, $options) ? $options[$k] : $v;
            }

            $select = $db->select();
            $select->from(array('i' => 'blog_posts_images'), array('i.*'));

            // filter results on specified post ids (if any)
            if (count($options['post_id']) > 0)
                $select->where('i.post_id in (?)', $options['post_id']);

            $select->order('i.ranking');
            // fetch post data from database
            $data = $db->fetchAll($select);

            // turn data into array of DatabaseObject_BlogPostImage objects
            $images = parent::BuildMultiple($db, __CLASS__, $data);

            return $images;
        }
    }
?>
```

Displaying Images on the Post Preview

The next step in managing images for a blog post is to display them on the preview page. To do this, we must make some changes to the `preview.tpl` template in the `./templates/blogmanager` directory, as well as adding some new styles to `./htdocs/css/styles.css`.

Earlier in this chapter we created a new element in this template called `#preview-images`. The code in Listing 11-32 shows the additions we must make to `preview.tpl` to display each of the images. We will output the images in an unordered list, which will help us later when we add the ability to reorder the images using Scriptaculous.

Listing 11-32. *Outputting Images on the Blog Post Preview Page (preview.tpl)*

```
<!-- // ... other code -->

<fieldset id="preview-images">
    <legend>Images</legend>

    {if $post->images|@count > 0}
        <ul id="post_images">
            {foreach from=$post->images item=image}
                <li id="image_{$image->getId()}">
                    <img src="{imagefilename id=$image->getId() w=200 h=65}"
                         alt="{$image->filename|escape}" />

                    <form method="post" action="{geturl action='images'}">
                        <div>
                            <input type="hidden"
                                   name="id" value="{$post->getId()}" />
                            <input type="hidden"
                                   name="image" value="{$image->getId()}" />
                            <input type="submit" value="Delete" name="delete" />
                        </div>
                    </form>
                </li>
            {/foreach}
        </ul>
    {/if}
    <form method="post"
          action="{geturl action='images'}"
          enctype="multipart/form-data">

        <div>
            <input type="hidden" name="id" value="{$post->getId()}" />
            <input type="file" name="image" />
            <input type="submit" value="Upload Image" name="upload" />
        </div>
    </form>
</fieldset>

<!-- // ... other code -->
```

As you can see in the code, we use the new `imagefilename` plug-in to generate the URL for an image thumbnail 200 pixels wide and 65 pixels high. We also include a form to delete each image in this template. We haven't yet implemented this functionality (you may recall that we left a placeholder for the delete command in the blog manager's `imagesAction()` method), but this will be added shortly.

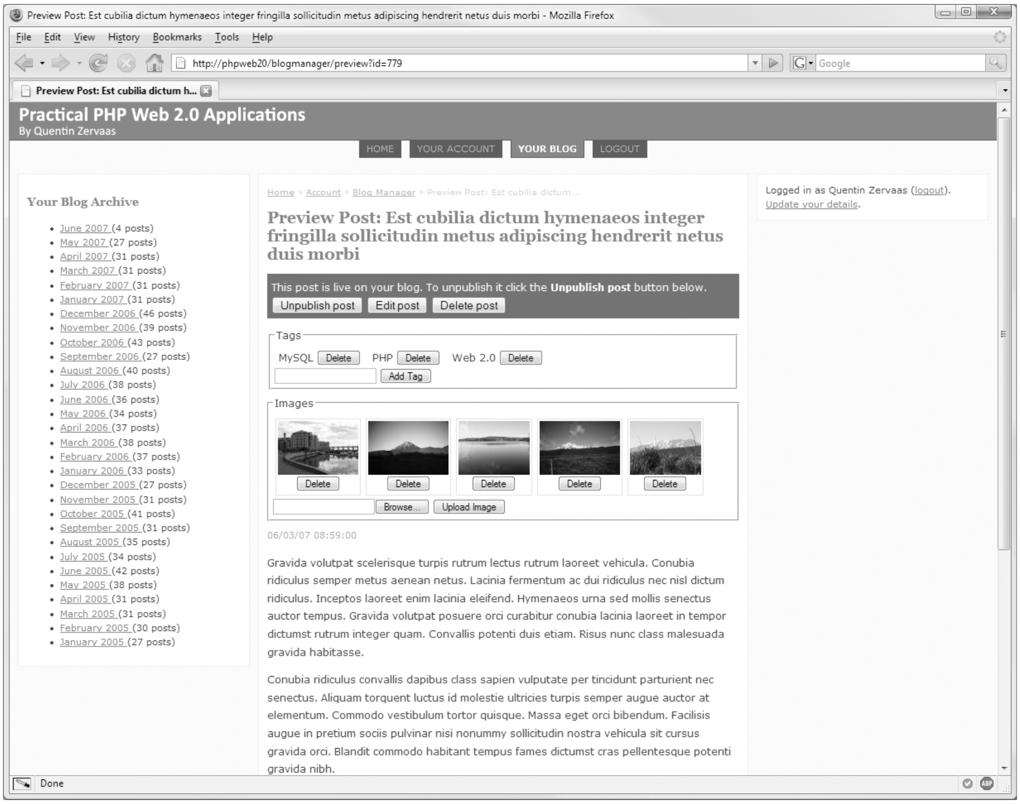
Listing 11-33 shows the new styles we will add to `styles.css` in `./htdocs/css`. These styles format the unordered list so list items are shown horizontally. We use floats to position list items next to each other (rather than using `inline` display), since this gives greater control over the style within each item. Note that we must add `clear : both` to the div holding the upload form in order to keep the display of the page intact.

Listing 11-33. *Styling the Image-Management Area (styles.css)*

```
#preview-images ul {
    list-style-type : none;
    margin          : 0;
    padding         : 0;
}

#preview-images li {
    float           : left;
    font-size       : 0.85em;
    text-align      : center;
    margin          : 3px;
    padding         : 2px;
    border          : 1px solid #ddd;
    background      : #fff;
}

#preview-images img {
    display : block;
}
```

```
#preview-images div {
    clear : both;
}
```

Once this code has been added, the image display area should look like the page in Figure 11-2.



Preview Post: Est cubilia dictum hymenaeos integer fringilla sollicitudin metus adipiscing hendrerit netus duis morbi - Mozilla Firefox

http://phpweb20/blogmanager/preview?id=779

Practical PHP Web 2.0 Applications
By Quentin Zervaas

Your Blog Archive

- June 2007 (4 posts)
- May 2007 (27 posts)
- April 2007 (31 posts)
- March 2007 (31 posts)
- February 2007 (31 posts)
- January 2007 (31 posts)
- December 2006 (46 posts)
- November 2006 (39 posts)
- October 2006 (43 posts)
- September 2006 (27 posts)
- August 2006 (40 posts)
- July 2006 (38 posts)
- June 2006 (36 posts)
- May 2006 (34 posts)
- April 2006 (37 posts)
- March 2006 (38 posts)
- February 2006 (37 posts)
- January 2006 (33 posts)
- December 2005 (27 posts)
- November 2005 (31 posts)
- October 2005 (41 posts)
- September 2005 (31 posts)
- August 2005 (35 posts)
- July 2005 (34 posts)
- June 2005 (42 posts)
- May 2005 (38 posts)
- April 2005 (31 posts)
- March 2005 (31 posts)
- February 2005 (30 posts)
- January 2005 (27 posts)

Home » Account » Blog Manager » Preview Post: Est cubilia dictum...

Preview Post: Est cubilia dictum hymenaeos integer fringilla sollicitudin metus adipiscing hendrerit netus duis morbi

This post is live on your blog. To unpublish it click the **Unpublish post** button below.

06/03/07 08:59:00

Gravida volutpat scelerisque turpis rutrum lectus rutrum laoreet vehicula. Conubia ridiculus semper metus aenean netus. Lacinia fermentum ac dui ridiculus nec nisl dictum ridiculus. Inceptos laoreet enim lacinia eleifend. Hymenaeos urna sed mollis senectus auctor tempus. Gravida volutpat posuere orci curabitur conubia lacinia laoreet in tempor dictumst rutrum integer quam. Convallis potenti duis etiam. Risus nunc class malesuada gravida habitasse.

Conubia ridiculus convallis dapibus class sapien vulputate per tincidunt parturient nec senectus. Aliquam torquent luctus id molestie ultricies turpis semper augue auctor at elementum. Commodo vestibulum tortor quisque. Massa eget orci bibendum. Facilisis augue in pretium sociis pulvinar nisi nonummy sollicitudin nostra vehicula sit cursus gravida orci. Blandit commodo habitant tempus fames dictumst cras pellentesque potenti gravida nibh.

Logged in as Quentin Zervaas (logout).
Update your details.

Figure 11-2. *Displaying the images on the blog post preview page*

Deleting Blog Post Images

The next step in the management of blog post images is to implement the delete functionality. We will first implement a non-Ajax version to delete images, and then modify it slightly to use Scriptaculous for a fancier solution.

Before we complete the delete section of the images action in the blog manager controller, we must make some small changes to the `DatabaseObject_BlogPostImage` class. Using `DatabaseObject` means we can simply call the `delete()` method on the image record to remove it from the database, but this will not delete the uploaded image from the filesystem. As we saw in Chapter 3, if we define the `postDelete()` method in a `DatabaseObject` subclass, it is automatically called after a record has been deleted. We will implement this method for `DatabaseObject_BlogPostImage` so the uploaded file is removed from the filesystem.

Additionally, since thumbnails are automatically created for each image, we will clean up the thumbnail storage area for the image being deleted. Note that this is quite easy, since we prefixed all generated thumbnails with their database ID.

Listing 11-34 shows the `postDelete()` function as it should be added to `DatabaseObject_BlogPostImage` in `./include/DatabaseObject`. First, we use `unlink()` to delete the main image from the filesystem. Next, we use the `glob()` function, which is a useful PHP function for retrieving an array of files based on the specified pattern. We loop over each of the files in the array and `unlink()` them.

Listing 11-34. *Deleting the Uploaded File and All Generated Thumbnails (BlogPostImage.php)*

