CHAPTER 5

Building the Entry Manager

At this point, you know enough to start building your blog! In this chapter, I'll walk you through how to build the backbone of your blogging application. The pieces you'll build include:

- A form to accept entry input
- A script to handle input from the form
- A script to save the entry in the database
- A script to retrieve the entry from the database
- An HTML document to display the retrieved information

By the end of this chapter, you will have a very basic, but fully functional blogging system.

Planning the Entry Database Table

One of the most important steps with any new application is the planning of the tables that will hold data. This has a huge impact on the ease of *scaling* our application later. Scaling is the expansion of an application to handle more information and/or users, and it can be a tremendous pain if we don't look ahead when starting a new project.

At first, your blog needs to store several types of entry information to function:

- Unique ID
- Entry title
- Entry text
- Date created

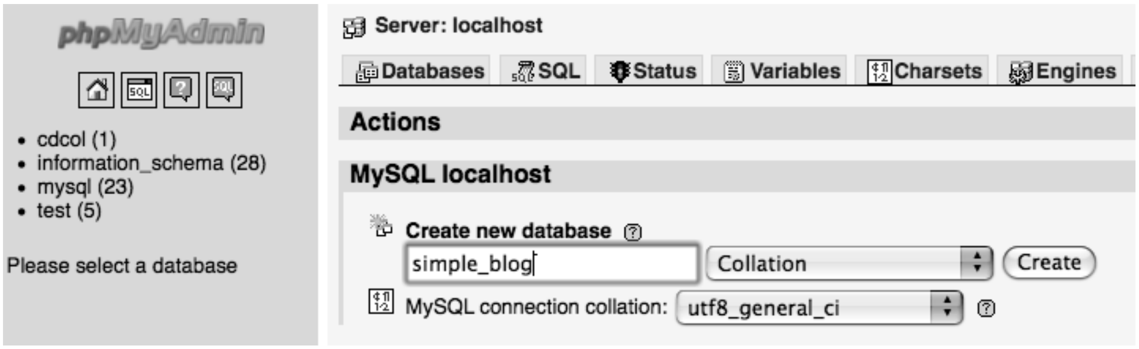
Using a unique ID for each entry in the entry table enables you to access the information contained with just a number. This is extremely helpful for data access, especially if this dataset changes in the future (if you add an "imageURL" column to the table, for example).

The first step is to determine the fields you will need for the `entries` table. Your table needs to define what type of information is stored in each column, so let's take a quick look at the information each column needs to store:

- `id`: *A unique number identifying the entry.* This will be a positive integer, and it makes sense for this number to increment automatically because that ensures the number is unique. You will also use this as the primary method for accessing an entry, so it will double as the primary key for the table.
- `title`: An alphanumeric string that should be relatively short. You'll limit the string to 150 characters.

- `entry`: *An alphanumeric string of indeterminate length.* You won't limit the length of this field (within reason).
- `created`: *The timestamp generated automatically at the original creation date of the entry.* You'll use this to sort your entries chronologically, as well as for letting tour users know when an entry was posted originally.

Now it's time to create the database. Do this by navigating to `http://localhost/phpmyadmin/` and creating a database called `simple_blog` using the "Create new database" field on the homepage (see Figure 5-1).

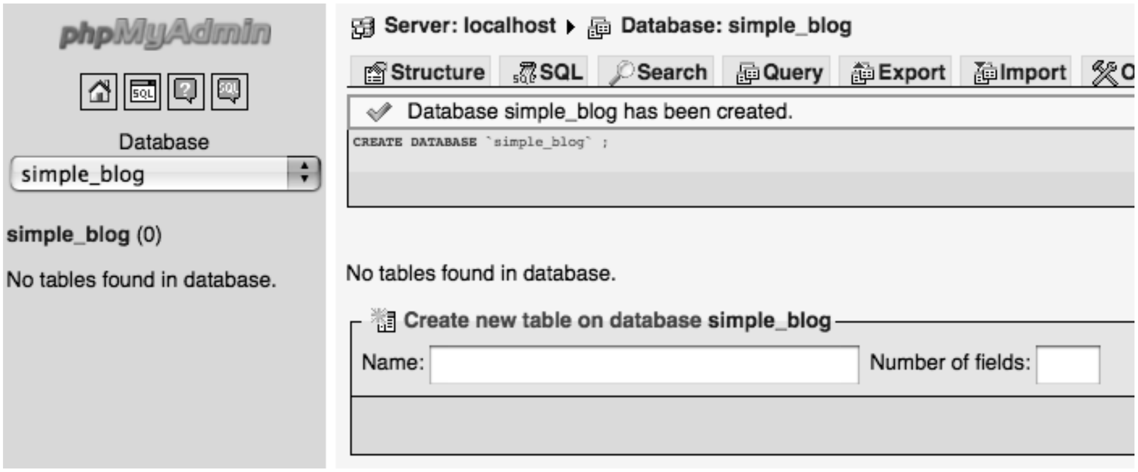


simple_blog

MySQL connection collation: utf8_general_ci

Figure 5-1. *Creating a new database in phpMyAdmin*

The new database is created after you click "Create." Next, you're shown a confirmation message and given options for interacting with your new database (see Figure 5-2).



Database simple_blog has been created.

```
CREATE DATABASE `simple_blog` ;
```

No tables found in database.

