CHAPTER 4

Building a Weblog

One of the most talked about Internet phenomenon in recent times is that of the Weblog (often shorted to *blog*). The concept of a blog—and the subsequent art of blogging—is that you provide your own online diary or journal in which you can scribe your thoughts for the world to see. The actual content that you pour into your blog is completely up to you, and the blog can be as formal, or informal, as you like. If you want to tell the world that your milk went bad and you need to pick up some from the store, a blog is where you write all about it.

The blog-reading public is not just obsessed with milk-longevity-related shenanigans, though. Although typically used as a vehicle to communicate thoughts online, blogs have also become a primary means by which various people connected to a hobby or project share what they are working on. This has been particularly popular with Open Source developers who use their blogs to give their readers a sneak peak of what's to come in the software they hack.

Although the basic function of a blog is to store a series of blog posts (often called *entries*), many blogs also include other features:

- **Commenting.** Readers of the blog can often leave comments. This can add real value to a blog, and conversations often strike up over varying subjects.
- **Categorization.** Blogs are often separated into different categories. This gives the blog author the ability to file entries into a specific section. This also allows readers to read only the category that interests them.
- **Archives.** Most blogs have some means of archiving and accessing previous entries. With blogs becoming as relevant a medium as "normal" Web sites, being able to access earlier entries is important.

In this project, you will build a blog that incorporates all of the preceding features. Aren't you lucky?

Project Overview: Blogtastic Use Case

The blog application created in this chapter is rather niftily titled Blogtastic. To get an overview of how to build the blog, here is a simple use case that demonstrates how typical users would interact with the application. Use cases are very handy for helping to visualize exactly how different interactions and content should be presented to users. The following is a synopsis of the use case for the Blogtastic application:

John visits Pauline's blog and, naturally, wants to see Pauline's latest entry. John is interested in reading the blog entry but would also like to see if any comments have been posted in response to the entry. If the blog entry has comments, the names of the commenters are added to the bottom of the blog, so John can see who posted each comment.

To access the blog and any corresponding comments, John clicks the title of the blog entry, and that specific entry (with comments) is displayed. In case John wants to leave a comment, a form for him to express his views is conveniently available on the same page. John fills outs the form and clicks the Submit button, after which the page is reloaded with John's comment added. John then whiles away the afternoon perusing through older blog entries in the archived entries page.

Later that day, Pauline decides she wants to add a new blog entry. Pauline visits a special page on the Web site where she can log in. As the blog's author, some additional options are made available only to her. Pauline can add a new blog entry to a specific category, or she can even add a new category. Pauline adds a new entry and then realizes she made a mistake. Fortunately, a special Edit button that she—and only she—can see displays on the page. Pauline uses this button to correct her mistake and avoid looking silly in front of John. Pauline secretly thinks she is better than John.

This is a typical example of a blog, and in this project, you will pour all of the preceding functionality into Blogtastic to match this use case.

NOTE

Take Your Time

Because this is the first database-driven project in this book, progress through the chapter at a pace that is comfortable to you. If you come across any concepts you don't understand, take a moment to stop, visit Google, and do some research to clear up the misunderstanding before you continue. When learning a new technology, never plough on if you don't understand the concepts; you will only dig a bigger hole to fall into.

Building the Database

The first step in the Blogtastic project is to build the database. To begin, create a new database in phpMyAdmin called *blogtastic*. Within this database, create four tables:

Table Name	What the Table Stores
categories	Different blog categories
entries	Blog postings
comments	Comments on blog entries
logins	Usernames and passwords

The schema of the tables is shown in Figure 4-1.

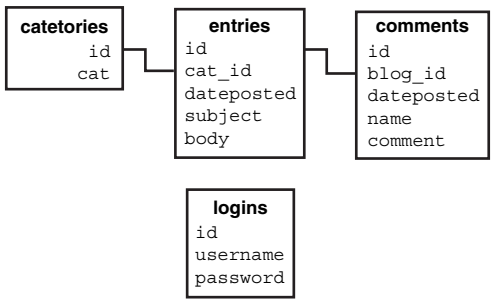


FIGURE 4-1
Even for a simple application such as a blogging engine, careful database design will save a lot of work in the long run.

Figure 4-1 shows how the four tables relate to each other. The first relationship is between the blog category (the *categories* table) and the blog entry (the *entries* table). To reference the correct category, the same `id` from the *categories* table is stored in the `cat_id` field of the *entries* table. In database parlance, the `id` field is known as the *primary key*, and the `cat_id` field is known as the *foreign key*. If these two values match, an explicit connection exists between the tables. In other words, if you know the `id` of the category (stored in `cat_id`), you can run a query to pull out the other category information (such as the name of the category) using that `id`. The second relationship—between the *entries* table and the *comments* table—works in exactly the same way; the `id` field from the *entries* table is connected to the `blog_id` in the *comments* table. This connects each comment with the correct blog entry.

The final table is the *logins* table, which has no relationship to the others; the table is used simply for logging each user into the application.

Implementing the Database

With the database design laid out, you can create the database tables. In phpMyAdmin, follow the steps discussed in Chapter 2 to add new tables, using these details:

The *categories* Table

- `id`. Make this a `TINYINT` (this type is used because there will not be many categories) and turn on `auto_increment` in the `Extras` column. Set this field as a primary key.
- `cat`. Make this a `VARCHAR`. Because a category title longer than 20 letters is unlikely, set the size to `20`.

The *entries* Table

- `id`. Make this an `INT` (several blog entries are possible) and turn on `auto_increment`. Set this field as a primary key.
- `cat_id`. Make this a `TINYINT` (the same type as the primary key it references—`id` in the *categories* table).
- `dateposted`. Use the `DATETIME` type. This data type stores the current date and time in the international ISO standard format, which is pretty clunky, but you can format the date later in PHP.
- `subject`. Make this a `VARCHAR`. Unless your blog title is extremely long, set the length of this field to `100`.
- `body`. Make this a `TEXT` field. If you ever want to store very large areas of text, the `TEXT` type is a good choice. You don't need to specify a length.

The *comments* Table

- `id`. Make this an `INT` (several comments are likely). Turn on `auto_increment` and set this field as a primary key.
- `blog_id`. Make this an `INT` (the same type as the `id` field in the *entries* table, to which it is related).
- `dateposted`. Use the `DATETIME` type.
- `name`. Make this a `VARCHAR`. Because comment titles longer than 50 characters is unlikely, set the length to `50`.
- `comment`. This is the main body of the comment. Set to the `TEXT` type.

The *logins* Table

- `id`. Make this a `TINYINT` (there will be very few logins; possibly only one). Turn on `auto_increment` and set this field as a primary key.
- `username`. Make this a `VARCHAR` and give it a length of `10`. (Enforcing a maximum length for the username is a common practice.)
- `password`. Make this a `VARCHAR` and give it a length of `10`. (As with usernames, enforcing a maximum length for a password is a common practice.)

Inserting Sample Data

With the tables created, insert some initial data into them so that you can test the code as you write it (again using phpMyAdmin). Remember that when you are adding data to any of these tables, do not fill in a number in the `id` column; this value will be handled by `auto_increment`.

Because this is the first project in the book, sample data has been add to the tables for you to ensure that everything connects as expected. As you work through the book and understand the database concepts in better detail, you can add additional sample data.

Sample Data for the *categories* Table

Add the following categories in this order: Life, Work, Music, and Food.

Sample Data for the *entries* Table

Add the information in Table 4-1 to the *entries* table.

Both entries reference the Life entry in the *categories* table, via the `cat_id`. In the `dateposted` field, use the Function combo box to select the NOW option, which fills the field with the date and time you add the entry to the table.

TABLE 4-1 The sample data for the entries table enables you to follow along with the rest of this chapter's samples.

CAT_ID	DATEPOSTED FIELD	SUBJECT	BODY
1	Select NOW from the function box	Welcome to my blog!	This is my very first entry in my brand-new blog.
1	Select NOW from the function box	Great blog!	I have decided this blog is: Really cool!

Sample Data for the *comments* Table

Add the information in Table 4-2 to the *comments* table.

TABLE 4-2 The comments table has just a few sample comments, used for demonstration purposes.

CAT_ID	DATEPOSTED FIELD	SUBJECT	BODY
1	Select NOW from the function box	Bob	Welcome!
1	Select NOW from the function box	Jim	Hope you have lots of fun!

In this table, reference the first blog entry (the one with the 'Welcome to my blog!' subject) by supplying a value of 1 to the `blog_id` field.

Sample Data for the *logins* Table

In this table, add a single entry with a username and password of your choice. This example includes a username of "jono" and a password of "bacon".

Starting to Code

Start out by creating your project configuration file. This configuration file makes customization of the blog easy, for you or for other users who run it.