```
<?php
    class DatabaseObject_BlogPostImage extends DatabaseObject
    {
        // ... other code

        public function preDelete()
        {
            unlink($this->getFullPath());

            $pattern = sprintf('%s/%d.*',
                               self::GetThumbnailPath(),
                               $this->getId());

            foreach (glob($pattern) as $thumbnail) {
                unlink($thumbnail);
            }

            return true;
        }

        // ... other code
    }
?>
```

Now when you call the `delete()` method on a loaded blog post image, the filesystem files will also be deleted. Remember to return `true` from `postDelete()`—otherwise the SQL transaction will be rolled back.

The other method we must add to this class is one that gives us the ability to load an image for a specified blog post. This is similar to the `loadForUser()` function we implemented for blog posts. We do this so that only the logged-in user will be able to delete an image on their blog posts. Listing 11-35 shows the code for the `loadForPost()` function, which is also added to `BlogPostImage.php`.

Listing 11-35. *Restricting the Load of Images to a Particular Blog Post (BlogPostImage.php)*

```
<?php
    class DatabaseObject_BlogPostImage extends DatabaseObject
    {
        // ... other code

        public function loadForPost($post_id, $image_id)
        {
            $post_id = (int) $post_id;
            $image_id = (int) $image_id;

            if ($post_id <= 0 || $image_id <= 0)
                return false;

            $query = sprintf(
                'select %s from %s where post_id = %d and image_id = %d',
                join(', ', $this->getSelectFields()),
                $this->_table,
                $post_id,
                $image_id
            );

            return $this->_load($query);
        }

        // ... other code
    }
?>
```

Now that these changes have been made to `DatabaseObject_BlogPostImage`, we can implement the non-Ajax version of deleting an image. To do this, we simply need to implement the delete part of `imagesAction()` in `BlogmanagerController.php`. Remember that we left a placeholder for this when we originally created this method in Listing 11-5. The code used to delete an image is shown in Listing 11-36.

Listing 11-36. *Deleting an Image from a Blog Post (BlogmanagerController.php)*

```
<?php
    class BlogmanagerController extends CustomControllerAction
    {
        // ... other code

        public function imagesAction()
        {
            // ... other code

            else if ($request->getPost('delete')) {
                $image_id = (int) $request->getPost('image');
                $image = new DatabaseObject_BlogPostImage($this->db);
                if ($image->loadForPost($post->getId(), $image_id)) {
                    $image->delete();
                    $this->messenger->addMessage('Image deleted');
                }
            }

            // ... other code
        }
    }
?>
```

If you now click on the "Delete" button below an image, the image will be deleted from the database and filesystem, and a message will appear in the top-right flash messenger when the page reloads.

Using Scriptaculous and Ajax to Delete Images

Now that we have a non-Ajax solution for deleting images, we can enhance this system slightly to use Ajax. Essentially what we will do is send an Ajax request to delete the image when the "Delete" button is clicked, and use Scriptaculous to make the image disappear from the screen.

There are a number of different Scriptaculous effects that can be used to hide elements, such as `Puff`, `SwitchOff`, `DropOut`, `Squish`, `Fold`, and `Shrink`, but we are going to use the `Fade` effect. Note, however, that we are not applying this effect to the image being deleted; we will apply it to the list item (`<li>`) surrounding the image.

Modifying the PHP Deletion Code

In the `imagesAction()` function of `BlogmanagerController.php`, the code redirects the browser back to the blog post preview page after completing the action (uploading, reordering, or deleting). This is fine for non-Ajax solutions, but if this occurs when using `XMLHttpRequest`, the contents of the preview page will unnecessarily be returned in the background.

To prevent this, we will make a simple change to the redirection code at the end of this function. As we have done previously, we will use the `isXmlHttpRequest()` function provided by `Zend_Controller_Front` to determine how to proceed.

Because we want to check whether or not the image deletion was successful in the JavaScript code, we will also modify the code so it sends back JSON data about the deleted image. We will send this back using the `sendJson()` method we added in Chapter 6.

Listing 11-37 shows the changes to this method in `BlogmanagerController.php`. This code now only writes the deletion message to the messenger if the delete request did not use Ajax. If this distinction about writing the message isn't made, you could delete an image via Ajax and then refresh the page, causing the "image deleted" message to show again.

Listing 11-37. *Handling Ajax Requests in imageAction() (BlogmanagerController.php)*

```
<?php
    class BlogmanagerController extends CustomControllerAction
    {
        // ... other code
        public function imagesAction()
        {
            // ... other code

            $json = array();

            // ... other code
            if ($request->getPost('upload')) {
                // ... other code
            }
            else if ($request->getPost('reorder')) {
                // ... other code
            }
            else if ($request->getPost('delete')) {
                $image_id = (int) $request->getPost('image');
                $image = new DatabaseObject_BlogPostImage($this->db);
                if ($image->loadForPost($post->getId(), $image_id)) {
                    $image->delete();
                    if ($request->isXmlHttpRequest()) {
                        $json = array(
                            'deleted'  => true,
                            'image_id' => $image_id
                        );
                    }
                    else
                        $this->messenger->addMessage('Image deleted');
                }
            }

            if ($request->isXmlHttpRequest()) {
                $this->sendJson($json);
            }
            else {
                $url = $this->getUrl('preview') . '?id=' . $post->getid();
                $this->_redirect($url);
            }
        }
    }
?>
```

Creating the BlogImageManager JavaScript Class

To create an Ajax solution for deleting blog post images, we will write a new JavaScript class called `BlogImageManager`. This class will find all of the delete forms in the image-management section of `preview.tpl` and bind the `submit` event listener to each of these forms. We will then implement a function to handle this event.

Listing 11-38 shows the constructor for this class, which we will store in a file called `BlogImageManager.class.js` in the `./htdocs/js` directory.

Listing 11-38. *The Constructor for BlogImageManager (BlogImageManager.class.js)*

```
BlogImageManager = Class.create();

