Apress™
Books for Professionals by Professionals

Chapter Twelve: “Templates”

A Programmer’s Introduction to PHP 4.0

by William Jason Gilmore
ISBN # 1-893115-85-2

info@apress.com

CHAPTER 12

Templates

You can think of templates as an extension to programming. Not only do they automate an otherwise rigorous process, but they also facilitate the division of design and coding in a team environment. This division of labor becomes increasingly important as a project and team grow in size and complexity due to logistics surrounding the initial design as well as subsequent maintenance of the Web application.

To put this into perspective, consider the developmental logistics of a team divided between Web designers and programmers. The ideal situation would be that the Web design team could embark on building the most eye-appealing, user-friendly site they can, while at the same time the programming team codes the most efficient, powerful Web application possible. Fortunately, templates make this process much easier. Creating a site templating system that facilitates this separation of labor is the subject of this chapter.

What You've Learned So Far

To this point, I've introduced two systems for creating PHP templates:

- HTML embedded in PHP code
- Including files in a page structure

Although the first strategy is the easier to understand and implement, it is also the more restrictive. A major problem lies in the fact that the actual PHP coding is intermingled with the components making up the page design. This presents a problem not only in terms of the need to provide support for the potential simultaneous access and modification to a single page, but also in regards to the increased possibility of introducing errors due to constant scrutiny and editing.

The second strategy can be a substantial improvement over the first in many situations. However, while a patchwork system of header, body, and footer files (see Chapter 9 for further information) works fine for piecing together relatively small sites of specific format, limitations become apparent as the project grows in size and complexity. Attempts to resolve these problems led to the development of another templating strategy, which, although more complex than the other two strategies, is substantially more flexible and superior in many situations. This

system calls for the separation of the two main components of a Web application: design and coding. Separating the design and coding of a Web application makes it possible for simultaneous development (Web design and programming) without the need of extensive coordination throughout the lifetime of the initial development cycle. Furthermore, it allows for the later exclusive modification of either component without interfering with the other. In the next section, I'll elaborate on how one of these advanced templating systems is constructed. Keep in mind that this problem is not specific to PHP. In fact, this general strategy certainly precedes PHP's lifetime, and is currently used in several languages, including PHP, Perl, and Java Server Pages. What is described in this chapter is merely an adaptation of this strategy as it would apply to PHP.

Developing an Advanced Template System

As you may have already surmised, the major hurdle to overcome when developing this type of templating system is efficiently dividing the labor between design and functionality. After all, the intended goal of this system is to allow both coders and designers to independently maintain their end of the application *without* interfering with the work of the other group.

Thankfully, this isn't as difficult as it sounds, provided that some planning takes place *before* the development process begins. To illustrate how this is accomplished, Listing 12-1 presents a basic template that you will actually be able to implement with the code provided in this chapter.

Listing 12-1: Sample template file

```
<html>
<head>
<title>:::::{page_title}:::::</title>
</head>
<body bgcolor="{bg_color}">
Welcome to your default home page, {user_name}!<br>
You have 5 MB and 3 email addresses at your disposal<br>
Have fun!
</body>
</html>
```

Notice that there are three strings enclosed in curly brackets ({}), namely `page_title`, `bg_color`, and `user_name`. The brackets are of special meaning to the template parsing engine, signifying that the enclosed string specifies the name of a variable that should be replaced with its respective value. Other than ensuring that these key strings are placed where necessary in the document, the designer is

free to structure the page as desired. Of course, the coder and the designer must come to terms with what exactly is to be placed in each page!

