15 VALIDATING DATABASE INPUT AND USER AUTHENTICATION

Dreamweaver provides you with the basic functionality of inserting and updating records in a database, but it's up to you to make sure that the data entered by a user meets the criteria you envisaged when designing the database structure. When designing database forms, you must remember the GIGO principle—garbage in, garbage out. Unless you control carefully what you allow to go into a database, a lot of your results will be useless garbage. Many developers rely on JavaScript validation to filter user input before it's submitted to the database, but JavaScript is easily turned off in the browser leaving your site vulnerable. JavaScript validation, such as that provided by Spry validation widgets (see Chapter 9), should be regarded as a convenience offered to the user. The only way to make sure data is safe to insert into a database is to validate it with PHP.

In this chapter, we're going to get down and dirty with PHP code. If you don't come from a programming background, that thought might fill you with horror, but you should never deploy dynamic code on a website without understanding what it's for. In any case, PHP is not difficult. A major reason for its popularity is that it's relatively easy to learn. If the code looks strange to you, it's because it's unfamiliar. The more you work with it, the more familiar—and easier—it becomes. If you feel inspired to study PHP more, for a hands-on approach take a look at my *PHP Solutions: Dynamic Web Design Made Easy* (friends of ED, ISBN: 978-1-59059-731-6). Or if you prefer a reference book, grab hold of *Beginning PHP and MySQL: From Novice to Professional, Third Edition* by W. Jason Gilmore (Apress, ISBN: 978-1-59059-862-7).

We'll start by examining the code that Dreamweaver created when you built the insert and update forms in the previous chapter. There's no need to study every line of code. The key thing is to recognize the code Dreamweaver generates, where it puts it, and what it's for. This makes it easy to adapt the code to do much more than the basic functionality provided by the server behaviors. I'll also show you how to create simple server behaviors of your own to speed up the process of creating interactive web pages.

By the end of this chapter you will have enhanced the insert and update forms and made them much more user-friendly by preventing invalid input, displaying error messages, and preserving user input when it fails validation. Once the forms have been updated, you'll be able to use the user registration system to control access to sensitive or protected areas of your site.

In this chapter, you'll learn about the following:

- Recognizing the code generated by Dreamweaver server behaviors
- Preventing the creation of duplicate usernames
- Building your own custom server behaviors
- Preserving information related to an individual visitor with PHP sessions
- Restricting access to your pages

This chapter builds on the user registration system created in the previous chapter, so it assumes you have built `register_user.php`, `list_users.php`, `update_user.php`, and `delete_user.php`. However, to make sure everyone begins from the same starting point, I have included versions of each file in the download files for this chapter. Let's begin by examining the code that Dreamweaver generated for you.

Analyzing the code generated by Dreamweaver

My first attempt at developing a database-driven website was with Dreamweaver UltraDev 4 using ASP. It was a disaster. There were two major problems. First, I didn't realize the importance of removing server behaviors cleanly through the Server Behaviors panel if I changed my mind about how I wanted the page to work. Second, the code didn't look anything like the ASP in any of the books I consulted. I was so frustrated; I went away and learned to hand-code everything in PHP.

Even if you have studied some PHP, you might find the code generated by Dreamweaver overwhelming at first sight. However, it's actually quite straightforward, and it's organized in blocks that are relatively easy to recognize. It needs to be, because Dreamweaver needs to recognize them in order to let you edit or remove them through the Server Behaviors panel. Once you learn to recognize the blocks, you can begin to modify them yourself to add much greater functionality and flexibility to your websites. Wherever possible, I try to leave Dreamweaver's code blocks intact, because that preserves the ability to edit them through the Server Behaviors panel. However, that's not always possible, but if you keep a cool head, you'll quickly find that Dreamweaver speeds up development by creating the basic code for you to improve upon. I have no difficulty hand-coding a database query, but Dreamweaver accomplishes in seconds what it would take me many minutes to type.

As I said before, I don't intend to go through the code line by line, nor will I cover all the code generated by Dreamweaver when building the user registration system in the previous chapter. This is intended as a quick overview so you can recognize the code associated with the main server behaviors. It should also help you troubleshoot some common problems.