BlogImageManager.prototype = {

    initialize : function(container)
    {
        this.container = $(container);

        if (!this.container)
            return;

        this.container.getElementsBySelector('form').each(function(form) {
            form.observe('submit',
                         this.onDeleteClick.bindAsEventListener(this));
        }.bind(this));
    },
```

This class expects the unordered list element that holds the images as the only argument to the constructor. We store it as a property of the object, since we will be using it again later when implementing the reordering functionality.

In this class, we find all the forms within this unordered list by using the `getElementsBySelector()` function. This function behaves in the same way as the `$$()` function we looked at in Chapter 5, except that it only searches within the element the function is being called from.

We then loop over each form that is found and observe the `submit` event on it. We must bind the `onDeleteClick()` event handler to the `BlogImageManager` instance so it can be referred to within the correct context when the event is handled.

The next thing we need to do is implement the `onDeleteClick()` event handler, as shown in Listing 11-39.

Listing 11-39. *The Event Handler Called When a Delete Link Is Clicked (BlogImageManager.class.js)*

```
    onDeleteClick : function(e)
    {
        Event.stop(e);
        var form = Event.element(e);

        var options = {
            method     : form.method,
            parameters : form.serialize(),
            onSuccess  : this.onDeleteSuccess.bind(this),
            onFailure  : this.onDeleteFailure.bind(this)
        }

        message_write('Deleting image...');
        new Ajax.Request(form.action, options);
    },
```

The first thing we do in this method is stop the event so the browser doesn't submit the form normally—a background Ajax request will be submitting the form instead.

Next, we determine which form was submitted by calling `Event.element()`. This allows us to perform an Ajax request on the form action URL, thereby executing the PHP code that is used to delete a blog post image.

We then create a hash of options to pass to `Ajax.Request()`, which includes the form values and the callback handlers for the request. Before instantiating `Ajax.Request()`, we update the page status message to tell the user that an image is being deleted.

The next step is to implement the handlers for a successful and unsuccessful request, as shown in Listing 11-40.

Listing 11-40. *Handling the Response from the Ajax Image Deletion (BlogImageManager.class.js)*

```
    onDeleteSuccess : function(transport)
    {
        var json = transport.responseText.evalJSON(true);

        if (json.deleted) {
            var image_id = json.image_id;

            var input = this.container.down('input[value=' + image_id + ']');
            if (input) {
                var options = {
                    duration    : 0.3,
                    afterFinish : function(effect) {
                        message_clear();
                        effect.element.remove();
                    }
                }

                new Effect.Fade(input.up('li'), options);
                return;
            }
        }

        this.onDeleteFailure(transport);
    },

    onDeleteFailure : function(transport)
    {
        message_write('Error deleting image');
    }
};
```

In Listing 11-37 we made the delete operation in `imagesAction()` return JSON data. To determine whether the image was deleted by the code in Listing 11-40, we check for the `deleted` element in the decoded JSON data.

Based on the image_id element also included in the JSON data, we try to find the corresponding form element on the page for that image. We do this by looking for a form input with the value of the image ID. Once we find this element, we apply the Scriptaculous fade effect to make the image disappear from the page. We don't apply this effect to the actual image that was deleted; rather, we remove the surrounding list item so the image, form, and surrounding code are completely removed from the page.

When the fade effect is called, the element being faded is only *hidden* when the effect is completed; it is not actually removed from the DOM. In order to remove it, we define the afterFinish callback on the effect, and use it to call the remove() method on the element. The callbacks for Scriptaculous effects receive the effect object as the first argument, and the element the effect is applied to can be accessed using the element property of the effect. We also use the afterFinish function to clear the status message.

After we've defined the options, we can create the actual effect. Since we want to remove the list item element corresponding to the image, we can simply call the Prototype up() function to find it.

Loading BlogImageManager in the Post Preview

Next, we will load the BlogImageManager JavaScript class in the preview.tpl template. In order to instantiate this class, we will add code to the blogPreview.js file we created in Chapter 7.

Listing 11-41 shows the changes we will make to preview.tpl in the ./templates/blogmanager directory to load BlogImageManager.class.js.

Listing 11-41. *Loading the BlogImageManager Class (preview.tpl)*

```
{include file='header.tpl' section='blogmanager'}

<script type="text/javascript" src="/js/blogPreview.js"></script>
<script type="text/javascript" src="/js/BlogImageManager.class.js"></script>

<!-- // ... other code -->
```

Listing 11-42 shows the changes we will make to blogPreview.js in ./htdocs/js to instantiate BlogImageManager automatically.

Listing 11-42. *Instantiating BlogImageManager Automatically (blogPreview.js)*

```
Event.observe(window, 'load', function() {

    // ... other code

    var im = new BlogImageManager('post_images');
});
```

If you now try to delete an image from a blog post, the entire process should be completed in the background. Once the "Delete" button is clicked, the background request to delete the image will be initiated, and the image will disappear from the page upon successful completion.

Deleting Images when Posts Are Deleted

One thing we have not yet dealt with is what happens to images when a blog post is deleted. As the code currently stands, if a blog post is deleted, any associated images will not be deleted. Because of the foreign key constraint on the `blog_posts_images` table, the SQL to delete a blog post that has one or more images will fail. We must update the `DatabaseObject_BlogPost` class so images are deleted when a post is deleted.

Doing this is very straightforward, since the instance of `DatabaseObject_BlogPost` we are trying to delete already has all the images loaded (so we know exactly what needs to be deleted), and it already has a delete callback (we implemented the `preDelete()` function earlier). This means we can simply loop over each image and call the `delete()` method.

■**Note** `DatabaseObject` automatically controls transactions when saving or deleting a record. You can pass `false` to `save()` or `delete()` so transactions are not used. Because a transaction has already been started by the `delete()` call on the blog post, we must pass `false` to the `delete()` call for each image.

Listing 11-43 shows the two new lines we need to add to `preDelete()` in the `BlogPost.php` file in the `./include/DatabaseObject` directory.

Listing 11-43. *Automatically Deleting Images When a Blog Post Is Deleted (BlogPost.php)*

```
<?php
    class DatabaseObject_BlogPost extends DatabaseObject
    {
        // ... other code

        protected function preDelete()
        {
            // ... other code

            foreach ($this->images as $image)
                $image->delete(false);

            return true;
        }

        // ... other code
    }
?>
```

Now when you try to delete a blog post, all images associated with the post will also be deleted.

Reordering Blog Post Images

We will now implement a system that will allow users to change the order of the images associated with a blog post. While this may not seem overly important, we do this because we are controlling the layout of images when blog posts are displayed.

Additionally, in the next section we will modify the blog index to display an image beside each blog post that has one. If a blog post has more than one image, we will use the first image for the post.

Drag and Drop

In the past, programmers have used two common techniques to allow users to change the order of list items, both of which are slow and difficult to use.

The first method was to provide “up” and “down” links beside each item in the list, which moved the items up or down when clicked. Some of these implementations might have included a “move to top” and “move to bottom” button, but on the whole they are difficult to use.

The other method was to provide a text input box beside each item. Each box contained a number, which determined the order of the list. To change the order, you would update the numbers inside the boxes.

For our implementation, we will use a drag-and-drop system. Thanks to Scriptaculous’s `Sortable` class, this is not difficult to achieve. We will implement this by extending the `BlogImageManager` JavaScript class we created earlier this chapter.

■ **Note** As an exercise, try extending this reordering system so it is accessible for non-JavaScript users. You could try implementing this by including a form on the page within `<noscript>` tags (meaning it won’t be shown to users who have JavaScript enabled).

Saving the Order to Database

Before we add the required JavaScript to the blog post management page, we will write the PHP for saving the image order to the database. First, we need to add a new function to the `DatabaseObject_BlogPost` class. This function accepts an array of image IDs as its only argument. The order in which each image ID appears in the array is the order it will be saved in.

Listing 11-44 shows the `setImageOrder()` function that we will add to the `BlogPost.php` file in `./include/DatabaseObject`. Before updating the database, it loops over the values passed to it and sanitizes the data by ensuring each of the values belongs to the `$images` property of the object. After cleaning the data, it checks that the number of image IDs found in the array matches the number of images in the post. Only then does it proceed to update the database.

Listing 11-44. *Saving the Updated Image Order in the Database (BlogPost.php)*