So how will the templating system operate? To start, you may be dealing with several templates simultaneously, all having the same general characteristics. This could be well suited for object-oriented programming (OOP). Therefore, all functions used to build and manipulate the templates will actually be methods in a class. The class definition starts as follows:

```
class template {
     VAR $files = array();
     VAR $variables = array();
     VAR $opening_escape = '{';
     VAR $closing_escape = '}';
```

The attribute `$files` is an array that will store the file identifiers and the corresponding contents of each file. The attribute `$variables` is a two-dimensional array used to store a file identifier (key) and all of the corresponding variables to be parsed in the template system. Finally, the attributes `$opening_escape` and `$closing_escape` refer to the variable delimiters that specify which parts of the template are to be replaced by the system. As you have seen in Listing 12-1, I use curly brackets ({}) as delimiters. However, you can use these final two attributes to change the delimiters to anything you please; just make sure the resulting combination will be unique for this purpose.

Each method in the class serves a well-defined purpose, each corresponding to a step in the templating process. At its most basic level, this process can actually be broken down into these four parts:

- **File registration**: Registration of all files that will be parsed by the templating scripts.
- **Variable registration**: Registration of all variables that will be replaced with corresponding values in the registered files.
- **File parsing:** Replacement of all delimited variables in the registered files.
- **File printing:** Display of the parsed registered files in the browser.

NOTE *OOP as it applies to PHP was discussed in Chapter 6. If you are unfamiliar with OOP, I would suggest quickly reviewing Chapter 6 before continuing.*

File Registration

Registering a file simply means assigning its contents to an array key, the key uniquely referring to the said file. This method opens and reads in the contents of the file passed in as an input parameter. Listing 12-2 illustrates this method.

Listing 12-2: The method for registering a file

```
function register_file($file_id, $file_name) {

     // Open $file_name for reading, or exit and print an error message.
     $fh = fopen($file_name, "r") or die("Couldn't open $file_name!");

     // Read in the entire contents of $file_name.
     $file_contents = fread($fh, filesize($file_name));

     // Assign these contents to a position in the array.
     // This position is denoted by the key $file_id
     $this->files[$file_id] = $file_contents;

     // We're finished with the file, so close it.
     fclose($fh);
} // end register_file
```

The input parameter `$file_id` is just a pseudonym for the file, which will make all subsequent method calls easier to understand. This parameter will serve as the array key for the `$files` array. Here is an example of how a file would be registered:

```
// Include the template class
include("template.class");

// Instantiate a new object
$template = new template;

// Register the file "homepage.html", assigning it the pseudonym "home"
$template->register_file("home", "homepage.html");
```

Variable Registration

After you register the files, all of the variables that are to be parsed must also be registered with the system. This method operates along the same premise as `register_file()`, retrieving each named variable and inserting it into the `$variables` array. Listing 12-3 illustrates this method.

Listing 12-3: The method for registering a variable

```
function register_variables($file_id, $variable_name) {

     // attempt to create array from passed in variable names
     $input_variables = explode(",", $variable_name);

     // assign variable name to next position in $file_id array
     while (list(,$value) = each($input_variables)) :

          // assign the value to a new position in the $this->variables array
          $this->variables[$file_id][] = $value;

     endwhile;

} // end register_variables
```

The input parameter `$file_id` refers to the *previously* assigned alias of a filename. In the previous example, "home" was the assigned alias of "homepage.html." Therefore, if you are registering variable names that are to be parsed in the homepage.html file, you must refer to it by its alias! The input parameter `$variable_name` refers to one or several variables that are to be registered under that alias name. An example follows:

```
// Include the template class
include("template.class");

// Instantiate a new object
$template = new template;

// Register the file "homepage.html", assigning it the pseudonym "home"
$template->register_file("home", "homepage.html");

// Register a few variables
$template->register_variables("home", "page_title,bg_color,user_name");
```

File Parsing

After the files and variables have been registered in the templating system, you are free to parse the registered files, replacing all variable references with their respective values. Listing 12-4 illustrates this method.