Create a new directory in *htdocs* on your computer. Inside this directory, create a new file called *config.php* (as shown in Example 4-1):

EXAMPLE 4-1 Using a configuration file makes customization and personalization a piece of cake.

```
<?php

$dbhost = "localhost";
$dbuser = "root";
$dbpassword = "";
$dbdatabase = "blogtastic";

$config_blogname = "Funny old world";

$config_author = "Jono Bacon";

$config_basedir = "http://127.0.0.1/sites/blogtastic/";

?>
```

NOTE

Configuration Files for Distributed Applications

If you plan on writing a Web application that you intend to distribute so that others can download, install, and run it, easy configuration is essential. This is where a standard configuration file is useful. Settings that the user may want to tweak can be kept out of the main code.

Most of this file is simple configuration. The first four lines should look familiar to you; they are the normal database settings. You can change these to match your own database setup.

Below the database settings, another three variables are set. The first one (`$config_blogname`) sets the name of the blog. The second variable (`$config_author`) enables the user to set his name as the author. The final variable (`$config_basedir`) refers to the location of the blog, in URL form. This variable is particularly important and is used later in various parts of the code, specifically to redirect to different pages.

TIP

You may have noticed that three of the configuration variables begin with "config." This distinguishes these variables from other, non-configuration-related variables in your code and is a great way to remember what a particular variable is associated with.

Designing a User Interface

In the previous chapter, you created a generic Web site and made use of a number of `include()` and `require()` functions to separate different parts of the site. This application uses the same concepts to provide a consistent look and feel.

NOTE

The stylesheet.css File

This project uses the stylesheet.css file created in Appendix A. Copy the file to the current project directory to apply the stylesheet to the project.

Creating the Header File

Create a file called *header.php* and add the code shown in Example 4-2.

EXAMPLE 4-2 This simple header file will be used across all pages.

```
<?php
require("config.php");
?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title><?php echo $config_blogname; ?></title>
<link rel="stylesheet" href="stylesheet.css" type="text/css" />
</head>
<body>
<div id="header">
<h1><?php echo $config_blogname; ?></h1>
[<a href="index.php">home</a>]
</div>

<div id="main">
```

There are a few important points to note about this code. Here, a PHP block is opened at the top to include the *config.php* code in the page. The `require()` function—as opposed to `include()`—has been used here, because *config.php* is essential to the correct behavior of this page. If *config.php* does not exist, the entire application breaks down during any database work.

Most of the HTML in this code should look fairly straightforward, but you might have also spotted a small chunk of PHP in the `<title>` tag. In the title, the contents

of the `$config_blogname` variable from *config.php* is displayed (refer to Example 4-1); this adds the blog name for the blog in the title bar of the browser window. This variable's value is also repeated inside the first `<div>` within the `<h1>` tag. This provides some basic (very, very basic!) branding.

The final addition to the code is a link beneath the `<h1>` tag to the main page (*index.php*, which you'll create shortly). To keep this project simple, links to different parts of the site will appear in this header `<div>`. The last line of code opens the main `<div>` in similar fashion to the Web site created in Appendix A.

Creating the Footer File

With the yin of the header file complete, it is now time to create the yang of the footer file. Create a new file called *footer.php* that looks like Example 4-3.

EXAMPLE 4-3 Like the header file, this footer will be shared across all pages.

```
</div>

<div id="footer">
&copy; <?php echo $config_author; ?>
</div>
</body>
</html>
```

The first line of the file ends the main `<div>` that was opened at the end of the header file (see Example 4-2 for the opening of this `<div>` tag). After this, you create a footer `<div>` to contain a copyright symbol (achieved with the special `&copy;` markup) and then add a small PHP block to (again) display the contents of a variable from the *config.php* file. This gives the site suitable credit for whoever runs it.

You can test that your header and footer files work by creating a file called *index.php* and adding the code in Example 4-4.

EXAMPLE 4-4 With a header and footer, actual site pages become very simple.

```
<?php

require("header.php");

require("footer.php");

?>
```

When you access the *index.php* page in your browser, you should see the simple design shown in Figure 4-2.

FIGURE 4-2 With a little configuration and a little HTML, the skeleton of the application is in place.

Displaying a Blog Entry

You are now ready to begin crafting some code that actually resembles a blogging application. With your database already loaded with sample content, the first logical step is to display the contents of the most recent blog entry. This involves creating a simple SQL query and then displaying the results of the query (the latest entry) on the page.

Before you create the query, however, you need to add he code to connect to the database. Within this particular application, database access occurs on every page; therefore, adding the database connection code to the main *header.php* file makes sense. This file is included in every page, so it is a logical home for the connection code.

After the `require("config.php")` line of *header.php*, add the following lines (which were explained in Chapter 2):

```
<?php
require("config.php");
$db = mysql_connect($dbhost, $dbuser, $dbpassword);
mysql_select_db($dbdatabase, $db);
?>
```

Building the Query

To build the SQL query, think about the kind of information you want the database to return. For the latest blog entry, you want all the information from the *entries* table (such as the subject, body, and date the blog was posted), but you also need to get the category to which the entry belongs.

The name of the category isn't stored in the *entries* table, however; only the `cat_id` is. With this in mind, you need to ask for all the information from the *entries* table and also ask for the category name in which the category `id` matches the `cat_id` from the *entries* table.

Here is the SQL you need:

```
SELECT entries.*, categories.cat FROM entries, categories
  WHERE entries.cat_id = categories.id
  ORDER BY dateposted DESC
  LIMIT 1;
```

One of the great benefits of SQL is that you can read it fairly easily from left to right. Additionally, if you lay out your SQL on separate lines, you easily see the four main parts of the SQL query:

1. The command (first line)
2. The conditions under which the command is executed (second line)
3. Any ordering requirements (third line)
4. Any limits (fourth line)

If you read the SQL from the beginning to the end, this is what happens:

> Select (`SELECT`) every field from the *entries* table (`entries.*`) and the `cat` field from the *categories* table (`categories.cat`) with the condition (`WHERE`) that the `cat_id` field from the *entries* table (`entries.cat_id`) is equal to (`=`) the `id` field from the *categories* table (`categories.id`). Order the results by (`ORDER BY`) the `dateposted` field in descending order (`DESC`) and only show a single result (`LIMIT 1`).

The aim of the query is to limit the results that come back and return only the last entry that was added to the database. Without the `ORDER BY` clause, the query would bring back every entry in the order that it was added. By adding the `ORDER BY` line, the results come back in descending date order (the last date is first). Finally,

to return only the latest entry (which is the first result from the query), you use `LIMIT 1` to return only a single record.

To run the query in your Web application, you need to first construct the SQL query code inside a variable and then send it off to the database. When you get the data back, you can access the row(s). Between the two `require` lines, add the following code to the *index.php* file:

```
<?php

require("header.php");

$sql = "SELECT entries.*, categories.cat FROM entries, categories
  WHERE entries.cat_id = categories.id
  ORDER BY dateposted DESC
  LIMIT 1;";
$result = mysql_query($sql);
$row = mysql_fetch_assoc($result);

require("footer.php");

?>
```

The first added line constructs the SQL query and stores it in a variable called `$sql`. To actually send the data to the MySQL server, use the `mysql_query()` function, which puts the result of the query in the `$result` variable. On the final line, the `mysql_fetch_assoc()` function pulls the row out of `$result` and stores it in an array.

A QUICK NOTE...

Because only one row is coming back, there is no need to use a `while()` loop to iterate through the rows returned from the query. Recall that more details on iterating through results are found in Chapter 2.

Displaying the Entry

With the query result stored in the `$row array`, you just need to crack open the array, pull out the data, and display it on your Web page. Refer to each field inside the square brackets in the array (such as `$row['subject']` for the `subject` field).

Add the following code after the `mysql_fetch_assoc()` line:

```
$row = mysql_fetch_assoc($result);
echo "<h2><a href='viewentry.php?id=" . $row['id']
 . "'>" . $row['subject'] .
     "</a></h2><br />";
echo "<i>In <a href='viewcat.php?id=" . $row['cat_id']
```

```
 ."'>" . $row['cat'] .
     "</a> - Posted on " . date("D jS F Y g.iA",
strtotime($row['dateposted'])) .
     "</i>";
echo "<p>";
echo nl2br($row['body']);
echo "</p>";

require("footer.php");

?>
```

This code creates a second-level heading tag and, within it, a link to a page called *viewentry.php* (which will be created later to view specific blog entries). To show a blog entry, the page needs to be passed the `id` of the specific blog entry to display. To achieve this, you add a question mark to the end of the filename and then add the variable (such as `id=1`). This process results in a link such as *viewentry.php?id=1*.

Instead of hard coding the value of the variable, however, use the contents of `$row['id']` (from the query) as the value. After you close the first part of the link tag, append the subject from the query and then add the closing link tag. You will see a number of these long, concatenated link tags in your programming, which can be better understood with the aid of a table.

Table 4-3 shows how the HTML is gradually built, step by step. Remember that the `.` glues these different parts together.

TABLE 4-3 It's often easiest to view long strings of code as a series of individual parts. On each line, the bolded text results from the code in the Code column.