```
<?php
    class DatabaseObject_BlogPost extends DatabaseObject
    {
        // ... other code

        public function setImageOrder($order)
        {
            // sanitize the image IDs
            if (!is_array($order))
                return;

            $newOrder = array();
            foreach ($order as $image_id) {
                if (array_key_exists($image_id, $this->images))
                    $newOrder[] = $image_id;
            }

            // ensure the correct number of IDs were passed in
            $newOrder = array_unique($newOrder);
            if (count($newOrder) != count($this->images)) {
                return;
            }

            // now update the database
            $rank = 1;
            foreach ($newOrder as $image_id) {
                $this->_db->update('blog_posts_images',
                                   array('ranking' => $rank),
                                   'image_id = ' . $image_id);

                $rank++;
            }
        }

        // ... other code
    }
?>
```

In order to use this function, we must update the `imagesAction()` function in `BlogmanagerController.php` (in `./include/Controllers`). Listing 11-45 shows the code we will use to call the `setImageOrder()` method in Listing 11-44. After calling this method, the code will fall through to the `isXmlHttpRequest()` call, thereby returning the empty JSON data. The submitted variable that holds the image order is called `post_images`. Scriptaculous uses the ID of the draggable DOM element as the form value, as we will see shortly.

Listing 11-45. *Handling the Reorder Action in the Action Handler (BlogManagerController.php)*

```
<?php
    class BlogmanagerController extends CustomControllerAction
    {
        // ... other code

        public function imagesAction()
        {
            // ... other code

            else if ($request->getPost('reorder')) {
                $order = $request->getPost('post_images');
                $post->setImageOrder($order);
            }

            // ... other code
        }
    }
?>
```

Adding Sortable to BlogImageManager

It is fairly straightforward to add Sortable to our unordered list; however, we must also add some Ajax functionality to the code. When a user finishes dragging an image, we need to initiate an Ajax request that sends the updated image order to the server so the setImageOrder() function (in Listing 11-44) can be called.

Sortable allows us to define a parameter called onUpdate, which specifies a callback function that is called after the image order has been changed. The callback function we create will initiate the Ajax request. Before we get to that, though, let's look at creating the Sortable list.

By default, Sortable operates an unordered list. It is possible to allow other types of elements to be dragged (although there may be some incompatibility with dragging table cells), but since we are using an unordered list we don't need to specify the type of list.

Another default that Sortable sets is for the list to be vertical. This means the dragging direction for items is up and down. Since our list is horizontal, we need to change this setting by specifying the constraint parameter. We could set this value to horizontal, but since the list of images for a single post may span multiple rows (such as on a low-resolution monitor) it would not be possible to drag images on the second row to the first (and vice versa). To deal with this, we simply set constraint to be false.

Since our list is horizontal, we must change the overlap value to be horizontal instead of its default of vertical. Sortable uses this value to determine how to calculate when an item has been dragged to a new location.

Listing 11-46 shows the code we must add to the constructor of the BlogImageManager JavaScript class in ./htdocs/js/BlogImageManager.class.js. Note that this code uses the onSortUpdate() function, which we have not yet defined.

Listing 11-46. *Creating the Sortable list (BlogImageManager.class.js)*

```
BlogImageManager = Class.create();

BlogImageManager.prototype = {

    initialize : function(container)
    {
        // ... other code

        var options = {
            overlap    : 'horizontal',
            constraint : false,
            onUpdate   : this.onSortUpdate.bind(this)
        };

        Sortable.create(this.container, options);
    },

    // ... other code
};
```

Now we must define the `onSortUpdate()` callback function. This is called when an item in the sortable list is dropped into a new location. In this function we initiate a new Ajax request that sends the order of the list to the `imagesAction()` function. `Sortable` will pass the container element of the sortable list to this callback.

When sending this request, we must send the updated order. We can retrieve this order using the `Sortable` utility function `serialize()`, which retrieves all values and builds them into a URL-friendly string that we can post. As mentioned previously, the unordered list we've made sortable has an ID of `post_images`. This means that if we have three images with IDs of 5, 6, and 7, calling `Sortable.serialize()` will generate a string such as this:

```
post_images[]=5&post_images[]=6&post_images[]=7
```

PHP will automatically turn this into an array. In other words, the equivalent PHP code to create this structure would be as follows:

```
<?php
    $post_images = array(5, 6, 7);
?>
```

This is exactly what we need in `setImageOrder()`.

Listing 11-47 shows the code for `onSortUpdate()`, as described above. Another thing we do in this code is to update the status message on the page to notify the user that the order is being saved. In addition, we define the `onSuccess()` callback, which we will use to clear the status message once the new order has been saved.

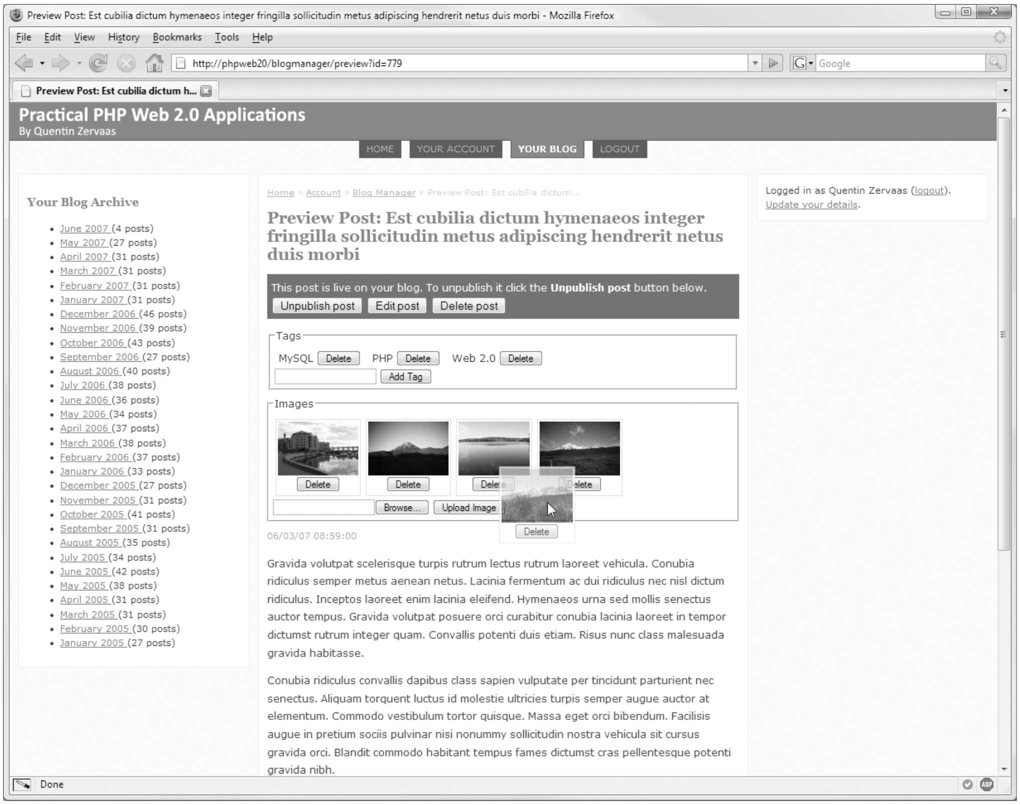
Listing 11-47. *The Callback Function That Is Called after the List Order Has Changed (BlogImageManager.class.js)*

```
BlogImageManager = Class.create();