Inspecting the server behavior code

I suggest you open the pages in Code view as you read through this section to help familiarize yourself with the code. Copies of the finished pages from the previous chapter are in `examples/ch15`. The insert and update pages are called `register_user_start.php` and `update_user_start.php`, because they will be used as the starting point for building the server-side validation later in this chapter. The other two pages, `list_users.php` and `delete_user.php`, don't need further improvement, so their names are unchanged.

Connecting to the database

If you open each of the pages, you'll see that they all begin with the following line of code:

```
<?php require_once('../../Connections/connAdmin.php'); ?>
```

If your site definition uses links relative to the site root, `require_once()` is replaced by `virtual()` like this:

```
<?php virtual('/Connections/connAdmin.php'); ?>
```

This includes the login details for the MySQL user account. If this code or the include file is missing, the rest of the script cannot connect to the database, so nothing will work. As explained in the previous chapter, `virtual()` is supported only by the Apache web server. So, a page that works perfectly on Apache will suddenly stop working if you move the site to any other web server.

Preventing SQL injection

Immediately following the line of code that includes the database connection details is the lengthy block of code shown in Figure 15-1. This defines a custom function called `GetSQLValueString()`, which prepares values submitted through a form or query string for insertion into a database query. Its main task is to prevent a malicious attack known as **SQL injection**, which attempts to pass spoof values to a database in the hope of extracting confidential information or corrupting the data.

```
<?php
if (!function_exists("GetSQLValueString")) {
function GetSQLValueString($theValue, $theType, $theDefinedValue = "", $theNotDefinedValue = "")
{
  if (PHP_VERSION < 6) {
    $theValue = get_magic_quotes_gpc() ? stripslashes($theValue) : $theValue;
  }

  $theValue = function_exists("mysql_real_escape_string") ? mysql_real_escape_string($theValue) :
mysql_escape_string($theValue);

  switch ($theType) {
    case "text":
      $theValue = ($theValue != "") ? "'" . $theValue . "'" : "NULL";
      break;
    case "long":
    case "int":
      $theValue = ($theValue != "") ? intval($theValue) : "NULL";
      break;
    case "double":
      $theValue = ($theValue != "") ? doubleval($theValue) : "NULL";
      break;
    case "date":
      $theValue = ($theValue != "") ? "'" . $theValue . "'" : "NULL";
      break;
    case "defined":
      $theValue = ($theValue != "") ? $theDefinedValue : $theNotDefinedValue;
      break;
  }
  return $theValue;
}
}
```

Figure 15-1. The GetSQLValueString() function helps protect your database from malicious attack.

The function also ensures that strings are correctly enclosed in quotes when incorporated in a SQL query.

Inserting a record into a database

Figure 15-2 shows the rest of the code Dreamweaver inserted above the `DOCTYPE` declaration in `register_user_start.php`.

```
34  $editFormAction = $_SERVER['PHP_SELF'];
35  if (isset($_SERVER['QUERY_STRING'])) {
36    $editFormAction .= "?" . htmlentities($_SERVER['QUERY_STRING']);
37  }
38
39  if ((isset($_POST["MM_insert"])) && ($_POST["MM_insert"] == "form1")) {
40    $insertSQL = sprintf("INSERT INTO users (first_name, family_name, username, pwd, admin_priv) VALUES (%s, %s,
     %s, %s, %s)",
41                         GetSQLValueString($_POST['first_name'], "text"),
42                         GetSQLValueString($_POST['family_name'], "text"),
43                         GetSQLValueString($_POST['username'], "text"),
44                         GetSQLValueString($_POST['pwd'], "text"),
45                         GetSQLValueString($_POST['admin_priv'], "text"));
46
47    mysql_select_db($database_connAdmin, $connAdmin);
48    $Result1 = mysql_query($insertSQL, $connAdmin) or die(mysql_error());
49
50    $insertGoTo = "list_users.php";
51    if (isset($_SERVER['QUERY_STRING'])) {
52      $insertGoTo .= (strpos($insertGoTo, '?')) ? "&" : "?";
53      $insertGoTo .= $_SERVER['QUERY_STRING'];
54    }
55    header(sprintf("Location: %s", $insertGoTo));
56  }
```

Figure 15-2. The Insert Record server behavior inserts a record and redirects the user to the next page.