CODE	HTML OUTPUT
`<h2><a href='viewentry.php?id=`	**`<h2><a href='viewentry.php?id=`**
`$row['id']`	`<h2><a href='viewentry.php?id=`**`1`**
`'>`	`<h2><a href='viewentry.php?id=`**`1'>`**
`$row['subject']`	`<h2><a href='viewentry.php?id=1'>`**`Welcome to my blog!`**
`</a></h2>`	`<h2><a href='viewentry.php?id=1'>Welcome to my blog!`**`</a></h2>`**

On the second line of code, another link is built in the same way; this link points to a page called *viewcat.php*. Instead of the entry `id` being passed as a variable, the category `id` is passed.

Next, the date is displayed. If you output an unformatted date from the database, the date would look something like this:

```
2005-08-01 18:02:32
```

Notice that the preceding result is not in the most useful of formats. Use `strtotime()` and `date()` to clean this up for human consumption.

The `strtotime()` function converts the date into a UNIX timestamp. This timestamp refers to the number of seconds since 12:00 a.m. on January 1, 1970. The time is known as the *Epoch*, and when you have this number of seconds, you can then feed it into the `date()` function to format those seconds into something more useful.

The `date()` function converts this number of seconds into a readable date, using several special format symbols (`D jS F Y g.iA` in this example). Each of these symbols formats a particular part of the date and time. You can find out more about these and other symbols in the PHP manual entry for dates at `http://www.php.net/date`.

Table 4-4 gives an example for 2:35 p.m. on April 6, 2005.

TABLE 4-4 Each letter represents a portion of the date, as well as how to format that date.

DATE() SYMBOLS	DATE
`D`	**`Wed`**
`D j`	`Wed` **`6`**
`D jS`	`Wed 6`**`th`**
`D jS F`	`Wed 6th` **`April`**
`D jS F Y`	`Wed 6th April` **`2005`**
`D jS F Y g`	`Wed 6th April 2005` **`2`**
`D jS F Y g.`	`Wed 6th April 2005 2.`
`D jS F Y g.i`	`Wed 6th April 2005 2.`**`35`**
`D jS F Y g.iA`	`Wed 6th April 2005 2.35`**`PM`**

Finally, in the last bit of the code, the body of the blog entry is presented. The first of these three lines opens a paragraph tag, and the second actually outputs the

content of the blog posting. You need to pass the contents of the database entry through `nl2br()`. This useful little function converts any empty lines into legitimate HTML `<br />` tags. The final line closes off the paragraph tag. See the final result in Figure 4-3.

FIGURE 4-3 Your blog entry is ready for the world to see.

Adding a Comments Summary

One of the planned features for this blog is the capability for visitors to add comments to a blog entry. These comments should be visible on the *viewentry.php* page, linked via the subject of the blog (which you just added to *index.php*).

When comments are posted to a blog entry, it's helpful to provide a comments summary. When comments have been posted, you can display the number of comments and the names of the posters. It's also useful to have the names of the posters double as hyperlinks; that is, when you click the poster's name, the application jumps to that poster's comment on the *viewentry.php* page.

After the code already in place in *index.php*, add the following lines:

```
echo nl2br($row['body']);
echo "</p>";

echo "<p>";

$commsql = "SELECT name FROM comments WHERE blog_id = " . $row['id'] .
           " ORDER BY dateposted;";
$commresult = mysql_query($commsql);
$numrows_comm = mysql_num_rows($commresult);

require("footer.php");

?>
```

This chunk of code creates a new paragraph tag, and then a new SQL query to select the `name` field from the *comments* table, where `blog_id` contains the `id` of the current blog entry (stored in `$row['id']`). The entire query is ordered by date (using the `dateposted` field). This query is then executed with the `mysql_query()` command, and the result is stored in `$commresult`.

On the final line, a new function called `mysql_num_rows()` is used to count how many rows are returned from the query, and this number is stored in the `$numrows_comm` variable. The `mysql_num_rows()` function is incredibly useful, because you can use it to determine how to format the comments summary. If no rows are returned, display `'No comments'`; if `1` or more results are returned, display the posters' names:

```
$commsql = "SELECT name FROM comments WHERE blog_id = " . $row['id'] .
           " ORDER BY dateposted;";
$commresult = mysql_query($commsql);
$numrows_comm = mysql_num_rows($commresult);
if($numrows_comm == 0) {
  echo "<p>No comments.</p>";
}
else {
  echo "(<strong>" . $numrows_comm . "</strong>) comments : ";
  $i = 1;
  while($commrow = mysql_fetch_assoc($commresult)) {
    echo "<a href='viewentry.php?id=" . $row['id'] ."#comment" . $i .
              "'>" . $commrow['name'] . "</a> ";
    $i++;
  }
}
echo "</p>";
```

In this block of code, an `if` statement is used to check if `$numrows_comm` has `0` rows. If it does, `No comments` is echoed to the screen. If `$numrows_comm` is not equal to `0`, control moves into the `else` statement.

Inside the `else`, an `echo` line prints a bracket and then, in bold typeface, outputs the number of rows stored in `$numrows_comm` and finally outputs a closing bracket and the word `comments`. If there were two comments, the output would be

```
(2) comments
```

The next step is to display each comment, as well as a link to that comment, using an anchor.

The anchors used in *viewentry.php* are in the form `#comment1`, `#comment2`, and so on. To add these numbered anchors in *index.php*, start at `1` and increment each time a comment link is output.

ALL ABOUT ANCHORS

Anchors are handy methods of linking to different parts of a single page. To reference an anchor, you add the name of the anchor to the URL. As an example, linking to *example.php#theory* jumps to the `theory` anchor on the *example.php* page. At some point in *example.php*, there should be something like this:

```
<a name="theory">
```

Now, when *example.php#theory* is referenced, the page will jump to that tag.

Back in the code, you'll see that a variable called `$i` is created and set to `1`. Next, a `while` loop iterates through the rows. A link to *viewentry.php* is created, and `id=[<entry-id>]` is added to each. In addition to the `id` being appended, the comment anchor (such as `#comment1`) is added, using `$i`. Finally, the value of `$i` is increased by `1`, ready for use on the next link. The completed output should look something like this (obviously with different names if you have added different comments):

```
(2) comments : Jim Bob
```

NOTE

If you are using the sample detail discussed earlier in the chapter, you will continue to see "No comments" because no comments are associated with the second blog entry. To resolve this, use phpMyAdmin to add some records to the *comments* table and specify a value of `2` in the `blog_id` field.

You can see the comments shown in Figure 4-4.

Displaying Previous Blog Entries

It is often convenient to see the last five or so blog entries, so that if a user misses a few entries, she can access them easily without having to dig through the archives.

First, create the query. Luckily, this query is the same as the one you used to find the latest blog entry—the only difference being that instead of limiting the results to a single entry, you limit the result set to five entries. Do this by changing the `LIMIT 1` line to `LIMIT 1, 5`. This ensures that you get records 0 to 4.

FIGURE 4-4 Displaying comments on the front page shows visitors that your blog entries cause discussion and debate.

TIP

When you use `LIMIT`, the first record returned is marked as the zeroth. As such, `LIMIT 1,5` returns the first record through to the fifth. `LIMIT 0, 1` is synonymous with `LIMIT 1`.

Add the following code to your page:

```
echo "</p>";
$prevsql = "SELECT entries.*, categories.cat FROM entries, categories
  WHERE entries.cat_id = categories.id
  ORDER BY dateposted DESC
  LIMIT 1, 5;";
$prevresult = mysql_query($prevsql);
$numrows_prev = mysql_num_rows($prevresult);
```

This query counts the number of rows returned so you can display the relevant information. Now, add the code to display the results:

```
$numrows_prev = mysql_num_rows($prevresult);

if($numrows_prev == 0) {
  echo "<p>No previous entries.</p>";
}
else {
```

```
  echo "<ul>";

  while($prevrow = mysql_fetch_assoc($prevresult)) {
    echo "<li><a href='viewentry.php?id="
 . $prevrow['id'] . "'>" . $prevrow  ['subject']
 . "</a></li>";
  }
}

echo "</ul>";
```

If no rows were returned in the query, the text `No previous entries.` is displayed. If rows are returned, the `else` block is executed and the previous entries are displayed in an unordered list.

Inside the `else` block, use a `while` loop to iterate through the results from the query to create the blog entry subjects with the `<li>` and `</li>` tags. The subject is linked to *viewentry.php* with the relevant `id` appended as a variable in the link. The end result is shown in Figure 4-5.