Listing 12-4: The method for parsing a file

```
function file_parser($file_id) {

     // How many variables are registered for this particular file?
     $varcount = count($this->variables[$file_id]);

     // How many files are registered?
     $keys = array_keys($this->files);

     // If the $file_id exists in the $this->files array and it
     // has some registered variables…
     if ( (in_array($file_id, $keys)) && ($varcount > 0) ) :

          // reset $x
          $x = 0;

          // while there are still variables to parse…
          while ($x < sizeof($this->variables[$file_id])) :

               // retrieve the next variable
               $string = $this->variables[$file_id][$x];

               // Retrieve this variable value! Notice that I'm using a
               // variable variable to retrieve this value. This value
               // will be substituted into the file contents, taking the place
               // of the corresponding variable name.
               GLOBAL $$string;

               // What exactly is to be replaced in the file contents?
               $needle = $this->opening_escape.$string.$this->closing_escape;

               // Perform the string replacement.
               $this->files[$file_id] = str_replace(
                           $needle,   // needle
                           $$string,   // string
                           $this->files[$file_id]); // haystack
               // increment $x
               $x++;
          endwhile;
     endif;
} // end file_parser
```

Basically, the `$file_id` that is passed in is verified to exist in the `$this->files` array. If it does, it is also verified that this `$file_id` has some variables registered that need to be parsed. If it does, the values of each of those variables are retrieved and substituted into the contents of `$file_id`. An example follows:

```
<?

// Include the template class
include("template.class");

$page_title = "Welcome to your homepage!";
$bg_color = "white";
$user_name = "Chef Jacques";

// Instantiate a new object
$template = new template;

// Register the file "homepage.html", assigning it the pseudonym "home"
$template->register_file("home", "homepage.html");

// Register a few variables
$template->register_variables("home", "page_title,bg_color,user_name");
$template->file_parser("home");

?>
```

Since the variables `page_title`, `bg_color`, and `user_name` have been registered, the corresponding values of each (assigned at the beginning of the script) will be substituted into the homepage.html contents, stored in the objects files array attribute. To this point, everything has been done except actually displaying the resulting template to the browser. This is the next step.

File Printing

After parsing the file, you will probably want to print it out, thereby displaying the parsed template to the user. I create this in a separate method shown in Listing 12-5, but you could also integrate this directly into the `file_parser()` method, depending on your usage.

Listing 12-5: The method for printing a file

```
function print_file($file_id) {

     // print out the contents pointed to by $file_id
     print $this->files[$file_id];

}
```

Quite simply, when `print_file()` is called, the contents represented by the key `$file_id` are output to the browser.

Listing 12-6 displays the finished templating system.

Listing 12-6: Complete example of templating system

```
<?

// Include the template class
include("template.class");

// Assign a few variables
$page_title = "Welcome to your homepage!";
$bg_color = "white";
$user_name = "Chef Jacques";

// Instantiate a new object
$template = new template;

// Register the file "homepage.html", assigning it the pseudonym "home"
$template->register_file("home", "homepage.html");

// Register a few variables
$template->register_variables("home", "page_title,bg_color,user_name");
$template->file_parser("home");
// output the file to the browser
$template->print_file("home");

?>
```

If the template first displayed in Listing 12-1 was saved under the name homepage.html and stored in the same directory as the script shown in Listing 12-6, then the following would be displayed in the browser:

```
<html>
<head>
<title>:::::Welcome to your homepage!:::::</title>
</head>
```

```
<body bgcolor=white>
Welcome to your default home page, Chef Jacques!<br>
You have 5 MB and 3 email addresses at your disposal<br>
Have fun!
</body>
</html>
```

As you can see, all of the previously delimited variable names have been replaced with their respected values. Although simple, this template class ensures 100 percent separation of design and coding. The entire contents of the template class are shown in Listing 12-7.