BlogImageManager.prototype = {

    // ... other code

    onSortUpdate : function(draggable)
    {
        var form = this.container.down('form');
        var post_id = $F(form.down('input[name=id]'));

        var options = {
            method     : form.method,
            parameters : 'reorder=1'
                       + '&id=' + post_id
                       + '&' + Sortable.serialize(draggable),
            onSuccess  : function() { message_clear(); }
        };

        message_write('Updating image order...');
        new Ajax.Request(form.action, options);
    }
};
```

■**Note** When you add this code to your existing class, remember to include a comma at the end of the previous function in the class (`onDeleteFailure()`). Unfortunately, this is one of the pitfalls of writing classes using Prototype: each method is really an element in its class's `prototype` hash, and therefore needs to be comma-separated.

Based on how the HTML is structured for the image-management area on the blog preview page, there is no simple way to define the URL for where image-reordering requests should be sent. Since all of our image operations use the same controller action, we will determine the URL by finding the form action of any form in the image-management area. We will also expect the form being used to have an element called `post_id` that holds the ID of the blog post.

If you now view the blog post preview page (with multiple images assigned to the post you are viewing), you will be able to click on an image and drag it to a new location within the list of images. Figure 11-3 shows how this might look.


Preview Post: Est cubilia dictum hymenaeos integer fringilla sollicitudin metus adipiscing hendrerit netus duis morbi - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://phpweb20/blogmanager/preview?id=779
Google
Preview Post: Est cubilia dictum h...
HOME
YOUR ACCOUNT
YOUR BLOG
LOGOUT
Unpublish post
Edit post
Delete post
Tags
MySQL Delete
PHP Delete
Web 2.0 Delete
Add Tag
Images
Delete
Browse...
Upload Image
Done

Practical PHP Web 2.0 Applications
By Quentin Zervaas

Your Blog Archive

- June 2007 (4 posts)
- May 2007 (27 posts)
- April 2007 (31 posts)
- March 2007 (31 posts)
- February 2007 (31 posts)
- January 2007 (31 posts)
- December 2006 (46 posts)
- November 2006 (39 posts)
- October 2006 (43 posts)
- September 2006 (27 posts)
- August 2006 (40 posts)
- July 2006 (38 posts)
- June 2006 (36 posts)
- May 2006 (34 posts)
- April 2006 (37 posts)
- March 2006 (38 posts)
- February 2006 (37 posts)
- January 2006 (33 posts)
- December 2005 (27 posts)
- November 2005 (31 posts)
- October 2005 (41 posts)
- September 2005 (31 posts)
- August 2005 (35 posts)
- July 2005 (34 posts)
- June 2005 (42 posts)
- May 2005 (38 posts)
- April 2005 (31 posts)
- March 2005 (31 posts)
- February 2005 (30 posts)
- January 2005 (27 posts)

Home » Account » Blog Manager » Preview Post: Est cubilia dictum...

Preview Post: Est cubilia dictum hymenaeos integer fringilla sollicitudin metus adipiscing hendrerit netus duis morbi

This post is live on your blog. To unpublish it click the **Unpublish post** button below.

06/03/07 08:59:00

Gravida volutpat scelerisque turpis rutrum lectus rutrum laoreet vehicula. Conubia ridiculus semper metus aenean netus. Lacinia fermentum ac dui ridiculus nec nisl dictum ridiculus. Inceptos laoreet enim lacinia eleifend. Hymenaeos urna sed mollis senectus auctor tempus. Gravida volutpat posuere orci curabitur conubia lacinia laoreet in tempor dictumst rutrum integer quam. Convallis potenti duis etiam. Risus nunc class malesuada gravida habitasse.

Conubia ridiculus convallis dapibus class sapien vulputate per tincidunt parturient nec senectus. Aliquam torquent luctus id molestie ultricies turpis semper augue auctor at elementum. Commodo vestibulum tortor quisque. Massa eget orci bibendum. Facilisis augue in pretium sociis pulvinar nisi nonummy sollicitudin nostra vehicula sit cursus gravida orci. Blandit commodo habitant tempus fames dictumst cras pellentesque potenti gravida nibh.

Logged in as Quentin Zervaas (logout).
Update your details.

Figure 11-3. *Changing the order of blog post images by dragging and dropping*

Displaying Images on User Blogs

The final thing we need to do to create a dynamic image gallery for users is to make use of the images they have uploaded and sorted. To do this, we must display the images both on the blog posts they belong to as well as in the blog index.

When displaying images on a post page, we will show all images (in their specified order) with the ability to view a full-size version of each. On the index page we will only show a small thumbnail of the first image.

Extending the GetPosts() Function

When we added the image-loading functionality to the `DatabaseObject_BlogPost` class in Listing 11-30, we didn't add the same functionality to the `GetPosts()` function within this class. If you recall, `GetPosts()` is used to retrieve multiple blog posts from the database at one time.

We must now make this change to `GetPosts()` so we display images on each user's blog index. We can use the `GetImages()` function in `DatabaseObject_BlogPostImage` to retrieve all images for the loaded blog posts, and then simply loop over the returned images and write them to the corresponding post.

The new code to be inserted at the end of GetPosts() in BlogPost.php is shown in Listing 11-48. Note that the $post_ids array is initialized earlier in the function.

Listing 11-48. *Modifying DatabaseObject_BlogPost to Load Post Images (BlogPost.php)*

```
<?php
    class DatabaseObject_BlogPost extends DatabaseObject
    {
        // ... other code

        public static function GetPosts($db, $options = array())
        {
            // ... other code

            // load the images for each post
            $options = array('post_id' => $post_ids);
            $images = DatabaseObject_BlogPostImage::GetImages($db, $options);

            foreach ($images as $image) {
                $posts[$image->post_id]->images[$image->getId()] = $image;
            }

            return $posts;
        }

        // ... other code
    }
?>
```

Because of this change, all controller actions that call this method now automatically have access to each image, meaning we now only need to change the output templates.

Displaying Thumbnail Images on the Blog Index

The other thing we have done during the development of the code in this book is to output all blog post teasers using the blog-post-summary.tpl template. This means that in order to add a thumbnail to the output of the blog post index (be it the user's home page or the monthly archive) we just need to add an <img> tag to this template.

Listing 11-49 shows the additions we will make to blog-post-summary.tpl in ./templates/user/lib. After checking that the post has one or more images, we will use the PHP current() function to retrieve the first image. Remember that we must precede this with @ in Smarty so current() is applied to the array as a whole and not to each individual element.

Listing 11-49. *Displaying the First Image for Each Post on the Blog Index (blog-post-summary.tpl)*

```
<div class="teaser">
    <!-- // ... other code -->

    <div class="teaser-date">
        <!-- // ... other code -->
    </div>

    {if $post->images|@count > 0}
        {assign var=image value=$post->images|@current}
        <div class="teaser-image">
            <a href="{$url|escape}">
                <img src="{imagefilename id=$image->getId() w=100}" alt="" />
            </a>
        </div>
    {/if}