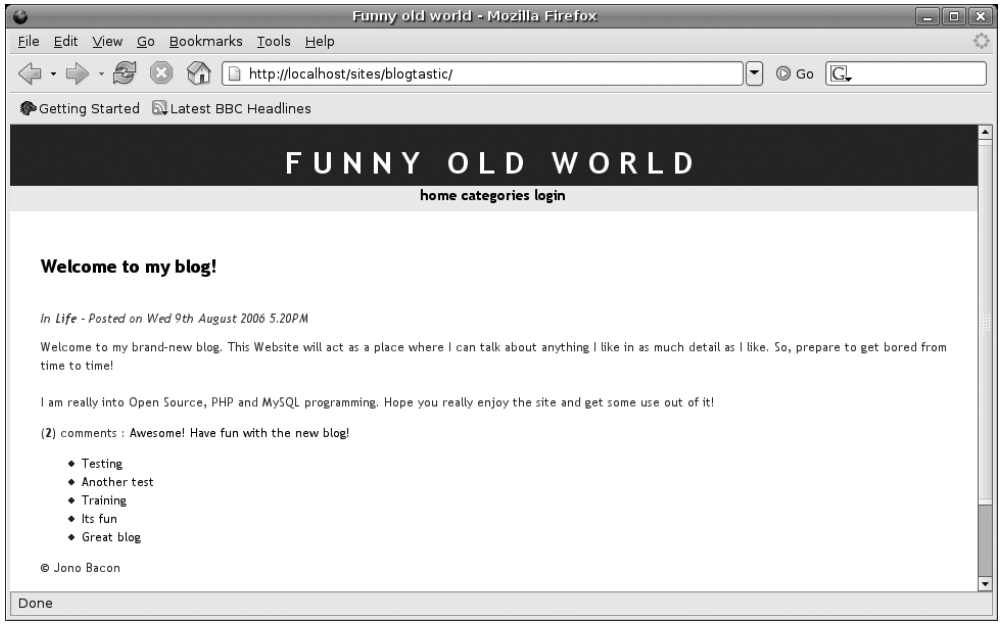
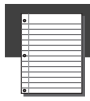


FIGURE 4-5 Including previous blog entries shows visitors that your blog gets updated frequently.



NOTE

Ordered and Unordered Lists

Within HTML, Web developers often use ordered and unordered lists to create bullet points. To create a *numbered* list, you use the `<ol>` and `</ol>` ordered list tags. To create an unnumbered bullet point list, use the unordered `<ul>` and `</ul>` tags.

List items are placed inside `<li>` and `</li>` tags. An example of an unordered list is shown as follows:

```
<ul>
   <li>One item</li>
   <li>Another item</li>
</ul>
```

VIEWING SPECIFIC ENTRIES

When *index.php* was created, three distinctive sections were added to the page:

- Main blog entry
- Number of comments
- Previous blog entries

In the main blog entry and previous entry sections, you link to another page called *viewentry.php*. The *viewentry.php* page has a number of important features:

- The page displays the contents of the blog entry.
- The page uses virtually the same code from *index.php*.
- The need to create the anchors that were added to the comment names (and links) in *index.php*.
- The page provides a form to post comments about the blog entry.
- The form is displayed, and when the user fills it in, the comment is added to the database.

This page is an important step in building the blog, so without further ado, it's time to get going and do some coding!

Validating the Request URL

The first step for the *viewentry.php* page is to ensure it's requested with valid date. Whenever you work on a Web project, it is important to verify that any changeable

information (such as the ID of an entry or comment) is legitimate. This verification process is known as *validation*. In this project, validation is applied to only the variables that are added to the address of the site. These variables are visible, and the user can change them by editing the address in the browser.

NOTE

Validation, Step by Step

The reason you will validate only GET variables, and not other types of information, is to make validation easier to learn. This application introduces some basic concepts and keeps things simple. Later projects in the book explore validation in more detail, and you can return to earlier projects and apply these skills later.

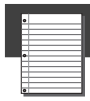
Although GET variables can be set to letters or numbers, virtually every GET variable in this book is set to a number. When you created *index.php* and the links to *viewentry.php*, each of them included a GET variable that contained a numeric id.

To validate a numeric variable, feed it into a block of code that runs some simple tests. Add the following code to the beginning of *viewentry.php*:

```
<?php

require("config.php");

if(isset($_GET['id']) == TRUE) {
  if(is_numeric($_GET['id']) == FALSE) {
    $error = 1;
  }

  if($error == 1) {
    header("Location: " . $config_basedir);
  }
  else {
    $validentry = $_GET['id'];
  }
}
else {
  $validentry = 0;
}
```

The first line includes *config.php*. Unlike the previous example, *header.php* has not been included (yet). If validation fails, you'll redirect users to another page, so there's no need to show the HTML in *header.php* until these tests have passed.

The next line is the first if condition. The isset() function checks to see if the GET variable exists. If it does, isset() returns TRUE; if not, validentry is set to 0.



NOTE

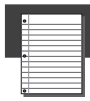
Redirection Fun and Games

Redirection is when you automatically jump to another page on the Web site. There are two main methods of redirecting to a page:

- Use JavaScript. The problems with this technique are that not all browsers fully support JavaScript, and users and network managers can also turn off JavaScript.
- Use HTTP headers. Use the HTTP headers that are present in every Web page to change the current page. This technique is supported by every browser, but it can occur only before any data is sent to the client. This same restriction applies to sessions when you use `session_start()` at the beginning of a page.

As a general rule, use HTTP headers for redirection, because of its availability in all browsers and ease of use.

Assuming a variable is being sent, a check is made to ensure the value is numeric; if someone sets the variable to "bananas," for example, this is obviously incorrect. The `is_numeric()` function tests the `GET` variable; if the result is `false`, `error` is set to `1`.



NOTE

The Nasty World of SQL Injection

One of the risks of using `GET` variables is SQL injection. Imagine that you have a SQL statement such as the following:

```
SELECT * FROM entries WHERE id = <id value>
```

and where *<id value>* is, you add the value from the `GET` variable:

```
$sql = "SELECT * FROM entries WHERE id = " . $_GET['id'];";
```

This code assumes that the value of `id` is numeric. If you don't check for this, a malicious user could try to inject SQL code into the query. Imagine what would happen if the user added `1; DROP DATABASE blogtastic;`. The following SQL is now executed:

```
SELECT * FROM entries WHERE id = 1; DROP DATABASE blogtastic;
```

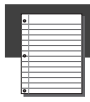
This code would result in a lost database (assuming the user had appropriate permissions)! To protect against this risk, always ensure that numeric `GET` values are actually numeric.

Next, if `error` is indeed equal to `1` (indicating a non-numeric value), the `header()` command redirects to the main page. The `header()` command is passed the `Location` header and the full location to redirect to (such as `Location: http://localhost/blogtastic/`). In the code, the "Location:" text is added, and then the location is picked out of the `config_basedir` variable from *config.php*.

TIP

When using the Location header, you will need to provide a complete URL such as http://www.foo.com/—as opposed to www.foo.com or foo.com.

If `error` is not set to `1`, the `validentry` variable is set to the value of the `GET` variable. With this validation in place, the code below the `header()` function runs only with a valid `GET` variable.



NOTE

Don't Confuse the User with Errors

When an invalid variable is detected, this script redirects to a legitimate page instead of displaying an error message. When considering the usability of your Web application, it generally makes sense to redirect rather than report an error. Error messages are rarely useful to users, and anyone who has the knowledge to adjust the `GET` variable on the URL is probably fully aware that they are tampering with the application instead of using the application. Automatically redirecting avoids potentially confusing the user with error messages.

Showing the Entry

With the validation complete, you can display the blog entry. This code looks very similar to the code on *index.php*. First, add the header HTML:

```
require("header.php");
```

You now need to determine which type of query to run. Inside the validation checking code, `validentry` is set to either `0` (if no variable was supplied) or to the `ID` of the entry to view.

If `validentry` is set to anything other than `0`, the query is simple—ask for that specific blog entry. If the value is `0`, however, the query should load the latest blog entry (the same behavior as *index.php*):

```
require("header.php");
if($validentry == 0) {
  $sql = "SELECT entries.*, categories.cat FROM entries, categories " .
         " WHERE entries.cat_id = categories.id " .
         "ORDER BY dateposted DESC " .
         " LIMIT 1;";
}
else {
  $sql = "SELECT entries.*, categories.cat FROM entries, categories " .
         "WHERE entries.cat_id = categories.id
 AND entries.id = " . $validentry .
         " ORDER BY dateposted DESC LIMIT 1;";
}
```

Send the query to to the database with the `mysql_query()` function:

```
else {
  $sql = "SELECT entries.*, categories.cat FROM entries, categories " .
         "WHERE entries.cat_id = categories.id
 AND entries.id = " . $validentry .
         " ORDER BY dateposted DESC LIMIT 1;";
}
$result = mysql_query($sql);
```

Now you can present the results to the user. This code is virtually identical to the code that you wrote on *index.php* to present the blog entry. The only real difference is that the subject of the entry is not linked to anything.

```
$result = mysql_query($sql);

$row = mysql_fetch_assoc($result);
echo "<h2>" . $row['subject'] . "</h2><br />";
echo "<i>In <a href='viewcat.php?id=" . $row['cat_id'] ."'>" .
     $row ['cat'] ."</a> - Posted on " .
     date("D jS F Y g.iA", strtotime($row['dateposted'])) ."</i>";
echo "<p>";
echo nl2br($row['body']);
echo "</p>";
```

The main blog entry section is now complete.