Listing 12-7: Complete template.class file.

```
<?
class template {

VAR $files = array();
VAR $variables = array();
VAR $opening_escape = '{';
VAR $closing_escape = '}';

// Function: register_file()
// Purpose: Store contents of file specified by $file_id

function register_file($file_id, $file_name) {

     // Open $file_name for reading, or exit and print an error message.
     $fh = fopen($file_name, "r") or die("Couldn't open $file_name!");

     // Read in the entire contents of $file_name.
     $file_contents = fread($fh, filesize($file_name));

     // Assign these contents to a position in the array.
     // This position is denoted by the key $file_id
     $this->files[$file_id] = $file_contents;

     // We're finished with the file, so close it.
     fclose($fh);
} // end register_file;

// Function: register_variables()
// Purpose: Store variables passed in via $variable_name under the corresponding
// array key, specified by $file_id
```

```
function register_variables($file_id, $variable_name) {

     // attempt to create array from passed in variable names
     $input_variables = explode(",", $variable_name);

     // assign variable name to next position in $file_id array
     while (list(,$value) = each($input_variables)) :

          // assign the value to a new position in the $this->variables array
          $this->variables[$file_id][] = $value;

     endwhile;

} // end register_variables

// Function: file_parser()
// Purpose: Parse all registered variables  in file contents
//                 specified by input parameter $file_id

function file_parser($file_id) {

     // How many variables are registered for this particular file?
     $varcount = count($this->variables[$file_id]);

     // How many files are registered?
     $keys = array_keys($this->files);

     // If the $file_id exists in the $this->files array and it
     // has some registered variables…
     if ( (in_array($file_id, $keys)) && ($varcount > 0) ) :

          // reset $x
          $x = 0;

          // while there are still variables to parse…
          while ($x < sizeof($this->variables[$file_id])) :

               // retrieve the next variable
               $string = $this->variables[$file_id][$x];

               // retrieve this variable value! Notice that I'm using a
               // variable variable to retrieve this value. This value
```

```
                // will be substituted into the file contents, taking the place
                // of the corresponding variable
                // name.
                GLOBAL $$string;

                // What exactly is to be replaced in the file contents?
                $needle = $this->opening_escape.$string.$this->closing_escape;

                // Perform the string replacement.
                $this->files[$file_id] = str_replace(
                            $needle,   // needle
                            $$string,   // string
                            $this->files[$file_id]); // haystack
                // increment $x
                $x++;
          endwhile;
     endif;
} // end file_parser


// Function: print_file()
// Purpose: Print out the file contents specified by input parameter $file_Id

function print_file($file_id) {
     // print out the contents pointed to by $file_id
     print $this->files[$file_id];
}

} // END template.class
```

Expanding the Template Class

Of course, this template class is rather limited, although it does the trick nicely for projects that need to be created in a hurry. The nice thing about using an object-oriented implementation strategy is that you can easily add functionality without worrying about potentially "breaking" existing code. For example, suppose you wanted to create a method that retrieved values from a database for later template substitution. Although slightly more complicated than the `file_parser()` method, which just substitutes globally-accessible variable values, an SQL-based file parser can be written with just a few lines and encapsulated in its own method. In fact, I create one of these parsers in the address book project at the conclusion of this chapter.

Several modifications could be made to this template class, the first likely being the consolidation of register_file() and register_variables(). This would automatically add the variables in each registered file. Of course, you will also want to insert error-checking functionality to ensure that invalid file and variable names are not registered.

You are also likely to begin thinking about how this system could be enhanced. Consider the following enhancement questions. How would you create a method that worked with entire arrays? Included Files? I think that you'll find it easier than it first seems. As a reference, check out the implementation I created for an SQL-parser in the address book project at the end of this chapter. You can easily transform this general methodology into whatever implementation you desire.