The first four lines (34–37) set a variable called `$editFormAction` to the name of the current page and preserve any query string in the URL. The variable is used later in the page to set the value of the `action` attribute in the insert form. You can normally leave this block of code alone unless you want to add anything to the query string.

Immediately following these four lines of code is the core of the Insert Record server behavior.

The server behavior is wrapped in a conditional statement that makes sure the code is run only when the insert form has been submitted. It's easy to tell that this is an Insert Record server behavior because all the variables begin with `$insert` (Dreamweaver's variables and functions use names that make it easy to guess their purpose). As you can see on lines 41–45 of Figure 15-2, the value of each form field is passed to the `GetSQLValueString()` function to prepare it for insertion in the SQL query.

Lines 40–45 build the SQL query; line 47 selects the correct database; and line 48 executes the query, inserting the new record into the database table.

The remaining lines redirect the user to the next page (in this case, `list_users.php`), preserving any values in the query string. The actual redirect is performed by the `header()` function on line 55.

If you don't specify a page to redirect to after the record is inserted, the code shown on lines 50–55 is omitted.

Understanding why a redirect doesn't work

A question that turns up regularly in online forums is why an insert or update form doesn't redirect the user to the next page after inserting or updating the record. The key to understanding the problem lies in knowing how the `header()` function works. I have mentioned this several times already, but it confuses so many people, it's worth repeating here. The `header()` function cannot do its job if any output is sent to the browser before you call the function.

This means you can't use `echo`, `print`, or any other function that outputs content anywhere before a call to `header()`. Nor can any HTML appear before `header()`. Other things that prevent `header()` from working are using the byte-order mark or whitespace outside PHP tags. A common cause of failure is extra whitespace at the end of an include file (see "Avoiding the 'headers already sent' error" in Chapter 12).

Updating a database record

Now take a look at `update_user_start.php`. Figure 15-3 shows the code immediately following the `GetSQLValueString()` function. Compare it with the code in Figure 15-2. It's almost identical. The differences are that all the variables begin with `$update` and the SQL query built on lines 40–46 uses the `UPDATE` command rather than `INSERT`.

```
34  $editFormAction = $_SERVER['PHP_SELF'];
35  if (isset($_SERVER['QUERY_STRING'])) {
36    $editFormAction .= "?" . htmlentities($_SERVER['QUERY_STRING']);
37  }
38
39  if ((isset($_POST["MM_update"])) && ($_POST["MM_update"] == "form1")) {
40    $updateSQL = sprintf("UPDATE users SET first_name=%s, family_name=%s, username=%s, pwd=%s, admin_priv=%s
    WHERE user_id=%s",
41                         GetSQLValueString($_POST['first_name'], "text"),
42                         GetSQLValueString($_POST['family_name'], "text"),
43                         GetSQLValueString($_POST['username'], "text"),
44                         GetSQLValueString($_POST['pwd'], "text"),
45                         GetSQLValueString($_POST['admin_priv'], "text"),
46                         GetSQLValueString($_POST['user_id'], "int"));
47
48    mysql_select_db($database_connAdmin, $connAdmin);
49    $Result1 = mysql_query($updateSQL, $connAdmin) or die(mysql_error());
50
51    $updateGoTo = "list_users.php";
52    if (isset($_SERVER['QUERY_STRING'])) {
53      $updateGoTo .= (strpos($updateGoTo, '?')) ? "&" : "?";
54      $updateGoTo .= $_SERVER['QUERY_STRING'];
55    }
56    header(sprintf("Location: %s", $updateGoTo));
57  }
```

Figure 15-3. The Update Record server behavior code is almost identical to the Insert Record server behavior.

Everything else works exactly the same way as an Insert Record server behavior.