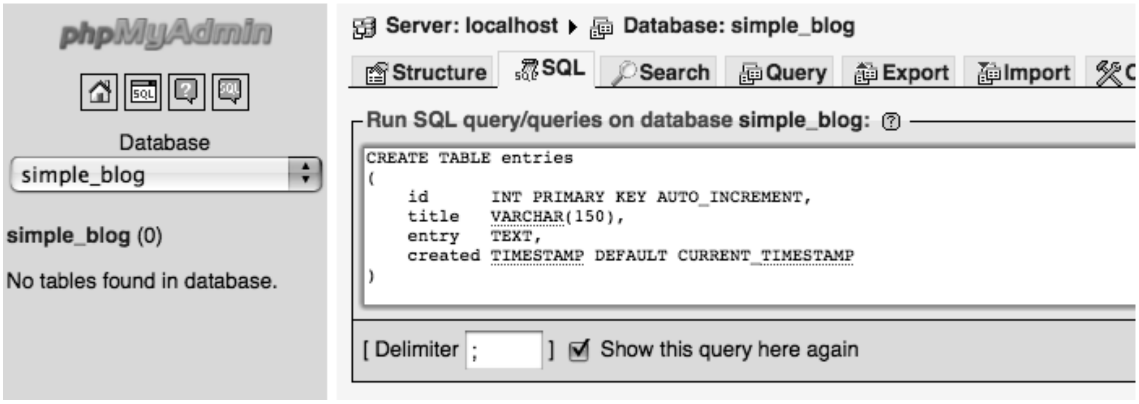
Create new table on database simple_blog

Figure 5-2. *The simple_blog database confirmation screen and options*

The next step is to write the code that creates your `entries` table:

```
CREATE TABLE entries
(
    id      INT PRIMARY KEY AUTO_INCREMENT,
    title   VARCHAR(150),
    entry   TEXT,
    created TIMESTAMP DEFAULT CURRENT_TIMESTAMP
)
```

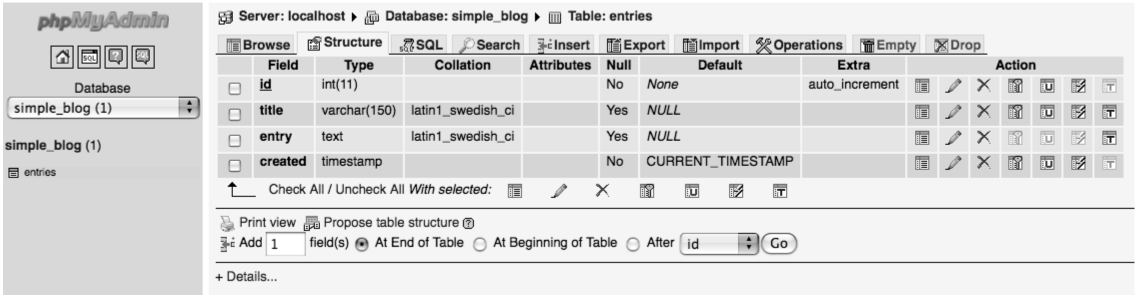
To create the `entries` table, click the SQL tab at the top of the page and enter the command that creates your table (see Figure 5-3).


phpMyAdmin
Database
simple_blog
simple_blog (0)
No tables found in database.
Server: localhost ▸ Database: simple_blog
Structure
SQL
Search
Query
Export
Import
Run SQL query/queries on database simple_blog:
[Delimiter ;]
Show this query here again

```
CREATE TABLE entries
(
    id      INT PRIMARY KEY AUTO_INCREMENT,
    title   VARCHAR(150),
    entry   TEXT,
    created TIMESTAMP DEFAULT CURRENT_TIMESTAMP
)
```

Figure 5-3. *Creating the* `entries` *table in phpMyAdmin*

After you click the Go button at the bottom right of the SQL text field, the `entries` table shows up in the left-hand column of the screen. You can see a table's structure by clicking the table you're interested in (see Figure 5-4).


phpMyAdmin
Database
simple_blog (1)
simple_blog (1)
entries
Server: localhost ▸ Database: simple_blog ▸ Table: entries
Browse
Structure
SQL
Search
Insert
Export
Import
Operations
Empty
Drop
Check All / Uncheck All With selected:
Print view
Propose table structure
Add 1 field(s) [✓ At End of Table] At Beginning of Table After id Go
+ Details...

	Field	Type	Collation	Attributes	Null	Default	Extra	Action
☐	id	int(11)			No	*None*	auto_increment	
☐	title	varchar(150)	latin1_swedish_ci		Yes	*NULL*		
☐	entry	text	latin1_swedish_ci		Yes	*NULL*		
☐	created	timestamp			No	CURRENT_TIMESTAMP		

Figure 5-4. *The structure of the* `entries` *table in phpMyAdmin*

At this point, you have created the `entries` table, and you're ready to create your input form.

Creating the Entry Input Form

Storing entries in your database requires that you allow site administrators to enter data via a web form. Before you can do this, you need to identify which fields the form must include.

Two of your fields are populated automatically when an entry is created: the id field will generate an automatically incremented number to identify the entry, and the created field automatically stores the timestamp for the entry's creation date. All you need to include are fields to enter the title and entry fields.

Your title field has a maximum length of 150 characters, so you use an input tag with a maxlength attribute for this. The entry field can be as long as you want it to, so create a textarea tag.

In Eclipse, create a new file in the simple_blog project called admin.php; this file should end up saved in the xampp folder at /xampp/htdocs/simple_blog/admin.php).

In the new file, add the following HTML:

```
<!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">

<head>
    <meta http-equiv="Content-Type"
        content="text/html;charset=utf-8" />
    <title> Simple Blog </title>
</head>

<body>
    <h1> Simple Blog Application </h1>

    <form method="post" action="inc/update.inc.php">
        <fieldset>
            <legend>New Entry Submission</legend>
            <label>Title
                <input type="text" name="title" maxlength="150" />
            </label>
            <label>Entry
                <textarea name="entry" cols="45" rows="10"></textarea>
            </label>
            <input type="submit" name="submit" value="Save Entry" />
            <input type="submit" name="submit" value="Cancel" />
        </fieldset>
    </form>
</body>

</html>
```

You can view the form you've created by navigating to `http://localhost/simple_blog/admin.php` in your browser (see Figure 5-5).