    <!-- // ... other code -->
</div>
```

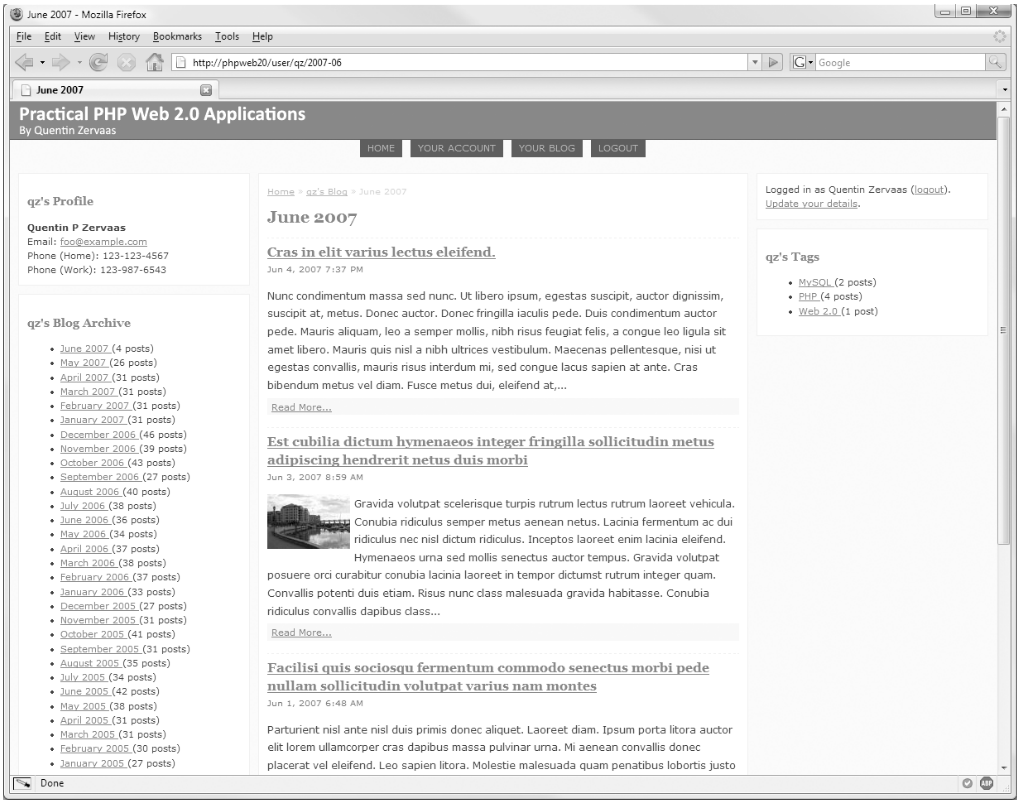
We must also add some style to this page so the output is clean. To do this, we will float the `.teaser-image` div to the left. The only problem with this is that the image may overlap the post footer (which displays the number of submitted comments). To fix this, we will also add `clear : both` to the `.teaser-links` class.

Listing 11-50 shows the changes to the `styles.css` file in `./htdocs/css`.

Listing 11-50. *Styling the Blog Post Image (styles.css)*

```
/* ... other code */

    .teaser-links {
        /* ... other code */
    }

    .teaser-image {
        float       : left;
        margin      : 0 5px 5px 0;
    }

/* ... other code */
```

Once you have added these styles, your blog index page should look similar the one in Figure 11-4.



June 2007 - Mozilla Firefox

http://phpweb20/user/qz/2007-06

June 2007

Practical PHP Web 2.0 Applications
By Quentin Zervaas

HOME YOUR ACCOUNT YOUR BLOG LOGOUT

qz's Profile

Quentin P Zervaas
Email: foo@example.com
Phone (Home): 123-123-4567
Phone (Work): 123-987-6543

qz's Blog Archive

- June 2007 (4 posts)
- May 2007 (26 posts)
- April 2007 (31 posts)
- March 2007 (31 posts)
- February 2007 (31 posts)
- January 2007 (31 posts)
- December 2006 (46 posts)
- November 2006 (39 posts)
- October 2006 (43 posts)
- September 2006 (27 posts)
- August 2006 (40 posts)
- July 2006 (38 posts)
- June 2006 (36 posts)
- May 2006 (34 posts)
- April 2006 (37 posts)
- March 2006 (38 posts)
- February 2006 (37 posts)
- January 2006 (33 posts)
- December 2005 (27 posts)
- November 2005 (31 posts)
- October 2005 (41 posts)
- September 2005 (31 posts)
- August 2005 (35 posts)
- July 2005 (34 posts)
- June 2005 (42 posts)
- May 2005 (38 posts)
- April 2005 (31 posts)
- March 2005 (31 posts)
- February 2005 (30 posts)
- January 2005 (27 posts)

Home » qz's Blog » June 2007

June 2007

Cras in elit varius lectus eleifend.
Jun 4, 2007 7:37 PM

Nunc condimentum massa sed nunc. Ut libero ipsum, egestas suscipit, auctor dignissim, suscipit at, metus. Donec auctor. Donec fringilla iaculis pede. Duis condimentum auctor pede. Mauris aliquam, leo a semper mollis, nibh risus feugiat felis, a congue leo ligula sit amet libero. Mauris quis nisl a nibh ultrices vestibulum. Maecenas pellentesque, nisi ut egestas convallis, mauris risus interdum mi, sed congue lacus sapien at ante. Cras bibendum metus vel diam. Fusce metus dui, eleifend at,...

Read More...

Est cubilia dictum hymenaeos integer fringilla sollicitudin metus adipiscing hendrerit netus duis morbi
Jun 3, 2007 8:59 AM

Gravida volutpat scelerisque turpis rutrum lectus rutrum laoreet vehicula. Conubia ridiculus semper metus aenean netus. Lacinia fermentum ac dui ridiculus nec nisl dictum ridiculus. Inceptos laoreet enim lacinia eleifend. Hymenaeos urna sed mollis senectus auctor tempus. Gravida volutpat posuere orci curabitur conubia lacinia laoreet in tempor dictumst rutrum integer quam. Convallis potenti duis etiam. Risus nunc class malesuada gravida habitasse. Conubia ridiculus convallis dapibus class...

Read More...

Facilisi quis sociosqu fermentum commodo senectus morbi pede nullam sollicitudin volutpat varius nam montes
Jun 1, 2007 6:48 AM

Parturient nisl ante nisl duis primis donec aliquet. Laoreet diam. Ipsum porta litora auctor elit lorem ullamcorper cras dapibus massa pulvinar urna. Mi aenean convallis donec placerat vel eleifend. Leo sapien litora. Molestie malesuada quam penatibus lobortis justo

Logged in as Quentin Zervaas (logout).
Update your details.

qz's Tags

- MySQL (2 posts)
- PHP (4 posts)
- Web 2.0 (1 post)

Done

Figure 11-4. *The blog index page displaying the first image for posts that have images*

Displaying Images on the Blog Details Page

The final change we must make to our templates is to display each of the images for a blog post when viewing the blog post details page. This will behave similarly to the blog post preview page, except that we will also allow users to view a larger version of each image. To improve the output of the larger version of each image, we will use a simple little script called Lightbox.

First, we must alter the `view.tpl` template in the `./templates/user` directory. This is the template responsible for displaying blog post details. We will make each image appear vertically on the right side of the blog by floating the images to the right. This means we must include them in the HTML output before the blog content, as shown in Listing 11-51.

Listing 11-51. *Displaying Each of the Post's Images (view.tpl)*

```
<!-- // ... other code -->

<div class="post-date">
    <!-- // ... other code -->
</div>

{foreach from=$post->images item=image}
    <div class="post-image">
        <a href="{imagefilename id=$image->getId() w=600}">
            <img src="{imagefilename id=$image->getId() w=150}" />
        </a>
    </div>
{/foreach}