Deleting a record

Figure 15-4 shows the Delete Record server behavior in `delete_user.php`. It's easy to recognize because all the variables begin with `$delete`. It simply deletes a record and redirects to another page.

```
38  if ((isset($_POST['user_id'])) && ($_POST['user_id'] != "")) {
39    $deleteSQL = sprintf("DELETE FROM users WHERE user_id=%s",
40                         GetSQLValueString($_POST['user_id'], "int"));
41
42    mysql_select_db($database_connAdmin, $connAdmin);
43    $Result1 = mysql_query($deleteSQL, $connAdmin) or die(mysql_error());
44
45    $deleteGoTo = "list_users.php";
46    if (isset($_SERVER['QUERY_STRING'])) {
47      $deleteGoTo .= (strpos($deleteGoTo, '?')) ? "&" : "?";
48      $deleteGoTo .= $_SERVER['QUERY_STRING'];
49    }
50    header(sprintf("Location: %s", $deleteGoTo));
51  }
```

Figure 15-4.
The Delete Record server behavior deletes a record without confirmation.

The key point to note about this server behavior, as I explained in the previous chapter, is that the conditional statement surrounding the server behavior checks only that the variable being used to identify the record exists. If it does, it goes ahead and deletes the record.

In the previous chapter, I told you to set the Primary key value in the Delete Record dialog box to Form Variable. This makes the server behavior use the `$_POST` array and gives you the opportunity to confirm that the correct record is being deleted. If, on the other hand, you use the default setting, URL Parameter, the server behavior uses the `$_GET` array. This results in the record being deleted immediately without confirmation.

Distinguishing between Form Variable and URL Parameter

A lot of server behavior dialog boxes ask you to specify the origin of a variable. The two most frequently used values are Form Variable and URL Parameter, so it's important to understand the difference.

- Form Variable: This uses the `$_POST` array and takes the value from a form submitted using the `post` method.
- URL Parameter: This uses the `$_GET` array and takes the value from a query string at the end of a URL or from a form submitted using the `get` method.

If a server behavior doesn't pick up a variable, check that you haven't selected the wrong one.

Many beginners get mixed up between `get` and `post`, but it makes a crucial difference to how your page works. If you're still unclear about the difference, skip back to Chapter 9 and refresh your memory.

Retrieving database records with a recordset

Figure 15-5 shows the remaining code inserted above the `DOCTYPE` declaration in `update_user_start.php`. This is the code for the `getUser` recordset.

```
59  $colname_getUser = "-1";
60  if (isset($_GET['user_id'])) {
61    $colname_getUser = $_GET['user_id'];
62  }
63  mysql_select_db($database_connAdmin, $connAdmin);
64  $query_getUser = sprintf("SELECT user_id, username, pwd, first_name, family_name, admin_priv FROM users WHERE
    user_id = %s", GetSQLValueString($colname_getUser, "int"));
65  $getUser = mysql_query($query_getUser, $connAdmin) or die(mysql_error());
66  $row_getUser = mysql_fetch_assoc($getUser);
67  $totalRows_getUser = mysql_num_rows($getUser);
```

Figure 15-5.
Dreamweaver uses the recordset name to create the variables.

If you cast your mind back to the previous chapter, I told you that you needed to create the recordset *before* using the Record Update Form Wizard, yet the recordset code has been inserted *after* the Update Record server behavior. This is the way that Dreamweaver works—the code for a recordset is always inserted immediately above the DOCTYPE declaration. Normally, this is fine, but a recordset often produces information that can be useful for validation and needs to be moved. The good news is that Dreamweaver doesn't mind you moving the code, just as long as you keep it all together.

The first thing to notice about a recordset is that the names of all the variables are based on the name you give the recordset. So, giving a recordset a name that describes its purpose makes it a lot easier to recognize the right code. This is what the variables mean (*recordsetName* changes depending on the name you give the recordset):

- $colname_*recordsetName*: This is the variable being used as a filter for the recordset. In the getUser recordset, you set the primary key, user_id, as the filter, so this holds the value of user_id passed in through the query string of the URL. As you'll see in later chapters, you can use more than one variable to filter results. When you use more than one variable, colname is replaced by the variable name you choose yourself.
- $query_*recordsetName*: This contains the SQL query used to create the recordset.
- $*recordsetName*: This contains the results of the database query.
- $row_*recordsetName*: This is an array that contains the results from the current record. Dreamweaver automatically gets the first record so that it's ready for display inside the page.
- $totalRows_*recordsetName*: This contains the number of records retrieved from the database. This is extremely useful in determining whether the query produced any results.