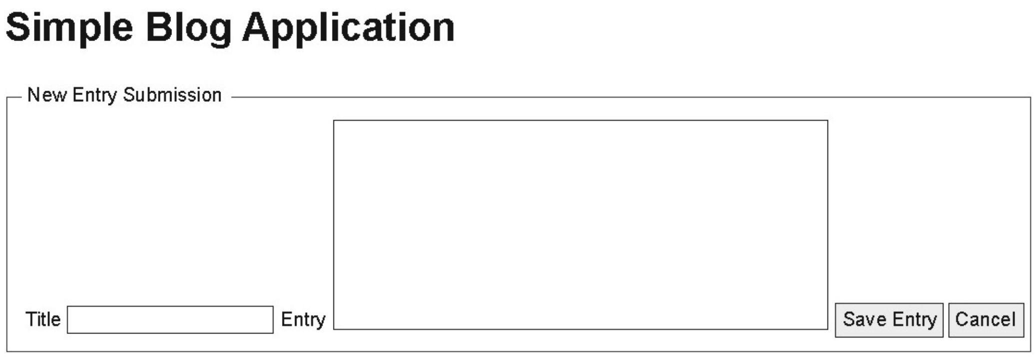


Figure 5-5. *The entry creation form*

This is not a book about *Cascading Style Sheets* (CSS), a language used to describe the presentation of HTML- and XML-based documents, but you'll take advantage of CSS to make your form easy-to-use. Begin by creating a new folder in the `simple_blog` project called `css`. Next, create a file called `default.css` in the css folder (full path: `/xampp/htdocs/simple_blog/css/default.css`), then add the following style information to your new file:

```
h1 {
    width:380px;
    margin:0 auto 20px;
    padding:0;
    font-family:helvetica, sans-serif;
}
ul#menu {
        width:350px;
        margin:0 auto;
        padding:0;
        list-style:none;
}
ul#menu > li {
        display:inline;
}
ul#menu > li > a {
        padding:6px;
        color:#FFF;
        background:#333;
        font-family:helvetica, sans-serif;
        text-decoration:none;
}
```

```
#control_panel, #entries, fieldset {
    width:350px;
    margin:0 auto;
    padding:10px;
    font-family:helvetica, sans-serif;
}
#control_panel {
    width:350px;
    margin:20px auto 0;
    padding:4px;
    font-size:80%;
    text-align:center;
    background:#DDD;
    border-top:1px dotted #000;
    border-bottom:1px dotted #000;
}
input, textarea {
    font-size:95%;
    font-family:helvetica, sans-serif;
    display:block;
    width:340px;
    margin:0 auto 10px;
    padding:4px;
    border:1px solid #333;
}
input[type=submit] {
    display:inline;
    width:auto;
}
input[type=hidden] {
        display:none;
}
.backlink {
    border:0;
    text-align:right;
}
#comment-form > fieldset {
  width:330px;
  border:1px solid #000;
}
#comment-form input[type=text],
#comment-form textarea {
  width:320px;
}
```

```
.error {
  color:#F00;
  text-align:center;
  font-weight:bold;
  margin:5px 0 15px;
}
.comment {
  padding:0 0 10px;
}
.comment > span,
.comment > .admin {
  display:block;
  font-size:80%;
  margin:0 0 10px;
  padding:4px;
  text-align:right;
  background:#DDD;
  border-bottom:1px dotted #000;
}
.comment > .admin {
  background:transparent;
  border:0;
}
.comment > span > strong {
  float:left;
}
```

I won't go into the specifics of how this works, but this code will provide styling information for all the components I'll be building in this book. To apply these styles to your form, you need to link to the CSS file in the `head` section of `admin.php`. Do this by adding the code highlighted in bold to `admin.php`:

```
<head>
    <meta http-equiv="Content-Type"
        content="text/html;charset=utf-8" />
    <link rel="stylesheet" href="/css/default.css" type="text/css" />
    <title> Simple Blog </title>
</head>
```

Once you save the linked stylesheet in `admin.php`, you can reload `admin.php` to see the cleaned up form (see Figure 5-6).

■**Note** You can learn more about CSS by checking out the book, *Beginning CSS Web Development: From Novice to Professional*, by Simon Collison (Apress, 2006).

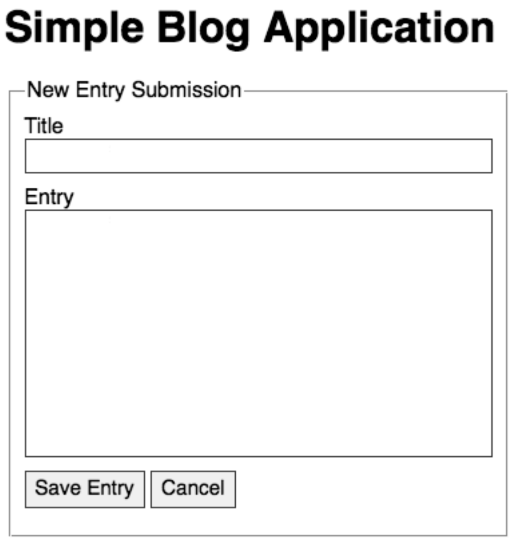


Figure 5-6. *The input-manager form styled with CSS*

Create a Script to Process the Form Input

Your entry form is set to submit entered values using the POST method to a file located at `inc/update.inc.php`. The next step is to create the file that will accept the input from the form and save entries to the database.

First, you need to create the `inc` folder. You create a folder for this script because it won't be accessed directly by a browser.

■**Tip** To keep our project organized, you can separate scripts that process information from scripts that display it. This makes maintenance a little easier because it groups similar files.

In your `simple_blog` project, create the `inc` folder, then create a file called `update.inc.php`. This script will have logic that determines whether input should be saved; it will also have the ability to save entries to the `entries` table.

■ **Tip** Be sure to save files that aren't accessed directly by the browser with a different file extension, such as `.inc.php`; this helps you identify files that should not be public easily.

It is critical that you plan your script that processes form input properly; a good way to do that is to break the process into small, discrete steps:

1. Verify that information was submitted via the POST method
2. Verify that the Save Entry button was pressed
3. Verify that both the title and entry form fields were filled out
4. Connect to the database
5. Formulate a MySQL query to store the entry data
6. Sanitize the input and store it in the entries table
7. Obtain the unique ID for the newly created entry
8. Send the user to the newly created entry

Performing the Initial Verification

You can combine the first three steps into one conditional statement. All conditions are required, so you can use the && operator to require that all conditions are true. The conditional statement looks like this:

```
<?php

if($_SERVER['REQUEST_METHOD']=='POST'
    && $_POST['submit']=='Save Entry'
    && !empty($_POST['title'])
    && !empty($_POST['entry']))
{
    // Continue processing information . . .
}

// If both conditions aren't met, sends the user back to the main page
else
{
    header('Location: ../admin.php');
    exit;
}

?>
```

You use the `$_SERVER` superglobal to determine whether the script was accessed using the `POST` method. Making this check helps you ensure that the page wasn't accessed by mistake. You use the `$_POST` superglobal to access the value of the button pressed to submit the form. If the pressed button wasn't the "Save Entry" button, the form isn't submitted. This makes it possible for the Cancel button to send the user back to the main page without saving any of the input from the form. Finally, you use the `$_POST` superglobal to verify that the user filled out the title and entry fields of the form; performing this check helps you ensure that you don't store any incomplete entries in the database.