Showing Blog Comments

To display the comments, first create a SQL query that will get the comments for the current blog entry:

```
echo "</p>";

$commsql = "SELECT * FROM comments WHERE blog_id = " . $validentry .
           " ORDER BY dateposted DESC;";
$commresult = mysql_query($commsql);
$numrows_comm = mysql_num_rows($commresult);
```

You count the number of rows again with `mysql_num_rows()` and use the value to determine if any comments have been posted. If `numrows_comm` is equal to `0`, the text `No comments` is displayed; otherwise, the `else` block of code is executed.

```
$numrows_comm = mysql_num_rows($commresult);

if($numrows_comm == 0) {
  echo "<p>No comments.</p>";
}
else {
  $i = 1;

  while($commrow = mysql_fetch_assoc($commresult)) {
    echo "<a name='comment" . $i . "'>";
    echo "<h3>Comment by " . $commrow['name'] . " on " .
         date("D jS F Y g.iA",
 strtotime($commrow['dateposted'])) . "</h3>";
    echo $commrow['comment'];
    $i++;
  }
}
```

Inside the `else`, you perform two basic functions: display each comment and then create an anchor on each one that can match up with the anchors referenced by *index.php*.

At the top of the `else` block, you first set `i` to `1`; this variable is used as a counter to implement the anchors. Next, the `while` loop iterates through each comment from the query and creates the anchor. A link is created with a `name` attribute set to the text `comment`, with the value of `i` appended (resulting in, for example, `comment2`). The main comment fields are then displayed in a similar way to the main blog entry. Finally, the `i` variable is incremented by `1`, preparing for the next comment's output.

Build the Comment Form

Allowing a user to add comments involves three distinct actions:

- Display the form for adding comments.
- Process the form and add its data to the database after the user clicks the Submit button.

- Reload the page and show the new comment.

QUICK NOTE...

This functionality is a little more complex than the previous sections, largely because you need to add some code to various parts of the page, instead of just adding one line at a time to the end of your page.

First, add the main form shown in Figure 4-6.

FIGURE 4-6 Forms are useful for allowing users to contribute comments.

To do this, close off the PHP block at the bottom of the page with ?> and add the following HTML:

```
    echo $commrow['comment'];
    $i++;
  }
}
```

```
?>

<h3>Leave a comment</h3>

<form action="<?php echo $SCRIPT_NAME
 . "?id=" . $validentry; ?>" method="post">
<table>
<tr>
  <td>Your name</td>
  <td><input type="text" name="name"></td>
</tr>
<tr>
  <td>Comments</td>
  <td><textarea name="comment" rows="10" cols="50"></textarea></td>
</tr>
<tr>
  <td></td>
  <td><input type="submit" name="submit" value="Add comment"></td>
</tr>
</table>
</form>
```

This code creates a table that contains a number of form elements. In the `<form>` tag, `action` specifies the page that will process the data from this form. As shown here, you can use a special PHP variable called `SCRIPT_NAME` to reference the name of the current file (in this case, *viewentry.php*). This reference is useful if you later decide to change the filename of your script; you then don't need to change your code. The `method` attribute in the tag indicates whether you want to submit your variables as `POST` or `GET` variables.

Inside the `action` attribute, the `validentry` variable is added as a `GET` variable. When you process the data from the form, you need to indicate the `ID` of the blog entry to which you are posting the comment.

The HTML in the preceding form itself is pretty self-explanatory.

NOTE

GET Variables Versus Hidden Form Fields

Another technique of sharing a variable between the form and the script that processes it is to use the `hidden` form element:

```
<input type="hidden" name="example" value="21">
```

The `value` attribute of the form can then be accessed as a normal variable with `_GET` or `_POST` in your PHP code.

Processing forms on Web pages works in a rather backwards fashion. At the top of your page—before showing any HTML—you need to check to see if the Submit button has been clicked by checking for the `_POST['submit']` variable. If this variable exists, the user has submitted a form. If the variable does not exist, you should assume that the user has not actually seen the form yet and, therefore, need to display it. It sounds crazy, but hang in there—it will all make sense momentarily.

Insert the following code after your validation code, before you include the *header.php* file:

```
else {
  $validentry = 0;
}

if($_POST['submit']) {
  $db = mysql_connect($dbhost, $dbuser, $dbpassword);
  mysql_select_db($dbdatabase, $db);

  $sql = "INSERT INTO comments(blog_id, dateposted,
 name, comment) VALUES(" .
    $validentry . ", NOW(), '" . $_POST['name']
 . "', '" . $_POST['comment'] . "');";
  mysql_query($sql);
  header("Location: http://" . $HTTP_HOST
 . $SCRIPT_NAME . "?id=" . $validentry);
}
else {
  // code will go here
}

require("header.php");
```

The first line checks if the `submit` POST variable exists. For explanation purposes, assume that the Submit button has been clicked and the variable exists. The code then connects to the database. (Remember, you have not included header.php yet, so no database connection is available.)

NOTE

Don't Blow Up Your Headers

When you use a header redirect, always ensure that *no* data is displayed on the page before the header is sent—this includes white space. As a simple example of how important this is, add a single space before the `<?php` instruction and reload the page. You should now get a lot of "headers been sent" errors. Whenever you see these errors, check that there are no erroneous letters or white space either in the page itself or within the files that are included (such as *config.php*).

The next line is the SQL query. This query inserts the data into the database with an INSERT statement. A typical INSERT statement looks like this:

```
INSERT INTO table(field1, field2)
     VALUES ('data for field 1', 'data for field 2');
```

When you construct the SQL statement in your sql variable, you concatenate the various variables from the form that are accessed with _POST. To demonstrate how this fits together, imagine that you are adding a comment to the blog entry with 2 as an ID, at 2:30 p.m. on August 10, 2005. Assume that the user types "Bob Smith" as the name and "I really like your blog. Cool stuff!" as the comment. Table 4-5 demonstrates how the query is built.

TABLE 4-5 The sql variable is built up into an INSERT statement

Concatenated Element	SQL Statement
`INSERT INTO comments(blog_id, dateposted, name, comment) VALUES(`	**`INSERT INTO comments(blog_id, dateposted, name, comment) VALUES(`**
`validentry`	`$INSERT INTO comments(blog_id, dateposted, name, comment) VALUES(2`
`, NOW(), '`	`INSERT INTO comments(blog_id, dateposted, name, comment) VALUES(2, 2005-08-10 14:30:00, '`
`$_POST['name']`	`INSERT INTO comments(blog_id, dateposted, name, comment) VALUES(2, 2005-08-10, 'Bob Smith`
`', '`	`INSERT INTO comments(blog_id, dateposted, name, comment) VALUES(2, 2005-08-10, 'Bob Smith','`
`$_POST['comment']`	`INSERT INTO comments(blog_id, dateposted, name, comment) VALUES(2, 2005-08-10, 'Bob Smith', 'I really like your blog. Cool stuff!`
`');`	`INSERT INTO comments(blog_id, dateposted, name, comment) VALUES(2, 2005-08-10, 'Bob Smith', 'I really like your blog. Cool stuff!');`

The left column lists each part of the code; the right column shows how the content of the page is built up in the query. As you read the table, remember that numbers don't need single quotes around them (such as the number in `validentry`) but strings (letters and sentences) do.

One part of the code that will be new to you is `NOW()`. This is a special MySQL function that provides the current date and time, and you will use `NOW()` to automatically fill the `dateposted` field.

NOTE

Built-In MySQL Functions

MySQL provides a range of these functions, and you can explore them from the comfort of phpMyAdmin. When you insert data, a Function drop-down box lists these different MySQL functions. Experiment with them to get a better idea of what they do.

The next line in the code—`mysql_query($sql);`—performs the actual query. You may have noticed that the line does not include a variable in which to store the result, such as `$result = mysql_query($sql)`. The reason is that the query is only sent; no results are returned. The final line uses the `header()` function to redirect to the current page.

Finally, the `if` block is closed, and the `else` begins (for cases when no Submit button has been clicked). At the bottom of the page, add the closing code:

```
</table>
</form>

<?php
}
require("footer.php");
?>
```

In effect, then, the entire page of HTML is shown if the user didn't reach *viewentry.php* via clicking the Submit button (on the form on that same page!).

Building the Category Browser

Within a site powered by Blogtastic, a large number of blog entries is going to build. With so much content available, it is important to have a means of easily browsing this content. In this section, you create a useful page for users to browse the different categories and see which blog entries have been posted in each category.

If you think about how this page should be designed, it seems logical to list the categories and let the user click on one to see any related blog entries (see Figure 4-7). This functionality is similar to a tree view in a file manager: The directories are listed, and then you click one to see the files and subdirectories.