The basic recordset code is on lines 63–67 of Figure 15-5. All recordsets contain these five lines of code. The code shown on lines 59–62 defines the variable for the filter. If more than one variable is used as a filter, each one is defined in the same way.

All the server behavior code you have looked at so far is placed above the DOCTYPE *declaration. This is perfectly OK because it doesn't send any output to the browser, except when redirecting the user to another page. When adapting server behaviors or writing PHP code of your own, don't put anything above the* DOCTYPE *that will send output to the browser, because it will render your CSS in quirks mode, possibly breaking your design. The only exception is when debugging code. Sometimes, it's useful to display the value of variables to see why your code isn't working as expected, but you should remove the debugging code when you have finished testing.*

Creating a repeat region

The code used to create a repeat region is very simple. It consists of just two lines wrapped around the code that you want to repeat. Figure 15-6 shows the repeat region that you applied to the second table row in list_users.php. The two lines that repeat the table row are highlighted on lines 62 and 70. They create a simple do . . . while loop (see Chapter 10). Dreamweaver uses a do . . . while loop because the first record is already

stored in $row_*recordsetName*, as explained in the preceding section. The code inside the parentheses at the end of the loop on line 70 gets the next row of results from the recordset.

```
62  <?php do { ?>
63    <tr>
64      <td><?php echo $row_listUsers['first_name']; ?> <?php echo $row_listUsers['family_name']; ?></td>
65      <td><?php echo $row_listUsers['username']; ?></td>
66      <td><?php echo $row_listUsers['admin_priv']; ?></td>
67      <td><a href="update_user.php?user_id=<?php echo $row_listUsers['user_id']; ?>">EDIT</a></td>
68      <td><a href="delete_user.php?user_id=<?php echo $row_listUsers['user_id']; ?>">DELETE</a></td>
69    </tr>
70  <?php } while ($row_listUsers = mysql_fetch_assoc($listUsers)); ?>
```

Figure 15-6. The code for a repeat region is simple, but its location is vital.

Usually when a repeat region goes haywire, it's the result of selecting the wrong elements in Design view before applying the server behavior. A quick look at the code should confirm what the problem is.

Adding server-side validation

The user registration form created by the Record Insertion Form Wizard has several problems. Figure 15-7 shows what happens if you submit the form without filling in any fields (top screenshot) or if a username is used more than once (bottom screenshot).

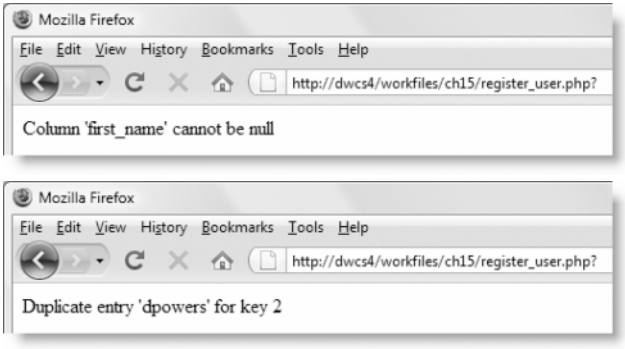


Figure 15-7. The default error messages are not user-friendly.

Setting all columns to `NOT NULL` in the table definition prevents anyone from submitting the form without filling in each field, but there's no guarantee that the right type of information will be input. As things currently stand, a single space would satisfy the form's definition of a required field. Applying a unique index to the `username` column certainly prevents duplicate entries, but the error message isn't very informative. More important, the form has disappeared, and the only way to get back to it is to click the back button in the browser.

Of course, you could prevent this sort of problem by applying the Spry validation widgets that you studied in Chapter 9. This would probably be sufficient for most bona fide users, but the Web is a dangerous place filled with people with less honorable intentions. Anyone intent on a malicious attack normally disables JavaScript, and even if the information in

your database remains intact, it could easily be filled with unwanted garbage. So, it's important to validate input on the server before inserting it into your database.

The registration form created by the Dreamweaver wizard needs the following improvements:

- All required fields must contain specified minimum content.
- When a field fails to validate, a suitable error message must be displayed.
- Existing input must be preserved when an error occurs.