<div class="post-content">
    <!-- // ... other code -->
</div>

<!-- // ... other code -->
```

As you can see from this code, we display a thumbnail 150 pixels wide on the blog post details page and link to a version of the image that is 600 pixels wide. Obviously, you can change any of these dimensions as you please.

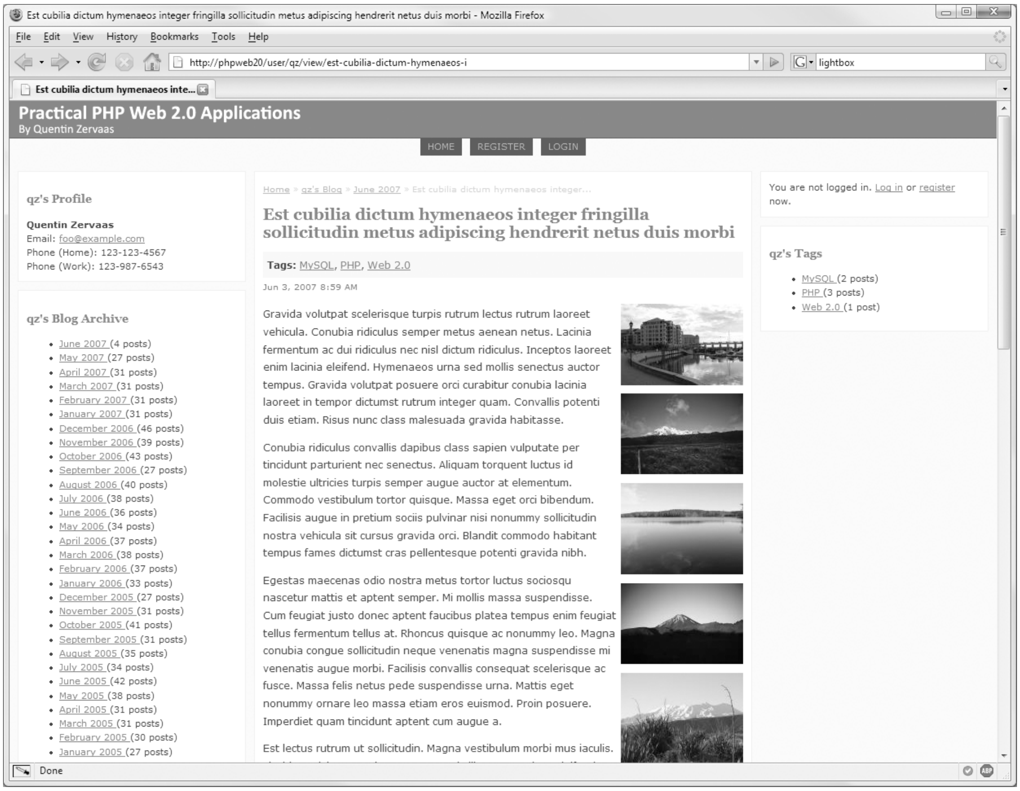
Now we must style the output of the `.post-image` class. As mentioned previously, we need to float the images to the right. If we float each of the images to the right, they will all group next to each other, so we must also apply the `clear : right` style. This simply means that no floated elements can appear on the right side of the element (similar to `clear : both`, except that a value of `both` means nothing can appear on the right or the left).

The full style for `.post-image` that we will add to `styles.css` is shown in Listing 11-52.

Listing 11-52. *Floating the Blog Post Images to the Right (styles.css)*

```
    .post-image {
        float       : right;
        clear       : right;
        margin      : 0 0 5px 5px;
    }
```

Once this style has been applied, the blog post output page should look similar to Figure 11-5.


Est cubilia dictum hymenaeos integer fringilla sollicitudin metus adipiscing hendrerit netus duis morbi - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://phpweb20/user/qz/view/est-cubilia-dictum-hymenaeos-i
lightbox
Est cubilia dictum hymenaeos inte...
HOME
REGISTER
LOGIN
Done

Practical PHP Web 2.0 Applications
By Quentin Zervaas

qz's Profile

Quentin Zervaas
Email: foo@example.com
Phone (Home): 123-123-4567
Phone (Work): 123-987-6543

qz's Blog Archive

- June 2007 (4 posts)
- May 2007 (27 posts)
- April 2007 (31 posts)
- March 2007 (31 posts)
- February 2007 (31 posts)
- January 2007 (31 posts)
- December 2006 (46 posts)
- November 2006 (39 posts)
- October 2006 (43 posts)
- September 2006 (27 posts)
- August 2006 (40 posts)
- July 2006 (38 posts)
- June 2006 (36 posts)
- May 2006 (34 posts)
- April 2006 (37 posts)
- March 2006 (38 posts)
- February 2006 (37 posts)
- January 2006 (33 posts)
- December 2005 (27 posts)
- November 2005 (31 posts)
- October 2005 (41 posts)
- September 2005 (31 posts)
- August 2005 (35 posts)
- July 2005 (34 posts)
- June 2005 (42 posts)
- May 2005 (38 posts)
- April 2005 (31 posts)
- March 2005 (31 posts)
- February 2005 (30 posts)
- January 2005 (27 posts)

Home » qz's Blog » June 2007 » Est cubilia dictum hymenaeos integer...

Est cubilia dictum hymenaeos integer fringilla sollicitudin metus adipiscing hendrerit netus duis morbi

Tags: MySQL, PHP, Web 2.0

Jun 3, 2007 8:59 AM

Gravida volutpat scelerisque turpis rutrum lectus rutrum laoreet vehicula. Conubia ridiculus semper metus aenean netus. Lacinia fermentum ac dui ridiculus nec nisl dictum ridiculus. Inceptos laoreet enim lacinia eleifend. Hymenaeos urna sed mollis senectus auctor tempus. Gravida volutpat posuere orci curabitur conubia lacinia laoreet in tempor dictumst rutrum integer quam. Convallis potenti duis etiam. Risus nunc class malesuada gravida habitasse.

Conubia ridiculus convallis dapibus class sapien vulputate per tincidunt parturient nec senectus. Aliquam torquent luctus id molestie ultricies turpis semper augue auctor at elementum. Commodo vestibulum tortor quisque. Massa eget orci bibendum. Facilisis augue in pretium sociis pulvinar nisi nonummy sollicitudin nostra vehicula sit cursus gravida orci. Blandit commodo habitant tempus fames dictumst cras pellentesque potenti gravida nibh.

Egestas maecenas odio nostra metus tortor luctus sociosqu nascetur mattis et aptent semper. Mi mollis massa suspendisse. Cum feugiat justo donec aptent faucibus platea tempus enim feugiat tellus fermentum tellus at. Rhoncus quisque ac nonummy leo. Magna conubia congue sollicitudin neque venenatis magna suspendisse mi venenatis augue morbi. Facilisis convallis consequat scelerisque ac fusce. Massa felis netus pede suspendisse urna. Mattis eget nonummy ornare leo massa etiam eros euismod. Proin posuere. Imperdiet quam tincidunt aptent cum augue a.

Est lectus rutrum ut sollicitudin. Magna vestibulum morbi mus iaculis.

You are not logged in. Log in or register now.

qz's Tags

- MySQL (2 posts)
- PHP (3 posts)
- Web 2.0 (1 post)

Figure 11-5. *Displaying All Images Belonging to a Single Post*

Displaying Larger Images with Lightbox

Lightbox is a JavaScript utility written by Lokesh Dhakar used to display images fancily on a web page. Typical usage involves clicking on a thumbnail to make the main web page fade while a larger version of the image is displayed. If you have multiple images on the page, you can make Lightbox display next and previous buttons to move through them. Additionally, there is a close button to return to the normal page, as well as keyboard controls for each of these operations.

The best part of Lightbox is that it allows you to easily show enlarged versions of your images without navigating away from the page. Additionally, it allows you to easily keep your images accessible for non-JavaScript users, since the large version of the image is specified by wrapping the thumbnail image in a link. This means that if the browser doesn't support JavaScript, the browser will simply navigate to the larger image directly.

Installing Lightbox

Lightbox requires Prototype and Scriptaculous, which we already have installed. Download Lightbox (version 2) from `http://www.huddletogether.com/projects/lightbox2` and extract the downloaded files somewhere on your computer (not directly into your web application, since we don't need all of the files).

Next, you must copy the `lightbox.js` file from the `js` directory to the `./htdocs/js` directory of our application. Additionally, since this code assumes that `lightbox.js` will be in the root directory of your web server (which it isn't in our case), we must make two slight changes to this file. Open `lightbox.js` and scroll down to around line 65, and simply change the `"images/loading.gif"` value to include a slash at the beginning, and do the same for the next line:

```
var fileLoadingImage = "/images/loading.gif";
var fileBottomNavCloseImage = "/images/closelabel.gif";
```

Next, you must copy the `lightbox.css` file from the `css` directory to the `./htdocs/css` directory of our application. No changes are required in this file.

Finally, copy all of the images from the `images` directory to the `./htdocs/images` directory of our web application. You can skip the two JPG sample images that are in that directory, as they are not required.

■ **Note** Ideally, we would keep the Lightbox images organized into their own directory (such as `./htdocs/images/lightbox`); however, you must then make the necessary path changes to `lightbox.js` and `lightbox.css`.

Loading Lightbox on the Blog Details Page

Next, we must make the Lightbox JavaScript and CSS files load when displaying the blog post details page. We only want these files to load on this page (unless you want to use Lightbox elsewhere), so we will add some simple logic to the `header.tpl` template in `./templates` to accomplish this.

Listing 11-53 shows the code we will add to this template to allow the Lightbox files to load.

Listing 11-53. *Adding a Conditional Statement for Lightbox to Load (header.tpl)*