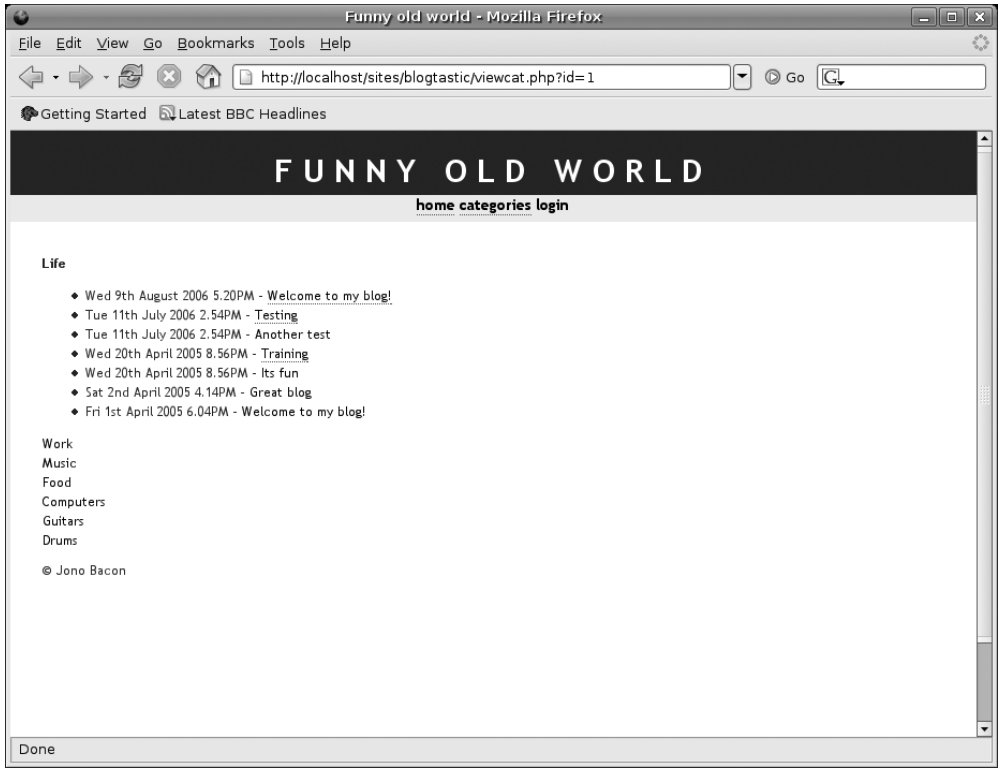


FIGURE 4-7 Click any category to view the entries in that category.

On *index.php* and *viewentry.php*, you made the category a link to a page called *viewcat.php*, and the ID of the category was passed as an `id` `GET` variable. To get started, create a new file called *viewcat.php* and add the following code:

```
require("config.php");

if(isset($_GET['id']) == TRUE) {
  if(is_numeric($id) == FALSE) {
    $error = 1;
  }
```

```
  if($error == 1) {
    header("Location: " . $config_basedir . "/viewcat.php");
  }
  else {
    $validcat = $_GET['id'];
  }
}
else {
  $validcat = 0;
}
```

This code should look familiar; it runs the `id` variable through the same validation tests used on *viewentry.php*. If no variable exists, `validcat` is set to `0`, but if the variable is indeed legitimate, `validcat` is set to the contents of the `GET` variable. If the variable fails the test to check if it is numeric, the page redirects to itself but without the `id` variable.

Select all of the records from the *categories* table:

```
else {
  $validcat = 0;
}

$sql = "SELECT * FROM categories";
$result = mysql_query($sql);

while($row = mysql_fetch_assoc($result)) {
```

Add the following code to check each row of the result set and see if $validcat is the same as the `id` variable. If it is, this means that the category is currently selected.

```
while($row = mysql_fetch_assoc($result)) {
  if($validcat == $row['id']) {
    echo "<strong>" . $row['cat'] . "</strong><br />";

    $entriessql = "SELECT * FROM entries WHERE cat_id = " . $validcat .
                  " ORDER BY dateposted DESC;";
    $entriesres = mysql_query($entriessql);
    $numrows_entries = mysql_num_rows($entriesres);

    echo "<ul>";
```

As the `while` loop iterates through each row, the first line checks if `validcat` is the same as the ID from the current row. If it is, the `if` block is executed. The first line inside the `if` outputs the name of the category in bold, instead of a link.

The query on the next line gets all blog entries in which `cat_id` is equal to `validcat`. These entries are requested in descending date order, so the most recent entry will display at the top of the list. The query is then run, and the returned rows are counted (to ensure that there are records to show). The final line starts the unordered list block that contains the results.

Check to see if any rows exist for the current category and display the relevant details:

```
    echo "<ul>";
    if($numrows_entries == 0) {
      echo "<li>No entries!</li>";
    }
    else {
      while($entriesrow = mysql_fetch_assoc($entriesres)) {
        echo "<li>" . date("D jS F Y g.iA", strtotime($entriesrow
['dateposted'])) .
          " - <a href='viewentry.php?id=" . $entriesrow['id'] . "'>" .
          $entriesrow['subject'] ."</a></li>";
      }
    }
    echo "</ul>";
  }
```

If `numrows_entries` has zero rows, the browser displays a list item with the text `No entries!`. If there are rows, another `while` loop is opened to run through the results. Inside this `while`, a list item that displays the date of the entry and a link to *viewentry.php* (using the correct `id` value) is created. The subject of the post is the body of the link.

Finally, you can display the currently unselected categories:

```
    echo "</ul>";
  }
  else {
    echo "<a href='viewcat.php?id=" . $row['id'] . "'>" . $row['cat'] .
"</a><br />";
  }
}

require("footer.php");
```

You now have a complete archive of blog entries organized by category!

Don't Just Let Anyone Log In

Everything created so far in this project has been designed to be accessible by anyone who stumbles across the blog. As such, these pages have no built-in security—that is, the pages are not restricted to certain users. Because of the open nature and accessibility of the site, it is recommended that only information suitable for public consumption is present on these pages. You should avoid adding your credit card number, personal information, or those embarrassing photos of you at a fancy dress party. (That is how rumors get started.)

Allowing restricted access for the owner to add and remove content is an essential feature, however. Having to log into phpMyAdmin to add content is not an ideal solution, so the master plan is to create pages to provide a convenient means of adding content. You need to provide a way for someone to log in, and the login details the user enters should match the ones in the *logins* table. You will use PHP sessions (covered in Chapter 2) to track the user by sharing variables across different pages. If the user successfully logs in, you can set a session variable and then check to ensure that session variable exists on the restricted pages.

To begin, create a new file called *login.php* and add the login form:

```
<form action="<?php echo $SCRIPT_NAME ?>" method="post">

<table>
<tr>
  <td>Username</td>
  <td><input type="text" name="username"></td>
</tr>
<tr>
  <td>Password</td>
  <td><input type="password" name="password"></td>
</tr>
<tr>
  <td></td>
  <td><input type="submit" name="submit" value="Login!"></td>
</tr>
</table>
</form>
```

This form contains some familiar-looking text boxes (see Figure 4-8).

You may have noticed that the second `<input>` tag uses `password` as the type. When you use this type of form element, the contents are disguised as stars or dots to hide the password from nosey onlookers.

The next step is to process the form and check if the database contains the login details. Before you do this, however, add the usual introductory code at the start of the file (before any HTML):

```
<?php

session_start();

require("config.php");

$db = mysql_connect($dbhost, $dbuser, $dbpassword);
mysql_select_db($dbdatabase, $db);
```

FIGURE 4-8 The login form looks like any other form.

NOTE

Forms Feel Insecure, Too

Although forms provide a means for people to securely identify themselves, the passwords transmitted to the server for processing are sent as plain text. This is a potential security risk inherent when using forms. The only solution to this risk is to encrypt form data with JavaScript when the form button is clicked, a technique beyond this project's scope.

Add the code that checks if the Submit button has been clicked (again, from the form you've already added):

```
mysql_select_db($dbdatabase, $db);

if($_POST['submit']) {
```

```
$sql = "SELECT * FROM logins WHERE username = '" . $_POST['username'] .
       "' AND password = '" . $_POST['password'] . "';";

$result = mysql_query($sql);
$numrows = mysql_num_rows($result);
```

The SQL statement is created to check if the `username` in the *logins* table is equal to the `username` box in the form and if the `password` field is equal to the `password` box in the form. The query is then run, and the rows are counted. The number of lines returned from the query indicates whether the details typed were correct. If the details are correct, a single row is returned—no more, no less. If no rows are returned, the details do not match.

Add the following code:

```
$numrows = mysql_num_rows($result);

if($numrows == 1) {
  $row = mysql_fetch_assoc($result);
  session_register("USERNAME");
  session_register("USERID");

  $_SESSION['USERNAME'] = $row['username'];
  $_SESSION['USERID'] = $row['id'];

  header("Location: " . $config_basedir);
}
else {
  header("Location: " . $config_basedir . "/login.php?error=1");
}
```

In the case where the login details are valid, a new session is created.