Let's begin by making sure that each field is filled in with a minimum amount of content.

Verifying that required fields have been filled in

All fields are required, so you need to check that they contain at least something. If a problem is detected, the validation code needs to prevent the `INSERT` command from being executed. The series of tests that you'll add to the code in `register_user.php` perform only simple checks on the user input. You can make them much more rigorous. The purpose of the following exercises is to demonstrate the principles behind server-side validation, rather than incorporate exhaustive tests. The level of testing you choose depends entirely on what the form is for. An insurance proposal form is likely to warrant far more rigorous validation than one for a community forum.

Adding server-side validation to the Insert Record server behavior is easy to implement, but it involves editing the server behavior, so it's no longer accessible through the Server Behaviors panel. The idea of losing access to server behaviors through the panel instills terror into the mind of most newcomers to dynamic design, but it's important to remember that Dreamweaver server behaviors cannot do everything. To get the best out of them, you frequently need to amend the code. If you cling tenaciously to the dialog box interface, you'll be severely limited in what you can achieve.

Checking required fields

This section uses the PHP functions, `trim()`, `empty()`, and `strlen()`, to trim whitespace from user input and check whether it's empty or how many characters it contains. If any problems are encountered, error messages are created for display later in the registration form. You can continue using `register_user.php` from the previous chapter. Alternatively, copy `register_user_start.php` from `examples/ch15` to `workfiles/ch15`, and rename it `register_user.php`.

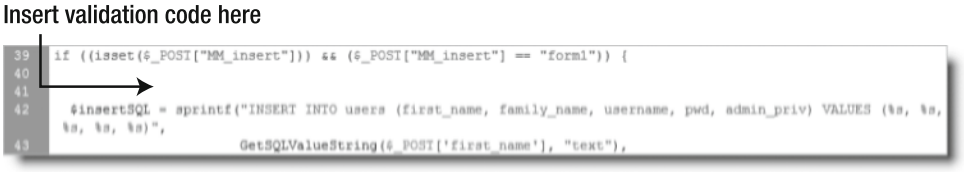
1. With `register_user.php` open in the Document window, switch to Code view. The validation code should run only if the form has been submitted. Locate the following code (it should be on or around line 39):

   ```
   if ((isset($_POST["MM_insert"])) && ($_POST["MM_insert"] == "form1")) {
   ```

   This conditional statement checks the value of a hidden field to see whether the insert form has been submitted. So, if you place the validation code inside the braces of this conditional statement, your new code runs only at the same time as the Insert Record server behavior. Doing so means that the server behavior ceases

to be editable through a dialog box, but this is a sacrifice you must make in the interests of data integrity.

Place your cursor at the end of this line, and press Enter/Return a couple of times to make room for the validation code, as shown in the following screenshot:



Insert the following code:

```
// Initialize array for error messages
$error = array();
// Remove whitespace and check first and family names
$_POST['first_name'] = trim($_POST['first_name']);
$_POST['family_name'] = trim($_POST['family_name']);
if (empty($_POST['first_name']) || empty($_POST['family_name'])) {
  $error['name'] = 'Please enter both first name and family name';
}
```

This initializes `$error` as an empty array. PHP treats an array with zero elements as `false` (see "The truth according to PHP" in Chapter 10), so this can be used later to test whether any errors have been found and, if so, to prevent the Insert Record server behavior from attempting to execute the `INSERT` query.

The remaining lines use `trim()` to remove leading and trailing whitespace from the `first_name` and `family_name` fields and then pass them to `empty()`. If either field has no value, an appropriate message is added to the `$error` array.