This basic templating strategy has been implemented in several languages and is certainly not a new concept. Therefore, you can find a wealth of information on the Web pertaining to template implementations. Two particularly interesting resources are this set of related articles, written with JavaScript in mind:

- `http://developer.netscape.com/viewsource/long_ssjs/long_ssjs.html`
- `http://developer.netscape.com/viewsource/schroder_template/schroder_template.html`

The following article touches upon templates as it applies to Java Server Pages:

- `http://www-4.ibm.com/software/webservers/appserv/doc/guide/asgdwp.html`

There are also quite a few PHP implementations that follow this templating strategy. Several of the more interesting ones include:

- PHPLib Base Library (`http://phplib.netuse.de`)
- Richard Heyes's Template Class (`http://www.heyes-computing.net`)
- Fast Template (`http://www.thewebmasters.net/php/`)

The PHP resource site PHPBuilder (`http://www.phpbuilder.com`) also contains a few interesting tutorials regarding template manipulation. Also check out PHP Classes Repository (`http://phpclasses.UpperDesign.com`). Several similar templating implementations are there.

Drawbacks to This Templating System

While this form of templating fulfills its purpose of completely separating the code from the design, it is not without its disadvantages. I'll highlight these disadvantages here.

Resulting Unfounded Belief in "Silver Bullet" Solution

While templates can aid in clearly defining the boundaries of a project in terms of coding and design, they are not a substitute for communication. In fact, they won't even operate correctly without concise communication between both parties about exactly what information will be templated in the application. As is the case with any successful software project, a thorough survey of the application specifications should always be drawn up before even one line of PHP is coded. This will greatly reduce the possibility for later miscommunication, resulting in unexpected template parsing results.

Performance Degradation

The dependence on file parsing and manipulation will cause the templating system to suffer a loss in performance in terms of speed. Exactly what the degree of this loss is depends on a number of factors, including page size, SQL query size (if any), and machine hardware. In many cases, this loss will be negligible; however there may be instances where it will be noticeable if it becomes necessary to simultaneously manipulate several template files in high-traffic situations.

Designer Is Still PHP-Impaired

One of the main reasons for creating this system at all lies in the fact that it could be problematic if the designer comes into contact with the code when editing the look and feel of the page. In an ideal environment, the designer would also be a programmer or at least know general programming concepts, such as a variable, loop, and conditional. A designer who is not privy to this information stands to gain nothing from using templates except education in a relatively useless syntax (the syntax used to delimit variable keywords). Therefore, regardless of what your final verdict is on using this form of page templating, I strongly recommend taking time to begin educating the designer on the finer points of the PHP language (or better, buy the designer a copy of this book!). This results in a win-win situation for both parties, as the designer will learn an extra skill, and in doing so, become an even more valuable member of the team. The programmer wins, as this person will be an extra brain to pick for new programming ideas, perhaps even a particularly valuable one, since chances are that the designer will look at things from a different perspective than the typical programmer would.

Project: Create an Address Book

Although templating systems are well suited for a variety of Web applications, they are particularly useful in datacentric applications in which formatting is important. One such application is an address book. Think about what a conventional (paper-based) address book looks like: each page looks exactly the same, save for perhaps a letter denoting which set of last names the particular page is reserved for. The same kind of idea could apply to a Web-based address book. In fact, formatting is even more important in this case, since it might be necessary to export the data to another application in a particularly rigorous format. This kind of application works great with the templating system, since the designer is left to create a single page format that will be used for all 26 letters of the alphabet.

To begin, you must decide what kind of data you want to store in the address book and how this data is to be stored. Of course, the most plausible choice for a storage media would be a database, since this also facilitates useful features such as searching and ordering data. I'll use a MySQL database to store the address information. The table looks like this:

```
mysql>CREATE table addressbook (
          last_name char(35) NOT NULL,
          first_name char(20) NOT NULL,
          tel char(20) NOT NULL,
          email char(55) NOT NULL );
```

Of course, you can add street address, city, and state columns. I'll use this abbreviated table for sake of illustration.

Next, I'll play the role of designer and create the templates. For this project, two templates are required. The first template, shown in Listing 12-8, could be considered the "parent" template.