When using PHP sessions, you must register your session variables. The `session_register()` lines create two variables, called `USERNAME` and `USERID`.

NOTE

Be Consistant When Naming Variables

Naming session variables in uppercase is not mandatory, but it's useful because this helps them to stand out in your code as different types of variables.

The next two lines then use `_SESSION` (representing the user's session information) to use the variables and store information from the SQL query (the `username` and the `id`) in them. The final line performs a header redirect to *index.php*.

If the Submit button has not been clicked, a small chunk of code is run before the form displays:

```
    header("Location: " . $config_basedir . "/login.php?error=1");
  }
}
else {

  require("header.php");

  if($_GET['error']) {
    echo "Incorrect login, please try again!";
  }
?>
```

Include the *header.php* file and then check to see if there is a GET variable called `error`. If there is, the error message is displayed to indicate that the user typed an invalid username or password.

At the bottom of the page, after the HTML, add the final bits of code:

```
}
require("footer.php");
```

Signing Out the User

With the user now able to log in, you also need to give him the ability to log out—by destroying the session created on login. Create a new file called *logout.php* and add the following code:

```
<?php

session_start();
session_destroy();

require("config.php");

header("Location: " . $config_basedir);

?>
```

To log out the user, just use the `session_destroy()` function to delete all the registered session variables. The session is now destroyed, and the user is no longer logged in. You can then perform a header redirect to *index.php*.

NOTE

The Life and Death of a Session

When dealing with session-based code, you should always clear out any sessions when testing your code. Apart from creating the *logout.php* script, another option is to close the Web browser. Sessions will live only for the duration that the browser is open, and when you close the browser (not just the window), the session is lost.

When developing your code, closing your browser when you want to clear a session can be quite frustrating. To relieve the pain, use the Web Developers Toolbar extension that is available for Mozilla Firefox on all platforms. Download it from the Mozilla Extensions Web site at http://extensionroom.mozdev.org.

Adding Session Support to the Application

With the new member login capability, you can supercharge your current pages to react differently when a member is logged in. The session variables created in the login page can be checked, and you can add extra options where appropriate.

Bolting On Options in the Header File

The first file to edit is *header.php*. In *login.php* and *logout.php*, you added `session_start()` at the beginning of the page. You will use `session_start()` in most of your pages, so add it to the top of *header.php*:

```
<?php

session_start();
```

This file already contains a list of links that are available to different parts of the site. When users are logged in, the Logout link should be visible; if they are not logged in, the Login link should be visible. Add the following code inside the PHP block under the *categories* link:

```
[<a href="viewcat.php">categories</a>]

<?php

if(isset($_SESSION['USERNAME']) == TRUE) {
```

```
  echo "[<a href='logout.php'>logout</a>]";
}
else {
  echo "[<a href='login.php'>login</a>]";
}
```

The `isset()` function is used to check if the `USERNAME` session variable is set. If it is, the Logout link is displayed; otherwise, the Login link is displayed.

Use the same method for adding additional links:

```
else {
  echo "[<a href='login.php'>login</a>]";
}

if(isset($_SESSION['USERNAME']) == TRUE) {
  echo " - ";
  echo "[<a href='addentry.php'>add entry</a>]";
  echo "[<a href='addcat.php'>add category</a>]";
}

?>
```

Adding Links to Update Blog Entries

When using Blogtastic, you will need to edit existing blog entries. Instead of just adding an Edit Blog Entry link to *header.php*, it is more intuitive to add an Edit link next to blog entry subjects. (Later in the project, you will create a file, called *updatentry.php*, to edit the blog entry.) Using a similar technique of checking if the session variable exists, add the following code in *index.php*, after the category and date line:

```
echo "<i>In <a href='viewcat.php?id=" . $row['cat_id'] ."'>" .
$row['cat'] ."</a> - Posted on " . date("D jS F Y g.iA",
strtotime($row['dateposted'])) ."</i>";

if(isset($_SESSION['USERNAME']) == TRUE) {
  echo " [<a href='updateentry.php?id=" . $row['id'] . "'>edit</a>]";
}
```

The *updateentry.php* file is passed an `id` variable that contains the ID of the blog entry to edit. Copy this same block of code to *viewentry.php*, after the same line where the date of the posting is listed. The links are displayed in Figure 4-9.

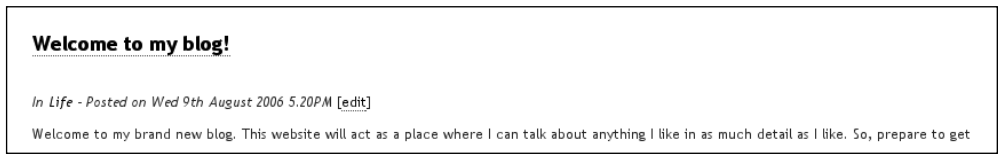


FIGURE 4-9 Adding contextual links to administer the blog makes the application easier to use.

Rolling Your Own Categories

Adding blog categories is the next part of the game. This page is similar to the page used to add comments.

First, create the form in a new file called *addcat.php*:

```
<form action="<?php echo $SCRIPT_NAME ?>" method="post">

<table>
<tr>
  <td>Category</td>
  <td><input type="text" name="cat"></td>
</tr>
<tr>
  <td></td>
  <td><input type="submit" name="submit" value="Add Entry!"></td>
</tr>
</table>
</form>
```

Add the usual lines of code at the start of the file, before any HTML:

```
<?php

session_start();

require("config.php");

$db = mysql_connect($dbhost, $dbuser, $dbpassword);
mysql_select_db($dbdatabase, $db);
```

With this page available to restricted users only, you need to check if the user is logged in. Do this by checking if one of the session variables exists; if it doesn't, redirect to another page:

```
if(isset($_SESSION['USERNAME']) == FALSE) {
  header("Location: " . $config_basedir);
}
```

NOTE

Never Assume

It might seem impossible to get to this page without clicking a link, and wouldn't that imply the user has already logged in? Although this sounds logical, someone could still type the URL directly. It's always better to explicitly check to ensure the user is logged in, rather than trust other pages to do that job for you.

Add the logic for when the user clicks the Submit button:

```
if(isset($_SESSION['USERNAME']) == FALSE) {
  header("Location: " . $config_basedir);
}

if($_POST['submit']) {
  $sql = "INSERT INTO categories(cat) VALUES('" . $_POST['cat'] . "');";
  mysql_query($sql);
  header("Location: " . $config_basedir . " viewcat.php");
}
else {
  require("header.php");

?>
```

Within this code, an `INSERT` query is created and sent to the database. After the query is run, the browser redirects to the *viewcat.php* page to view the newly created category.

Finally, close the `else` and include the *footer.php* file (all after the form):

```
<?php
}
require("footer.php");
?>
```

Creating New Blog Entries

So far in Blogtastic, the capability to actually *add* a blog entry has been suspiciously missing. This essential feature requires almost all of the techniques you've seen so far, hence the delay. You are now ready, though, so it's time to knock out this page. The page behaves in a similar way to previous pages that added content, but this page also includes a drop-down combo box that is used to select the category under which the entry is filed.

Create a new file called *addentry.php* and start the form:

```
<h1>Add new entry</h1>
<form action="<?php echo $SCRIPT_NAME ?>" method="post">

<table>
```

Previously, you added the entire form first, but in this page, the very first form control will be the special drop-down box just discussed:

```
<tr>
  <td>Category</td>
  <td>
  <select name="cat">
  <?php
    $catsql = "SELECT * FROM categories;";
    $catres = mysql_query($catsql);
    while($catrow= mysql_fetch_assoc($catres)) {
      echo "<option value='" . $catrow['id']
 . "'>" . $catrow['cat'] . "</option>";
    }
  ?>
  </select>
  </td>
</tr>
```

The drop-down combo box presents a visual box with a series of options that the user can select. This involves two basic steps. First, create a `<select>` tag that contains the items within the box. Each item is housed within `<option>` tags. In these tags, add the text that you would like to appear in the box (in this case, the category name) and a `value` attribute. This contains the value that is passed when the user selects an item. Set this attribute to contain the ID of the category item.

In terms of making this work in code, the SQL query selects everything from the *categories* table. A loop iterates through the categories that are returned in the query. Within the `while` loop, the `<option>` tags are created, and the `id` from the query is added to the `value` attribute.

Complete the rest of the form:

```
  </select>
  </td>
</tr>

<tr>
  <td>Subject</td>
  <td><input type="text" name="subject"></td>
</tr>
<tr>
  <td>Body</td>
  <td><textarea name="body" rows="10" cols="50"></textarea></td>
</tr>
<tr>
  <td></td>
  <td><input type="submit" name="submit" value="Add Entry!"></td>
</tr>
</table>
</form>
```

The form is shown in Figure 4-10.