> *You might wonder why I haven't reassigned the values of the* `$_POST` *array variables to shorter ones, as with the mail processing script in Chapter 11. It's because they're required by the Insert Record server behavior. Changing them here would involve further changes to the code generated by Dreamweaver, increasing not only your workload but also the likelihood of errors creeping in.*

2. The next check makes sure that the username contains at least six characters. It uses the PHP function `strlen()`, which determines the number of characters in any string passed to it. Add the following code immediately after the code in the preceding step:

```
// Check the username for length
$_POST['username'] = trim($_POST['username']);
if (strlen($_POST['username']) < 6) {
  $error['length'] = 'Please select a username that contains at least ➥
    6 characters';
}
```

3. A similar check is done next on the password. The following code goes immediately after the code in the previous step:

```
// set a flag that assumes the password is OK
$pwdOK = true;
// trim leading and trailing white space
$_POST['pwd'] = trim($_POST['pwd']);
// if less than 6 characters, create alert and set flag to false
if (strlen($_POST['pwd']) < 6) {
  $error['pwd_length'] = 'Your password must be at least 6 characters';
  $pwdOK = false;
}
```

This code starts by setting a variable that assumes the password is OK. After trimming any whitespace, `strlen()` is used to check that the trimmed password contains at least six characters. If it doesn't, an error message is added to the `$error` array, and `$pwdOK` is set to `false`. You'll see the purpose of the `$pwdOK` variable in the next section.

If you would like to check your code so far, compare it against `register_user_01.php` in `examples/ch15`.

Verifying and encrypting the password

Since the password won't appear onscreen, you should get the user to type it in twice to confirm the spelling. Also, to keep the password secure, it should be encrypted before it's stored in the database. Encryption is important because it keeps the passwords secret, even if someone manages to compromise the security of the database and expose the stored passwords.

Improving password validation

In this section, you'll add an extra field for the user to retype the password to ensure that both versions match. You'll also encrypt the password before it's passed to the SQL query. Continue working with the same file as in the preceding section.

1. Adding a new field for the user to confirm the password means adding a new row to the table that contains the registration form. You can do this in several ways. Start by switching back to Design view and clicking inside the table cell that contains the Administrator label. If you have a good memory for keyboard shortcuts, the quickest and easiest way to add a new table row is to press Ctrl+M/Cmd+M. This always inserts a new row *above* the current one.

   Alternative ways of adding a new row are to use the menu system. Modify ➤ Table ➤ Insert Row does the same as the keyboard shortcut: the new row goes above the current one. Modify ➤ Table ➤ Insert Rows or Columns opens a dialog box that lets you specify the number of rows or columns to be inserted and on which side of the

current selection to put them. Finally, the Layout tab of the Insert bar offers a visual way of doing it.

Use whichever method you prefer to create a new row between Password and Administrator. Then type Confirm password as the label in the left cell, and insert a text field in the right cell. Name the text field `conf_pwd`, and set Type to Password in the Property inspector (form creation was covered in Chapter 9).

> *The table layout for the insert form created by the wizard doesn't use* `<label>` *tags, so choose the* No label tag *option in the* Input Tag Accessibility Attributes *dialog box. Using the wizard is best avoided except when you're developing a prototype as a proof of concept, which will be rebuilt using your own forms and designs later. I'll show you how to apply Insert Record and Update Record server behaviors to custom-built forms in the next chapter.*

2. You can now compare the content of the `pwd` and `conf_pwd` fields. Switch to Code view, and add the following code immediately after the code you inserted in step 3 of the previous section:

```
    $error['pwd_length'] = 'Your password must be at least 6 characters';
    $pwdOK = false;
}
// if no match, create alert and set flag to false
if ($_POST['pwd'] != trim($_POST['conf_pwd'])) {
    $error['pwd'] = "Your passwords don't match";
    $pwdOK = false;
}
```

This trims whitespace off both ends of `$_POST['conf_pwd']` and compares the result with `$_POST['pwd']`. There's no need to pass the original password to `trim()` because that was already done in the previous section and the value reassigned to `$_POST['pwd']`. Also, there's no need to store the result of `trim($_POST['conf_pwd'])`, because you're using it only to make sure the two entries match. If they do, this conditional statement will be ignored. However, if there's a mismatch, an error message is created, and `$pwdOK` is set to `false`.

3. Finally, if `$pwdOK` is still `true`, you can encrypt the password by passing it to the `sha1()` function like this (the code goes immediately after the code in the previous step):

```
// if password OK, encrypt it
if ($pwdOK) {
    $_POST['pwd'] = sha1($_POST['pwd']);
}
```

The `sha1()` function converts any string passed to it into a 40-character hexadecimal number—in effect, encrypting the string ready for insertion into the database.