Listing 12-8: Parent address book template, entitled "book.html"

```
<html>
<head>
<title>:::::{page_title}:::::</title>
</head>
<body bgcolor="white">

<table cellpadding=2 cellspacing=2 width=600>
<h1>Address Book: {letter}</h1>
<tr><td>
<a href="index.php?letter=a">A</a> | <a href="index.php?letter=b">B</a> |
<a href="index.php?letter=c">C</a> | <a href="index.php?letter=d">D</a> |
<a href="index.php?letter=e">E</a> | <a href="index.php?letter=f">F</a> |
```

```
<a href="index.php?letter=g">G</a> | <a href="index.php?letter=h">H</a> |
<a href="index.php?letter=i">I</a> | <a href="index.php?letter=j">J</a> |
<a href="index.php?letter=k">K</a> | <a href="index.php?letter=l">L</a> |
<a href="index.php?letter=m">M</a> | <a href="index.php?letter=n">N</a> |
<a href="index.php?letter=o">O</a> | <a href="index.php?letter=p">P</a> |
<a href="index.php?letter=q">Q</a> | <a href="index.php?letter=r">R</a> |
<a href="index.php?letter=s">S</a> | <a href="index.php?letter=t">T</a> |
<a href="index.php?letter=u">U</a> | <a href="index.php?letter=v">V</a> |
<a href="index.php?letter=w">W</a> | <a href="index.php?letter=x">X</a> |
<a href="index.php?letter=y">Y</a> | <a href="index.php?letter=z">Z</a>
</td></tr>

{rows.addresses}
</table>
</body>
</html>
```

As you can see, the bulk of this file is given to the links displaying each letter of the alphabet. Clicking a particular letter, the user will be presented with all persons stored in the address book having a last name beginning with that letter.

There are also three delimited variable names: `page_title`, `letter`, and `rows.addresses`. The purpose of the first two variables should be obvious: the title of the page and the letter of the address book currently used to retrieve address information, respectively. The third variable refers to the child template and is used to specify which table configuration file should be inserted into the parent. I say "table configuration file" because, in a complex page, you might be simultaneously using several templates, each employing HTML tables for formatting data. Therefore, "rows" specifies that a table template will be inserted, and "addresses" tells us that it is the table used to format addresses.

The second template, shown in Listing 12-9, is the "child" template, because it will be embedded in the parent. Why this is necessary will soon become clear.

Listing 12-9: Child address book template, entitled "rows.addresses"

```
<tr><td bgcolor="#c0c0c0">
<b>{last_name},{first_name}</b>
</td></tr>
<tr><td>
<b>{telephone}</b>
</td></tr>
<tr><td>
<b><a href = "mailto:{email}">{email}</a></b>
</td></tr>
```

There are four delimited variable names in Listing 12-9: `last_name`, `first_name`, `telephone`, and `email`. The meanings of each should be obvious. It is important to notice that this file only contains table row (`<tr>…</tr>`) and table cell (`<td>…</td>`) tags. This is because this file will be repeatedly inserted into the template, one time for each address retrieved from the database. Since the `rows.addresses` variable name is enclosed in table tags in Listing 12-8, the HTML formatting will parse correctly. To illustrate how this works, take a look at Figure 12-1, which is essentially a screenshot of the completed address book in address. Then examine Listing 12-10, which contains the source code for that screen shot. You'll see that the rows.addresses file is used repeatedly in the source code.