If any of these conditions isn't met, the user is sent back to the main page, and your script performs no further processing. This means that any information submitted won't be saved to the database.

Connect to the Database

If all conditions were met, the script can proceed to Step 4, where you save the information to your database. You need to open a connection to the database before you can save to it; you open the connection using PHP Data Objects (PDO).

Keeping Database Credentials Separate

It's a good habit to keep database credentials and other site-wide information separate from the rest of your scripts. The reason: This allows you to change an entire project's configuration quickly and easily by altering a single file.

You might wonder why skipping this step could matter. Imagine that you build a project that has dozens of scripts, all of which need to contact the database for some reason or another. Now imagine that the database is moved to a new server, and the login credentials need to be updated. If you did not keep site-wide information separate from the rest of your scripts in this scenario, you would be required to open every single file in your project to swap in the new login information—this would be a tedious and potentially time-consuming task.

If, however, you store all the login credentials and other scripts that access the database in one file, you're able to move the site to a new database by altering a single file.

You store your database credentials in a file you create and store in the `inc` folder called `db.inc.php` (full path: `/xampp/htdocs/simple_blog/inc/db.inc.php`). You can define the credentials as constants with the following code:

```
<?php

define('DB_INFO', 'mysql:host=localhost;dbname=simple_blog');
define('DB_USER', 'root');
define('DB_PASS', '');

?>
```

All that remains is to include `db.inc.php` in any file that needs database access, and you have access to your credentials.

Connecting to the Database in update.inc.php

Next, add the bolded lines to `update.inc.php` to include your credentials and open a connection to the database:

```
<?php

if($_SERVER['REQUEST_METHOD']=='POST'
    && $_POST['submit']=='Save Entry')
{
    // Include database credentials and connect to the database
    include_once 'db.inc.php';
    $db = new PDO(DB_INFO, DB_USER, DB_PASS);

    // Continue processing data...
}

// If both conditions aren't met, send the user back to the main page
else
{
    header('Location: ../admin.php');
    exit;
}

?>
```

Save the Entry to the Database

When you're sure that all the necessary conditions have been met and a connection to the database is open, you're ready to proceed with Steps 5 and 6: formulating a MySQL query to store the entry data and then sanitizing the input and storing it in the `entries` table. To accomplish these tasks, you need to create a prepared statement. Begin by creating a query template, which you use to save the title and entry fields entered to the `title` and `entry` columns in the `entries` table. The query looks like this:

```
INSERT INTO entries (title, entry) VALUES (?, ?)
```

You store this query in a variable that you pass to PDO's `prepare()` method. With your query prepared, you can execute the statement using the supplied form information, confident that the input is being escaped properly.

Add the code in bold to `update.inc.php`:

```
<?php

if($_SERVER['REQUEST_METHOD']=='POST'
    && $_POST['submit']=='Save Entry')
{
    // Include database credentials and connect to the database
    include_once 'db.inc.php';
    $db = new PDO(DB_INFO, DB_USER, DB_PASS);
    // Save the entry into the database
    $sql = "INSERT INTO entries (title, entry) VALUES (?, ?)";
    $stmt = $db->prepare($sql);
    $stmt->execute(array($title, $entry));
    $stmt->closeCursor();

    // Continue processing data...
}

// If both conditions aren't met, sends the user back to the main page
else
{
    header('Location: ../admin.php');
    exit;
}

?>
```

The `execute()` method saves the information into the `entries` table. Finally, call the `closeCursor()` method to end the query.

Retrieve the Entry's Unique ID and Display the Entry to the User

You've saved your new entry successfully; the final pair of steps is to obtain the unique ID of the new entry and enable the user to view his new entry.

To accomplish this, you need the ID generated for the entry you just saved. Fortunately, MySQL provides a built-in function for tackling the first part of this; you can use the `LAST_INSERT_ID()` function to structure a query that retrieves the unique ID of the new entry:

```
SELECT LAST_INSERT_ID()
```

When you access the results of the query using the `fetch()` method, you're given an array in which the first index (`0`) contains the ID of the last entry inserted into the database.

Once you have the ID, you want to send the user to the publicly displayed page that contains his entry, which you call `index.php`. To do this, you need to insert the `id` of the entry you want to display in a URL:

```
http://localhost/simple_blog/index.php?id=1
```

You can shorten the URL like this:

```
http://localhost/simple_blog/?id=1
```

■ **Tip** This script uses *relative paths* to access the publicly displayed site. This approach allows the scripts to exist in any directory, as long as they remain in the same relationship to each other within the file structure. The relative path `../` means, in plain English: “Go up one folder.” In this case, the relative path takes you out of the `inc` folder and back into the `simple_blog` folder.

Now add the following code to `update.inc.php` to retrieve the entry’s ID and direct the user to the entry’s public display:

```
<?php

if($_SERVER['REQUEST_METHOD']=='POST'
    && $_POST['submit']=='Save Entry')
{
    // Include database credentials and connect to the database
    include_once 'db.inc.php';
    $db = new PDO(DB_INFO, DB_USER, DB_PASS);

    // Save the entry into the database
    $sql = "INSERT INTO entries (title, entry) VALUES (?, ?)";
    $stmt = $db->prepare($sql);
    $stmt->execute(array($_POST['title'], $_POST['entry']));
    $stmt->closeCursor();

    // Get the ID of the entry we just saved
    $id_obj = $db->query("SELECT LAST_INSERT_ID()");
    $id = $id_obj->fetch();
    $id_obj->closeCursor();

    // Send the user to the new entry
    header('Location: ../admin.php?id='.$id[0]);
    exit;
}

// If both conditions aren't met, sends the user back to the main page
else
{
    header('Location: ../admin.php');
    exit;
}

?>
```

Note You haven't created `index.php` yet, so this code redirects to `admin.php`. You'll change this when you create `index.php` in the next step.