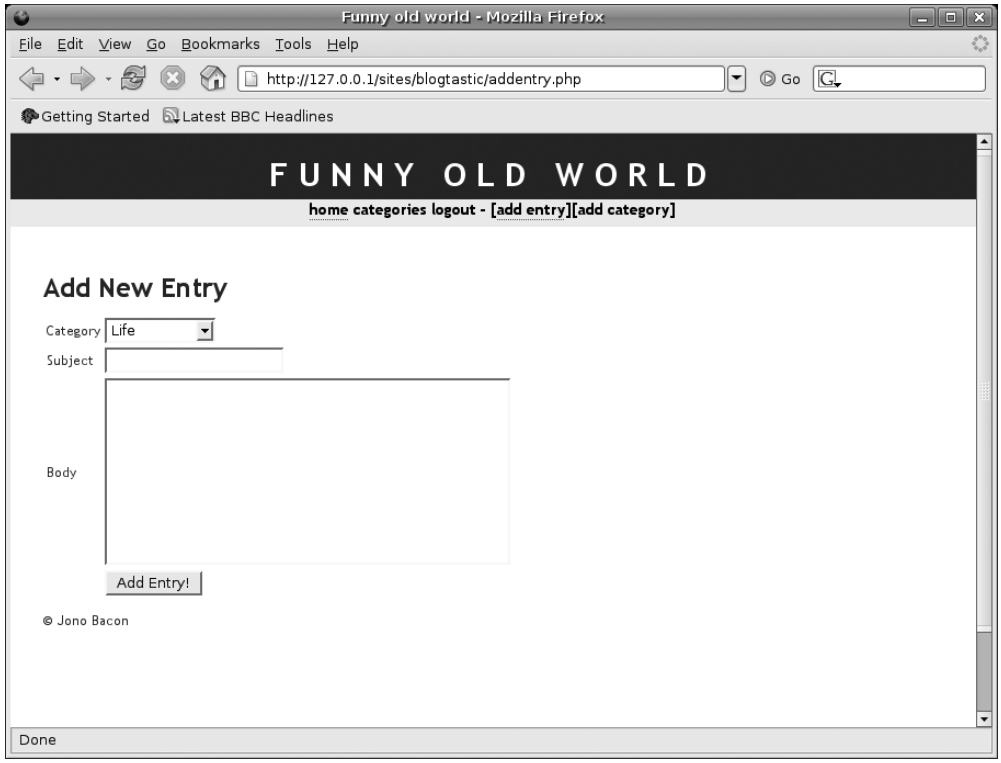


FIGURE 4-10 Adding new blog posts is simple.

Move to the beginning of the file and add the boilerplate introductory code:

```
<?php

session_start();

require("config.php");

$db = mysql_connect($dbhost, $dbuser, $dbpassword);
mysql_select_db($dbdatabase, $db);

if(isset($_SESSION['USERNAME']) == FALSE) {
  header("Location: " . $config_basedir);
}
```

Add the logic that actually processes the form:

```
if(isset($_SESSION['USERNAME']) == FALSE) {
  header("Location: " . $config_basedir);
}

if($_POST['submit']) {
  $sql = "INSERT INTO entries(cat_id, dateposted, subject, body)
VALUES(" .
         $_POST['cat'] . ", NOW(), '" . $_POST['subject'] . "', '" .
         $_POST['body'] . "');";
  mysql_query($sql);
  header("Location: " . $config_basedir);
}
else {
  require("header.php");
?>
```

This code creates an `INSERT` query that is very similar to the ones on previous form-driven pages.

Finally, close the `else` block and add the *footer.php* code:

```
</tr>
</table>
</form>

<?php
}
require("footer.php");
?>
```

Update a Blog Entry

The final page is for updating blog entries. Earlier, when you added session support to Blogtastic, you went through some of the pages and added links to edit a particular blog entry. The link to edit blog entries was for a page called *updateentry.php*, which is passed an `id` variable. This ID indicates which record to update.

Instead of adding the form first, on this page you will work from the top to the bottom.

First, add the boilerplate code:

```
<?php

session_start();

require("config.php");
```

```
if(isset($_SESSION['USERNAME']) == FALSE) {
  header("Location: " . $config_basedir);
}

$db = mysql_connect($dbhost, $dbuser, $dbpassword);
mysql_select_db($dbdatabase, $db);
```

The next block of code is identical to the validation code written earlier:

```
if(isset($_GET['id']) == TRUE) {
  if(is_numeric($id) == FALSE) {
    $error = 1;
  }

  if($error == 1) {
    header("Location: " . $config_basedir);
  }
  else {
    $validentry = $_GET['id'];
  }
}
else {
  $validentry = 0;
}
```

Add the code to process the form:

```
else {
  $validentry = 0;
}

if($_POST['submit']) {
  $sql = "UPDATE entries SET cat_id = "
 . $_POST['cat'] . ", subject = '" .
        $_POST['subject'] ."', body = '"
 . $_POST['body'] . "' WHERE id = " .
        $validentry . ";";
  mysql_query($sql);

  header("Location: " . $config_basedir . "/viewentry.php?id=" .
$validentry);
}
```

The SQL query implements an `UPDATE` command that will update each field in the database that has the id of `validentry` (the validated `id` variable). The `UPDATE` query indicates which table to update (`UPDATE` *entries*) and then provides a number of *database field = form element* sections. When the query is complete, another header redirect takes the user to the *viewentry.php* page with the correct `id` variable.

If the Submit button has not been clicked, the details of the entry are grabbed from MySQL so you can populate the form fields, starting with a query:

```
  header("Location: " . $config_basedir
 . "/viewentry.php?id=" . $validentry);
}
else {

  require("header.php");

  $fillsql = "SELECT * FROM entries WHERE id = " . $validentry . ";";
  $fillres = mysql_query($fillsql);
  $fillrow = mysql_fetch_assoc($fillres);

?>
```

Next, begin creating the form:

```
  $fillrow = mysql_fetch_assoc($fillres);

?>

<h1>Update entry</h1>

<form action="<?php echo $SCRIPT_NAME . "?id="
 . $validentry; ?>" method="post">

<table>
```

The first part of the form is the `category` field. You will need to have the chosen category automatically selected when the page is loaded. To do this, add `selected` at the end of the tag to be selected. An example of this in HTML is shown here (this is not actually in the project code, so don't add it):

```
<select name="example">
  <option value="1">Option 1</option>
  <option value="2" selected>Option 2</option>
  <option value="3">Option 3</option>
</select>
```

To accomplish this, add the following code to your form:

```
<form action="<?php echo $SCRIPT_NAME . "?id=" . $validentry; ?>"
method="post">

<table>

<tr>
  <td>Category</td>
  <td>
```

```
  <select name="cat">
  <?php
    $catsql = "SELECT * FROM categories;";
    $catres = mysql_query($catsql);
    while($catrow= mysql_fetch_assoc($catres)) {
      echo "<option value='" . $catrow['id'] . "'";

      if($catrow['id'] == $fillrow['cat_id']) {
        echo " selected";
      }

      echo ">" . $catrow['cat'] . "</option>";
    }
  ?>
  </select>
  </td>
</tr>
```

The query is run, and then the `while` loop iterates through each record. Inside the `while` loop, the `<option value=<id from the record>` is first printed and then a check is made to see if the category ID of the entry (`fillrow['cat_id']`) is the same as the current category row ID (`catrow['id']`). If the values match, `" selected"` (notice the space before the word) is added. After this, the rest of the line is created: `>category</option>`.

In the remaining parts of the form, small PHP blocks add the information from the query to the value attributes and between the `<textarea>` tags to populate the form:

```
  </select>
  </td>
</tr>

<tr>
  <td>Subject</td>
  <td><input type="text" name="subject"
 value="<?php echo $fillrow['subject']; ?>">
</td>
</tr>
<tr>
  <td>Body</td>
  <td><textarea name="body" rows="10" cols="50">
    <?php echo $fillrow['body']; ?></textarea></td>
</tr>
<tr>
  <td></td>
  <td><input type="submit" name="submit" value="Update Entry!"></td>
</tr>
</table>
</form>
```

Finally, close `else` and insert the footer:

```
<?php
}
require("footer.php");
?>
```

You can see the updated page in Figure 4-11.

FIGURE 4-11 Updating blog entries uses a similar interface to adding new entries.

SUMMARY

In this project, you created your first full-featured, database-driven Web application. This application flexed your programming muscles, and covered an entire range of essential techniques. This included using database queries, adding data to the database, joining tables, updating records, performing validation, managing archived data, separating code across different pages, and ensuring interface usability.

Aside from providing a fun project to work on, this project also provided a base in which the rest of the projects in the book are based upon. You learned a number of skills that will be refined and built upon as you continue though the book. This is the start of an exciting journey, and reading this means that you have completed a large and important step. Stretch your fingers, dust off your keyboard, grab a cup of something hot, and get ready for the next project.