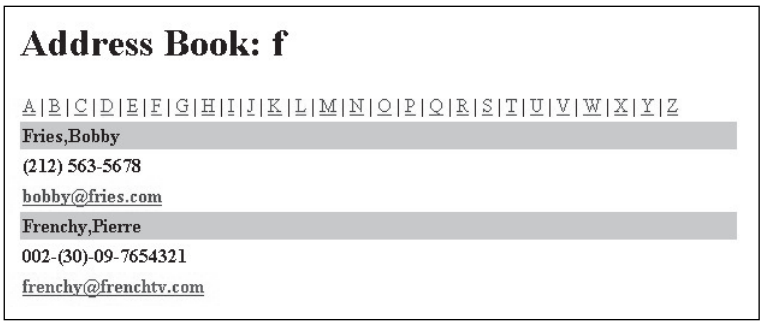

Figure 12-1. Screenshot of the address book in action

Listing 12-10: Source code for Figure 12-1

```
<html>
<head>
<title>:::::Address Book:::::</title>
</head>
<body bgcolor="white">

<table cellpadding=2 cellspacing=2 width=600>
<h1>Address Book: f</h1>
<tr><td><a href="index.php?letter=a">A</a> | <a href="index.php?letter=b">B</a> |
<a href="index.php?letter=c">C</a> |
<a href="index.php?letter=d">D</a> | <a href="index.php?letter=e">E</a> | <a
href="index.php?letter=f">F</a> | <a href="index.php?letter=g">G</a> | <a
href="index.php?letter=h">H</a> | <a href="index.php?letter=i">I</a> | <a
href="index.php?letter=j">J</a> | <a href="index.php?letter=k">K</a> | <a
```

```
href="index.php?letter=l">L</a> | <a href="index.php?letter=m">M</a> | <a
href="index.php?letter=n">N</a> | <a href="index.php?letter=o">O</a> | <a
href="index.php?letter=p">P</a> | <a href="index.php?letter=q">Q</a> | <a
href="index.php?letter=r">R</a> | <a href="index.php?letter=s">S</a> | <a
href="index.php?letter=t">T</a> | <a href="index.php?letter=u">U</a> | <a
href="index.php?letter=v">V</a> | <a href="index.php?letter=w">W</a> | <a
href="index.php?letter=x">X</a> | <a href="index.php?letter=y">Y</a> | <a
href="index.php?letter=z">Z</a></td></tr>

<tr><td bgcolor="#c0c0c0">
<b>Fries,Bobby</b>
</td></tr>
<tr><td>
<b>(212) 563-5678</b>
</td></tr>
<tr><td>
<b><a href = "mailto:bobby@fries.com">bobby@fries.com</a></b>
</td></tr>
<tr><td bgcolor="#c0c0c0">
<b>Frenchy,Pierre</b>
</td></tr>
<tr><td>
<b>002-(30)-09-7654321</b>
</td></tr>
<tr><td>
<b><a href = "mailto:frenchy@frenchtv.com">frenchy@frenchtv.com</a></b>
</td></tr>
</table>
</body>
</html>
```

As you can see, there are apparently two persons having a last name that begins with *F* stored in the address book, Bobby Fries and Pierre Frenchy. Therefore, two table rows have been inserted in the table.

The design process for the address book project is complete. Now, I'll don the hat of a coder. You'll be surprised to know that there are no changes to the `template.class` file in Listing 12-7, save for one new method, `address_sql()`. This method is displayed in Listing 12-11.