No matter how the script is accessed, the user will receive a resolution: either the script executes successfully and the user is shown her new entry, or the script takes her back out to the main display and nothing is saved.

You can test the new system by adding three dummy entries to the system:

- Title: *First Entry*; Entry: *This is some text.*
- Title: *Second Entry*; Entry: *More text and a <a href="#">link</a>*.
- Title: *Third Entry*; Entry: *A third entry in the database*.

These entries will give you some test data to work with when you move on to the next step, which is to build the script that retrieves entries from the database and displays them.

Displaying the Saved Entries

As I stated earlier, you will call your script to display the entries `index.php`, and you will store it in the root of the `simple_blog` project (full path: `/xampp/htdocs/simple_blog/index.php`). Your first step is to put together the structure of the page that will display the information.
Add the following HTML to `index.php`:

```
<!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">

<head>
    <meta http-equiv="Content-Type"
        content="text/html;charset=utf-8" />
    <link rel="stylesheet" href="/css/default.css" type="text/css" />
    <title> Simple Blog </title>
</head>

<body>

    <h1> Simple Blog Application </h1>

    <div id="entries">
```

```
<?php

    // Format the entries from the database

?>
        <p class="backlink">
            <a href="/admin.php">Post a New Entry</a>
        </p>

    </div>

</body>

</html>
```

This code creates a valid HTML page, complete with a link to the CSS file you created earlier this chapter. It also creates a page heading (“Simple Blog Application”), a container for the entries, and a link to `admin.php`, which you can use to create additional entries.

Planning Our Scripts

You want your script to scale to your future needs, so now is a good time to explore best practices, organizational techniques, and other ways to eliminate unnecessary rewrites as you modify your code going forward. This might seem like a bit of a departure from the simple blogging application, but taking a moment to learn how to separate your code properly now can save you a lot of time in the future (as you’ll see in upcoming chapters).

Separation of Logic in Programming

When you write scripts, it’s important to know what your script is doing and why. This helps you separate different parts of your script into smaller chunks with specific purposes, which simplifies the organization, maintenance, and readability of your applicatoin’s code.

To separate your code, you need to identify the different categories that scripts can fall into. In general, there are three main types of coding logic:

- Database logic
- Business logic
- Presentation logic

Database Logic

Database logic refers to any code that connects to the database, whether that code creates, modifies, or retrieves data.

Business Logic

Business logic is a much broader area of coding, and it includes any script that *processes* data. Business logic typically exists between the database and presentation logic, and it serves to modify the data in some capacity that doesn't involve specific output properties. For example, you might use business logic to replace words in some text, convert a timestamp to a date, and so on.

Business logic represents a large area of coding, so it can be tough to describe it precisely. However, it's generally safe to assume that if a given bit of code doesn't access the database or generate the presentation (such as HTML markup), it's probably performing business logic.

Presentation Logic

Presentation logic is the part of a script that displays output to a user. This code can include generating HTML markup, XML output, or any other format that allows a user to access the information a script is working with.

Organizational Philosophies and Programming Patterns

There isn't a hard and fast "right way" to separate logic in an application, and the methods a developer chooses for a given project are determined largely by her preferences. However, there are several popular *programming patterns*, or general philosophies, you can choose from when deciding how to structure an application.

A couple of the more popular patterns are the Multitier Architecture pattern and the Model-View-Controller (MVC) pattern—you can learn more about the MVC pattern at `http://en.wikipedia.org/wiki/Model-View-Controller`. The simple blog application relies on the Multitier Architecture pattern—you can more about the basics of this pattern in a great article on Wikipedia at `http://en.wikipedia.org/wiki/Multitier_architecture`.

Mapping Your Functions to Output Saved Entries

Planning the necessary steps of your script can help you identify the different types of logic involved, enabling you to group different steps into their respective functional categories.

Your script needs to allow users to see a list of entry titles if no entry is selected; it also needs to let users see a full entry if an entry ID is supplied. To accomplish this, your script needs to accomplish several tasks:

- Connect to the database
- Retrieve all entry titles and IDs if no entry ID was supplied
- Retrieve an entry title and entry if an ID was supplied
- Sanitize the data to prepare it for display
- Present a list of linked entry titles if no entry ID was supplied
- Present the entry title and entry if an ID was supplied

If you look at what each step is doing, you can assign each step to a *database, business,* or *presentation* layer. For example, a simple breakdown of tasks might look like this:

Database layer

- Connect to the database
- Retrieve all entry titles and IDs if no entry ID was supplied
- Retrieve an entry title and entry if an ID was supplied

Business layer

- Sanitize the data to prepare it for display

Presentation layer

- Present a list of linked entry titles if no entry ID was supplied
- Present the entry title and entry if an ID was supplied

A good way to approach this problem is to separate your tasks into a database function called `retrieveEntries()`, a business function called `sanitizeData()`, and the presentation logic, which you will store in `index.php`.

You can reinforce these logical separations by defining your database and business functions in a separate file, which you'll call `functions.inc.php` and create in the `inc` folder. This makes your functions accessible to other pages in your application, should that become necessary in the future.

Writing the Database Functions

Begin by creating the file that will contain your functions. In the `inc` folder, create a new file called `functions.inc.php` (full path: `/xampp/htdocs/simple_blog/inc/functions.inc.php`).

In your new file, define tour database function, `retrieveEntries()`. This function accepts two parameters: your database connection and an optional parameter for the entry ID. Your defined function should look like this:

```
<?php

function retrieveEntries($db, $id=NULL)
{
    // Get entries from database
}

?>
```

You declare the default value for `$id` to be `NULL`; doing this means you can omit it without causing an error. Before you design your database query, you must determine whether an `entry ID` was passed; if so, you need to retrieve different information. The default value is `NULL`, so you simply need to check whether `$id` is `NULL`. Do this by adding the code in bold to `retrieveEntries()`:

```
<?php

function retrieveEntries($db, $id=NULL)
{
    /*
     * If an entry ID was supplied, load the associated entry
     */
    if(isset($id))
    {
        // Load specified entry
    }

    /*
     * If no entry ID was supplied, load all entry titles
     */
    else
    {
        // Load all entry titles
    }

    // Return loaded data
}

?>
```

Your next step is to write a script that executes if no entry ID is supplied. Your database query needs to retrieve two pieces of information from the entries table: the `id` and `title` fields. You need to store this information so it can be returned from the `retrieve_entries()` function and used by your business and presentation layers. A function can only return one variable, so you need to store the entry information in an array.