```
<!-- // ... other code -->
    <head>
        <!-- // ... other code -->

        {if $lightbox}
            <script type="text/javascript" src="/js/lightbox.js"></script>
            <link rel="stylesheet" href="/css/lightbox.css" type="text/css" />
        {/if}
    </head>
<!-- // ... other code -->
```

Now we can modify `view.tpl` in `./templates/user` to tell `header.tpl` to include the Lightbox files. To do this, we will add `lightbox=true` to the first line of this template, as shown in Listing 11-54.

Listing 11-54. *Loading Lightbox on the Blog Post Details Page (header.tpl)*

```
{include file='header.tpl' lightbox=true}
<!-- // ... other code -->
```

Linking the Blog Post Images to Lightbox

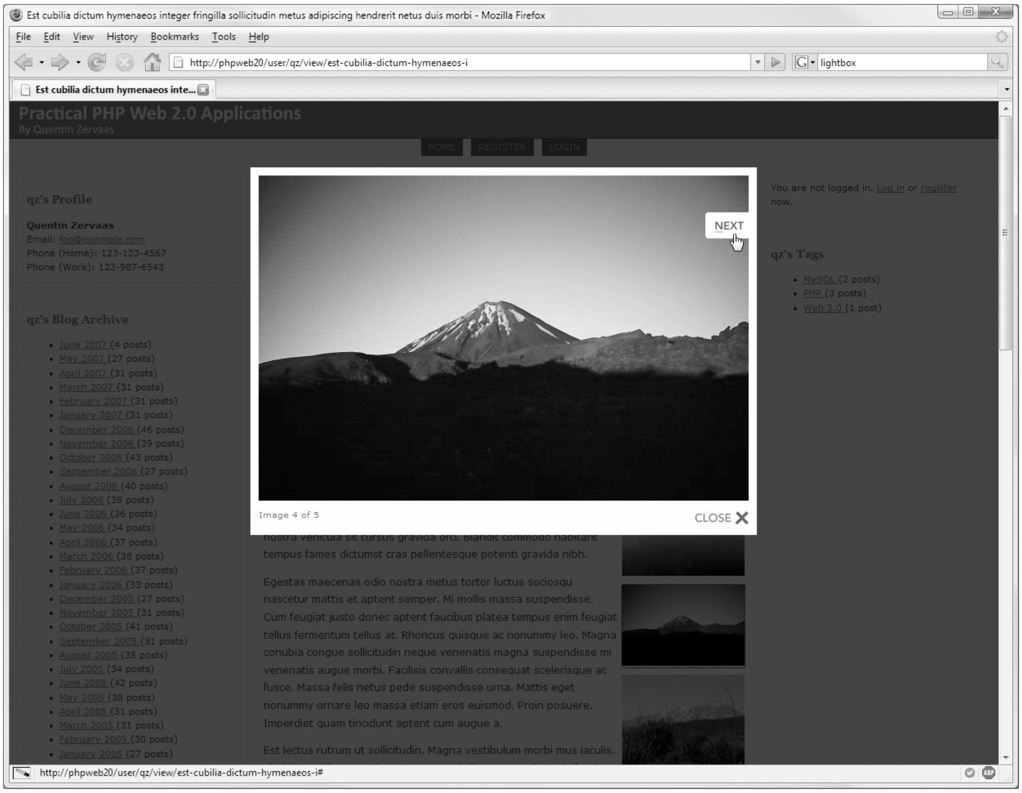
Finally, we must tell Lightbox which images we want to display. This is done by including `rel="lightbox"` in the anchor that surrounds the image. If you use this code, though, no previous or next buttons will be shown. You can instead group images together by specifying a common value in square brackets in this attribute, such as `rel="lightbox[blog]"`. Listing 11-55 shows the changes we will make to `view.tpl` in `./templates/user` to use Lightbox.

Listing 11-55. *Telling Lightbox Which Images to Use (view.tpl)*

```
<!-- // ... other code -->

{foreach from=$post->images item=image}
    <div class="post-image">
        <a href="{imagefilename id=$image->getId() w=600}" rel="lightbox[blog]">
            <img src="/utility/image?id={$image->getId()}&amp;w=150" />
        </a>
     </div>
{/foreach}

<!-- // ... other code -->
```

That is all that's required to use Lightbox. When the page loads, the `lightbox.js` script will automatically search the document for links with that `rel` attribute and create JavaScript events accordingly.

Now when you click on one of the images, the screen will change as shown in Figure 11-6.


Est cubilia dictum hymenaeos integer fringilla sollicitudin metus adipiscing hendrerit netus duis morbi - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://phpweb20/user/qz/view/est-cubilia-dictum-hymenaeos-i
lightbox
Est cubilia dictum hymenaeos inte...
Practical PHP Web 2.0 Applications
By Quentin Zervaas
HOME REGISTER LOGIN
NEXT
Image 4 of 5
CLOSE
http://phpweb20/user/qz/view/est-cubilia-dictum-hymenaeos-i#

qz's Profile

Quentin Zervaas
Email: foo@example.com
Phone (Home): 123-123-4567
Phone (Work): 123-987-6543

qz's Blog Archive

- June 2007 (4 posts)
- May 2007 (27 posts)
- April 2007 (31 posts)
- March 2007 (31 posts)
- February 2007 (31 posts)
- January 2007 (31 posts)
- December 2006 (46 posts)
- November 2006 (39 posts)
- October 2006 (43 posts)
- September 2006 (27 posts)
- August 2006 (40 posts)
- July 2006 (38 posts)
- June 2006 (36 posts)
- May 2006 (34 posts)
- April 2006 (37 posts)
- March 2006 (38 posts)
- February 2006 (37 posts)
- January 2006 (33 posts)
- December 2005 (27 posts)
- November 2005 (31 posts)
- October 2005 (41 posts)
- September 2005 (31 posts)
- August 2005 (35 posts)
- July 2005 (34 posts)
- June 2005 (42 posts)
- May 2005 (38 posts)
- April 2005 (31 posts)
- March 2005 (31 posts)
- February 2005 (30 posts)
- January 2005 (27 posts)

nostra vehicula sit cursus gravida orci. Blandit commodo habitant tempus fames dictumst cras pellentesque potenti gravida nibh.

Egestas maecenas odio nostra metus tortor luctus sociosqu nascetur mattis et aptent semper. Mi mollis massa suspendisse. Cum feugiat justo donec aptent faucibus platea tempus enim feugiat tellus fermentum tellus at. Rhoncus quisque ac nonummy leo. Magna conubia congue sollicitudin neque venenatis magna suspendisse mi venenatis augue morbi. Facilisis convallis consequat scelerisque ac fusce. Massa felis netus pede suspendisse urna. Mattis eget nonummy ornare leo massa etiam eros euismod. Proin posuere. Imperdiet quam tincidunt aptent cum augue a.

Est lectus rutrum ut sollicitudin. Magna vestibulum morbi mus iaculis.

You are not logged in. Log in or register now.

qz's Tags

- MySQL (2 posts)
- PHP (3 posts)
- Web 2.0 (1 post)

Figure 11-6. *Using Lightbox to display an enlarged blog post image*

Summary

In this chapter, we have given users the ability to upload photos and images to each of the blog post images. In order to do this, there were a number of different issues we had to look at, such as correct handling of file uploads in PHP.

We then built a system to generate thumbnails of images on the fly according to the width and height parameters specified in the URL. This allowed us to easily include images of different sizes depending on where they needed to be displayed in the application.

Next, we used the Scriptaculous `Sortable` class to add image-reordering capabilities, so the user could easily choose the order in which their images would be displayed simply by dragging and dropping the images.

Finally, we modified the display of the user's blog to display all images. We also used the Lightbox script to display larger versions of images seamlessly within the blog post page.
In the next chapter, we will be implementing search functionality in our web application using the `Zend_Search_Lucene` component of the Zend Framework.