Listing 12-11: SQL parsing method, address_sql()

```
class template {

     VAR $files = array();
     VAR $variables = array();
     VAR $sql = array();
     VAR $opening_escape = '{';
     VAR $closing_escape = '}';

     VAR $host = "localhost";
     VAR $user = "root";
     VAR $pswd = "";
     VAR $db = "book";
     VAR $address_table = "addressbook";
. . .
     function address_sql($file_id, $variable_name, $letter) {

     // Connect to MySQL server and select database
     mysql_connect($this->host, $this->user, $this->pswd)
            or die("Couldn't connect to MySQL server!");
     mysql_select_db($this->db) or die("Couldn't select MySQL database!");

     // Query database
     $query = "SELECT last_name, first_name, tel, email
                    FROM $this->address_table WHERE last_name LIKE '$letter%'";
     $result = mysql_query($query);

     // Open "rows.addresses" file and read contents into variable.
     $fh = fopen("$variable_name", "r");

     $file_contents = fread($fh, filesize("rows.addresses") );

     // Perform replacements of delimited variable names with table data
     while ($row = mysql_fetch_array($result)) :

          $new_row = $file_contents;

          $new_row = str_replace(
                    $this->opening_escape."last_name".$this->closing_escape,
                    $row["last_name"],
                    $new_row);
```

```
            $new_row = str_replace(
                      $this->opening_escape."first_name".$this->closing_escape,
                      $row["first_name"],
                      $new_row);

            $new_row = str_replace(
                      $this->opening_escape."telephone".$this->closing_escape,
                      $row["tel"],
                      $new_row);

            $new_row = str_replace(
                      $this->opening_escape."email".$this->closing_escape,
                      $row["email"],
                      $new_row);

            // Append new table row onto complete substitution string
            $complete_table .= $new_row;

        endwhile;

        // Assign table substitution string to SQL array key
        $sql_array_key = $variable_name;

        $this->sql[$sql_array_key] = $complete_table;

        // add the key to the variables array for later lookup
        $this->variables[$file_id][] = $variable_name;

        // Close the filehandle
        fclose($fh);

        } // end address_sql
        . . .
} // end template.class
```

The comments in Listing 12-11 should suffice for understanding the mechanics of what is taking place. However, there are still a few important points to make. First, notice that the rows.addresses file is opened only *once*. An alternative way to code this method would be to repeatedly open and close the rows.addresses file, replacing information each time and appending it to the `$complete_table` variable. However, this would be highly inefficient coding practice. Therefore, take some time to review how the loop is used to continuously append new table information to the `$complete_table` variable.

A second point to make about Listing 12-11 is that five new class attributes are used: `$host`, `$user`, `$pswd`, `$db`, and `$address_table`. Each of these pertains to information that the MySQL server requires, and the meaning of each should be obvious. If it isn't obvious, take a moment to read through Chapter 11, "Databases."

All that's left to do now is code the file that triggers the template parsing. This file is shown in Listing 12-12. By clicking one of the letter links (index.php?letter=someletter) in book.html (Listing 12-8), this file will be called, in turn regenerating the book.html file with appropriate information.

Listing 12-12: Template parser index.php

```
<?
include("template.class");

$page_title = "Address Book";

// The default page will retrieve persons having last name beginning with 'a'
if (! isset($letter) ) :
     $letter = "a";
endif;

$tpl = new template;
$tpl->register_file("book", "book.html");
$tpl->register_variables("book", "page_title,letter");
$tpl->address_sql("book", "rows.addresses","$letter");
$tpl->file_parser("book");
$tpl->print_file("book");
?>
```

There you have it: a practical example of how templates can be used to efficiently divide labor between coder and designer. Take some time to think about how you can use templates to further streamline your development process. I'll bet that you find a number of different implementations for templates.

What's Next?

This chapter introduced a particularly useful concept of both PHP and Web programming in general: advanced template usage. It began with a synopsis of the two templating systems covered thus far, simple variable substitution via PHP embedding, and the use of INCLUDE files to separate page components. I then introduced the third and most advanced template strategy, which completely separates the code from the design of the page. The remainder of the chapter was

spent examining a class built to implement this type of template, concluding with a practical implementation of the template system, using a Web-based address book as an example. This example also built on the simple template class, implementing an SQL parser.

In particular, the following topics were discussed in this chapter:

- Why templates?
- Simple template #1: embedding PHP in HTML
- Simple template #2: using INCLUDE files to separate components
- Advanced templating through the complete division of design and code
- The template class
- File registration
- Variable registration
- File parsing
- File printing
- Disadvantages to using templates
- Address book project that expands on the default class, implementing an SQL parser

Next chapter, I continue the discussion of dynamic Web application development, introducing how cookies and session tracking can add a new degree of user interactivity to your Web site!