The query to retrieve the necessary information looks like this:

```
SELECT id, title
FROM entries
ORDER BY created DESC
```

There aren't any user-supplied parameters in the query, so you don't need to prepare the statement. This means you can execute the query immediately and loop through the results, storing the `id` and `title` in a multidimensional array.

Add the lines in bold to `functions.inc.php`:

```
<?php

function retrieveEntries($db, $id=NULL)
{
    /*
     * If an entry ID was supplied, load the associated entry
     */
    if(isset($id))
    {
        // Load specified entry
    }

    /*
     * If no entry ID was supplied, load all entry titles
     */
    else
    {
        $sql = "SELECT id, title
                FROM entries
                ORDER BY created DESC";
        // Loop through returned results and store as an array
        foreach($db->query($sql) as $row) {
            $e[] = array(
                'id' => $row['id'],
                'title' => $row['title']
            );
        }

        // Set the fulldisp flag for multiple entries
        $fulldisp = 0;
    }

    // Return loaded data
}

?>
```

If no entry ID is supplied, your script now loads all entry titles and IDs into an array called $e; your script also sets a flag called `$fulldisp` to `0`, which tells your presentation layer that the supplied information is *not* for full display.

As a safeguard, you should set some default values in the event that no entries come back from the entries table. To do this, you check whether the $e variable is an array. If it isn't, you know that no entries were returned, and you can create a default entry in $e and set the $fulldisp flag to 1, which signifies that your default entry should be displayed as a full entry.

Again, add the code highlighted in bold code to functions.inc.php:

```
<?php

function retrieveEntries($db, $id=NULL)
{
    /*
     * If an entry ID was supplied, load the associated entry
     */
    if(isset($id))
    {
        // Load specified entry
    }

    /*
     * If no entry ID was supplied, load all entry titles
     */
    else
    {
        $sql = "SELECT id, title
                FROM entries
                ORDER BY created DESC";

        // Loop through returned results and store as an array
        foreach($db->query($sql) as $row) {
            $e[] = array(
                'id' => $row['id'],
                'title' => $row['title']
            );
        }

        // Set the fulldisp flag for multiple entries
        $fulldisp = 0;

        /*
         * If no entries were returned, display a default
         * message and set the fulldisp flag to display a
         * single entry
         */
        if(!is_array($e))
        {
            $fulldisp = 1;
            $e = array(
                'title' => 'No Entries Yet',
                'entry' => '<a href="/admin.php">Post an entry!</a>'
            );
        }
    }

    // Return loaded data
}

?>
```

You can now run your function safely without an error, so long as no entry ID is supplied. Next, you need to modify the script so it retrieves an entry if an ID is supplied.

This code needs to use the supplied ID in a query to retrieve the associated entry `title` and `entry` fields. As before, you store the returned data in an array called `$e`.

Add the code in bold to `functions.inc.php`:

```
<?php

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

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

        // Set the fulldisp flag for a single entry
        $fulldisp = 1;
    }
    /*
     * If no entry ID was supplied, load all entry titles
     */
    else
    {
        $sql = "SELECT id, title
                FROM entries
                ORDER BY created DESC";

        // Loop through returned results and store as an array
        foreach($db->query($sql) as $row) {
            $e[] = array(
                'id' => $row['id'],
                'title' => $row['title']
            );
        }

        // Set the fulldisp flag for multiple entries
        $fulldisp = 0;

        /*
         * If no entries were returned, display a default
         * message and set the fulldisp flag to display a
         * single entry
         */
        if(!is_array($e))
        {
            $fulldisp = 1;
            $e = array(
                'title' => 'No Entries Yet',
                'entry' => '<a href="/admin.php">Post an entry!</a>'
            );
        }
    }

    // Return loaded data
}

?>
```

At this point, your function has two variables: `$e` and `$fulldisp`. Both variables must be returned from the function for further processing; however, a function can return only one value, so you need to somehow combine these variables into a single variable.

You do this using a function called `array_push()`, which adds a value to the end of an array. Using this function, you can add the value of `$fulldisp` to the end of `$e` and return `$e`.

You can accomplish this by adding the code in bold to functions.inc.php:

```
<?php

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

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

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

    /*
     * If no entry ID was supplied, load all entry titles
     */
    else
    {
        $sql = "SELECT id, title
                FROM entries
                ORDER BY created DESC";

        // Loop through returned results and store as an array
        foreach($db->query($sql) as $row) {
            $e[] = array(
                'id' => $row['id'],
                'title' => $row['title']
            );
        }

        // Set the fulldisp flag for multiple entries
        $fulldisp = 0;
        /*
         * If no entries were returned, display a default
         * message and set the fulldisp flag to display a
         * single entry
         */
        if(!is_array($e))
        {
            $fulldisp = 1;
            $e = array(
                'title' => 'No Entries Yet',
                'entry' => '<a href="/admin.php">Post an entry!</a>'
            );
        }
    }

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

    return $e;
}

?>
```

Writing the Business Function

At this point in your application, the business layer is pretty simple. All you need to do at this point is escape your output to avoid potential issues. You can accomplish this by writing a function called `sanitizeData()`, which you declare right below `retrieveEntries()` in `functions.inc.php`.

This function accepts one parameter, `$data`, and performs basic sanitization using the `strip_tags()` function. Sanitizing the function removes all HTML from a string unless a tag is specifically *whitelisted*, or placed in a collection of allowed tags, in `strip_tags()` second parameter.

The data you pass to `sanitizeData()` is potentially a mixture of both array and string data, so you need to check whether `$data` is an array before you process any data—doing this can help you avoid any parsing errors.

If `$data` isn't an array, you use `strip_tags()` to eliminate all HTML tags except the `<a>` tag; this enables your entries to contain links.

If `$data` *is* an array, you use the `array_map()` function to call `sanitizeData()` *recursively* on each element in the array.

Recursive Functions

In some cases, it becomes necessary to call a function from within itself. This technique is known as a recursive function call, and it has a number of useful applications. In this instance, you use recursion to ensure that every element in an array is sanitized, no matter how deep your array goes. In other words, the first element contains an array where its first element is another array, and so on. Recursion allows your function to be called repeatedly until you reach the bottom of the array.

Sanitizing the Data

The next step is to declare `sanitizeData()` and write the code to perform the recursive technique just described. Add this code to `functions.inc.php`, just below `retrieveEntries()`:

```
function sanitizeData($data)
{
    // If $data is not an array, run strip_tags()
    if(!is_array($data))
    {
        // Remove all tags except <a> tags
        return strip_tags($data, "<a>");
    }

    // If $data is an array, process each element
    else
    {
        // Call sanitizeData recursively for each array element
        return array_map('sanitizeData', $data);
    }
}
```

Writing the Presentation Code

Your last step in this phase of creating the blog is to use the information retrieved and formatted by your database and business layers to generate HTML markup and display the entries.

You will write this code in `index.php` inline with the HTML markup. The reason for this approach: This code is strictly for inserting your processed data into HTML markup.

Begin by including both `db.inc.php` and `functions.inc.php` in `index.php`. At the very top of `index.php`, add the following code:

```
<?php

    /*
     * Include the necessary files
     */
    include_once 'inc/functions.inc.php';
    include_once 'inc/db.inc.php';

?>
```

Next, you need to open a connection to the database. You also need to check whether an entry ID was passed in the URL.

■**Note** Passing entry IDs in the URL (*i.e.*, `http://localhost/simple_blog/`**`?id=1`** is a popular and straightforward way of using one page to display different entries. You accomplish this in PHP using the `$_GET` superglobal.

Now add the bold lines to `index.php`:

```
<?php

    /*
     * Include the necessary files
     */
    include_once 'inc/functions.inc.php';
    include_once 'inc/db.inc.php';

    // Open a database connection
    $db = new PDO(DB_INFO, DB_USER, DB_PASS);

    // Determine if an entry ID was passed in the URL
    $id = (isset($_GET['id'])) ? (int) $_GET['id'] : NULL;

?>
```

So far, you've determined whether an ID is set using the ternary operator, which allows you to compress an `if` statement into one line. Translated into plain English, the previous code snippet would read like this: "if `$_GET['id']` is set to some value, save its value as an integer in `$id`, or else set the value of `$id` to `NULL`."

Next, you need to load the entries from the database. Do this by calling your `retrieveEntries()` function and passing it your database connection (`$db`) and the ID you collected (`$id`) as parameters. Now add the lines in bold to `index.php`:

```
<?php

    /*
     * Include the necessary files
     */
    include_once 'inc/functions.inc.php';
    include_once 'inc/db.inc.php';

    // Open a database connection
    $db = new PDO(DB_INFO, DB_USER, DB_PASS);
    // Determine if an entry ID was passed in the URL
    $id = (isset($_GET['id'])) ? (int) $_GET['id'] : NULL;

    // Load the entries
    $e = retrieveEntries($db, $id);

?>
```

The appropriate entries for the page are stored in the $e array and are ready to be displayed. You know that the last element of the array contains a flag telling you whether a full entry is stored, so your next step is to pop the last element off the array and store it in a variable (`$fulldisp`) that you'll use in just a moment.

Also, you need to sanitize the entry data, which we do by calling `sanitizeData()` and passing `$e` as the parameter. Next, add the lines in bold to `index.php`:

```
<?php

    /*
     * Include the necessary files
     */
    include_once 'inc/functions.inc.php';
    include_once 'inc/db.inc.php';

    // Open a database connection
    $db = new PDO(DB_INFO, DB_USER, DB_PASS);

    // Determine if an entry ID was passed in the URL
    $id = (isset($_GET['id'])) ? (int) $_GET['id'] : NULL;

    // Load the entries
    $e = retrieveEntries($db, $id);

    // Get the fulldisp flag and remove it from the array
    $fulldisp = array_pop($e);

    // Sanitize the entry data
    $e = sanitizeData($e);

?>
```

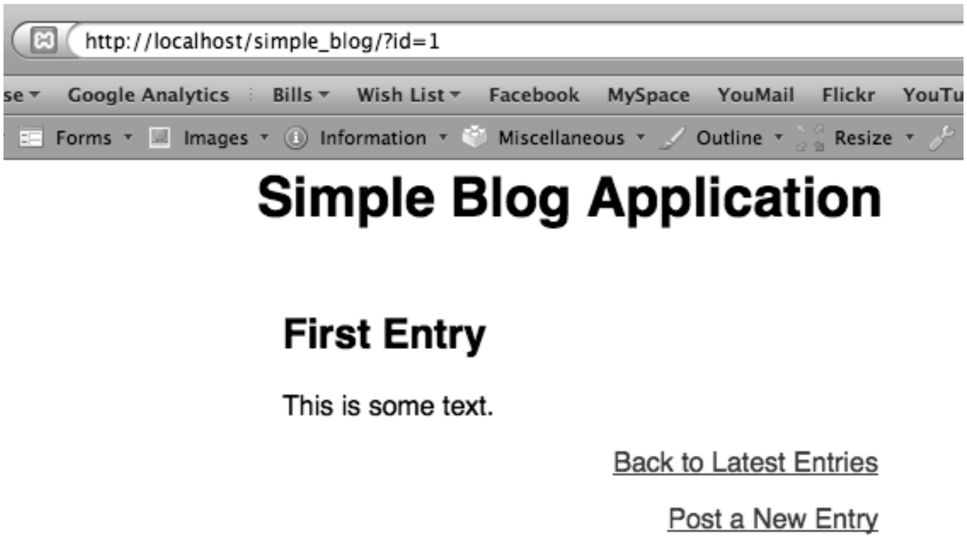
At this point, you have a flag to let you know whether you're displaying a full entry or a list of entry titles (`$fulldisp`), as well as an array of information to insert into HTML markup (`$e`).

To create the output, you need to determine whether the flag is set to `1`, which would signify a full entry. If so, you insert the entry title into an `<h2>` tag and place the entry in a `<p>` tag.

In index.php, in the middle of the page below <div id="entries">, add the following lines of bold code:

```
    <div id="entries">

<?php

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

?>

        <h2> <?php echo $e['title'] ?> </h2>
        <p> <?php echo $e['entry'] ?> </p>
        <p class="backlink">
            <a href="./">Back to Latest Entries</a>
        </p>

<?php

} // End the if statement

?>

        <p class="backlink">
            <a href="/admin.php">Post a New Entry</a>
        </p>

    </div>
```

Navigating to the http://localhost/simple_blog/?id=1 address enables you to see the first entry (see Figure 5-7).



Simple Blog Application

First Entry

This is some text.

Back to Latest Entries

Post a New Entry

Figure 5-7. *The first entry loaded using a variable passed in the URL*

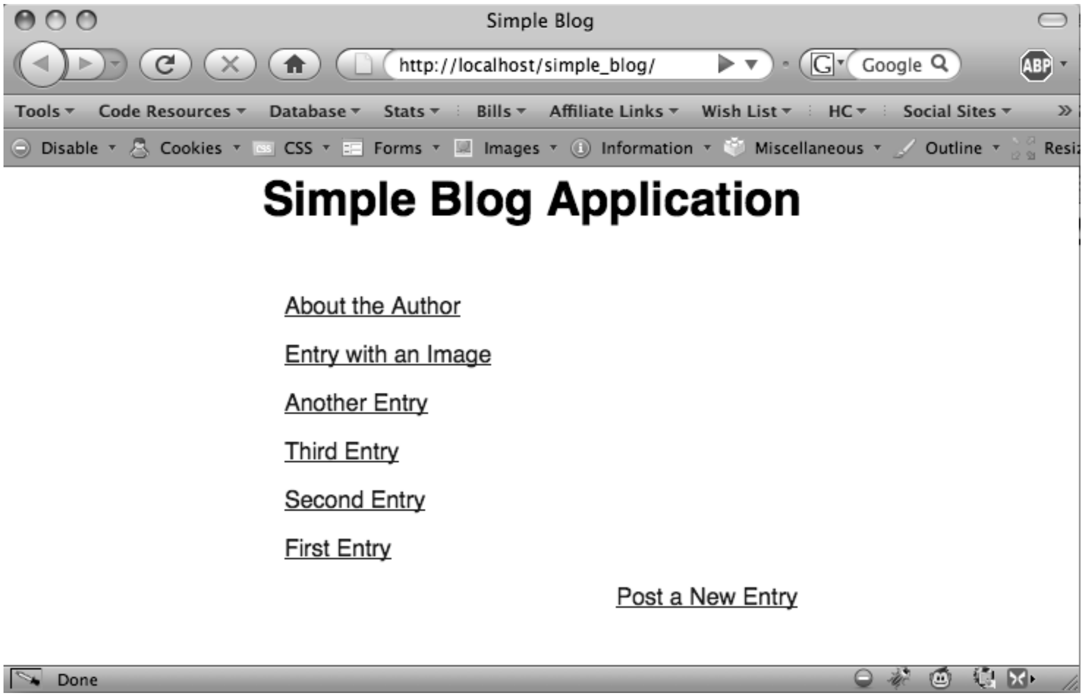
Next, you need to determine how you should display your list of entry titles. Ideally, you want to show the title as a link that takes the user to view the full entry.

This list of links is displayed if the `$fulldisp` flag is set to `0`, so add an `else` to the conditional statement that checks whether `$fulldisp` is set to `1`. Inside the else statement, you need to create a loop to process each paired ID and title together.

Just after the `if` statement, add the bold lines of code to `index.php`:

```
<?php

} // End the if statement

// If the full display flag is 0, format linked entry titles
else
{
    // Loop through each entry
    foreach($e as $entry) {

?>


        <p>
            <a href="?id=<?php echo $entry['id'] ?>">
                <?php echo $entry['title'] ?>

            </a>
        </p>
<?php

    } // End the foreach loop
} // End the else

?>

        <p class="backlink">
            <a href="/admin.php">Post a New Entry</a>
        </p>

    </div>
```

Now, navigate to `http://localhost/simple_blog/`, and you should see the title of each entry listed as a link(see Figure 5-8). Clicking any of the links takes you to the associated entry.


Simple Blog
http://localhost/simple_blog/
Google
Tools Code Resources Database Stats Bills Affiliate Links Wish List HC Social Sites
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resi
Done

Simple Blog Application

About the Author

Entry with an Image

Another Entry

Third Entry

Second Entry

First Entry

Post a New Entry

Figure 5-8. *The title of each entry is listed as a link*

Fix the Redirect

Now that index.php exists, you want to be taken to your new entries after they are submitted. To do this, you need to change the address of the header() calls to take the user to index.php. Change the code in bold in update.inc.php to make this happen:

```
<?php

if($_SERVER['REQUEST_METHOD']=='POST'
    && $_POST['submit']=='Save Entry')
{
    // Include database credentials and connect to the database
    include_once 'db.inc.php';
    $db = new PDO(DB_INFO, DB_USER, DB_PASS);

    // Save the entry into the database
    $sql = "INSERT INTO entries (title, entry) VALUES (?, ?)";
    $stmt = $db->prepare($sql);
    $stmt->execute(array($_POST['title'], $_POST['entry']));
    $stmt->closeCursor();

    // Get the ID of the entry we just saved
    $id_obj = $db->query("SELECT LAST_INSERT_ID()");
    $id = $id_obj->fetch();
    $id_obj->closeCursor();

    // Send the user to the new entry
    header('Location: ../?id='.$id[0]);
    exit;
}

// If both conditions aren't met, sends the user back to the main page
else
{
    header('Location: ../');
    exit;
}

?>
```

Summary

You have now created a blog in the basic sense! Basic techniques you learned in this chapter included:

- How to use a web form to create and save entries in the database
- How to retrieve and display entries based on variables passed in the URL

As you continue on, you'll add several cool features to the blog, including a formatted date, authoring information, and images. In the next chapter, you'll learn how to make your blog support multiple pages, which in turn will enable you to build an "About the Author" page.