You can check your code, if necessary, against `register_user_02.php` in `examples/ch15`.

Dealing with duplicate usernames

Dreamweaver has a server behavior called Check New User that queries your database to find out whether a username is already in use. Unfortunately, it's badly designed and guaranteed to enrage visitors to your site. If it finds a duplicate username, it takes the visitor to another page and wipes out all the information that had been entered into the form. Applying a unique index to the `username` column, as you did in the previous chapter, is a much more elegant way of handling the situation, but you need a way to prevent the form from disappearing when a duplicate entry is detected. This is done by checking the error code returned by MySQL.

Creating an error message for duplicate usernames

The following instructions show you how to amend the Insert Record server behavior to generate a user-friendly error message when the `INSERT` query fails as the result of a duplicate username being submitted. Continue working with the same file.

1. Approximately ten lines below the last section of code you have just inserted, locate the line that looks like this (it should be on or around line 80):

   ```
   $Result1 = mysql_query($insertSQL, $connAdmin) or die(mysql_error());
   ```

   What this line does is execute the `INSERT` query; but if there's a problem, the section highlighted in bold displays an error message and brings all further processing to a halt.

   The draconian-sounding function `die()` tells a PHP script to terminate immediately if it encounters an error. It takes a single argument: the error message you want to display onscreen. In this case, the message is generated by another function, `mysql_error()`, which gives you access to the most recent error message from MySQL.

   Instead of bringing the script to a halt, it's far more user-friendly to redisplay the form ready for the user to submit an alternative username.

2. Remove the section highlighted in bold so the line of code looks like this:

   ```
   $Result1 = mysql_query($insertSQL, $connAdmin);
   ```

   Make sure you don't lose the semicolon at the end of the line.

3. In addition to `mysql_error()`, PHP has a function called `mysql_errno()`, which returns an error code from MySQL. Although error messages are easier for human beings to understand, it's easier for PHP to work with numbers. Add the conditional statement highlighted in bold, as shown here:

   ```
   $Result1 = mysql_query($insertSQL, $connAdmin);
   if (!$Result1 && mysql_errno() == 1062) {
     $error['username'] = $_POST['username'] . ' is already in use. ➥
       Please choose a different username.';
   } elseif (mysql_error()) {
   ```

```
  $error['dbError'] = 'Sorry, there was a problem with the database. ➥
    Please try later.';
}
$insertGoTo = "list_users.php";
```

If the Insert Record server behavior succeeds, `$Result1` is `true`. So, placing the negative operator (`!`) in front of `$Result1` tests whether it is *not* `true`—in other words, whether it fails. A duplicate value entered into a unique index column produces the MySQL error code `1062`. So if the Insert Record server behavior fails and the error code is `1062`, you know it's because of a duplicate value.

The code inside the first conditional statement uses `$_POST['username']`, the value submitted from the registration form, to create an error message and stores the message in `$error['username']`.

You should never display the contents of MySQL error messages in a live web page, because it can reveal information that might be helpful to an attacker. So, the second conditional statement checks for any other MySQL error and creates a generic error message.

If you encounter problems when testing the page, substitute the line of code in the second conditional statement with the following:

```
$error['dbError'] = mysql_error();
```

This gives you access to the MySQL error message. Once you have identified the problem, replace `mysql_error()` with the neutral message.

4. If the database returns an error, you need to prevent the script from redirecting the user to the next page, so wrap the code that redirects the page in a final `else` statement like this:

```
  $error['dbError'] = 'Sorry, there was a problem with the database.
Please try later.';
  } else {
    $insertGoTo = "list_users.php";
    if (isset($_SERVER['QUERY_STRING'])) {
     $insertGoTo .= (strpos($insertGoTo, '?')) ? "&" : "?";
      $insertGoTo .= $_SERVER['QUERY_STRING'];
    }
    header(sprintf("Location: %s", $insertGoTo));
  }
}
```

By placing the redirection code in the final `else` block of the conditional statement, the redirect goes ahead only if the database doesn't return an error (you can see the full chain of conditions in Figure 15-8).

You can check your code so far against `register_user_03.php` in `examples/ch15`.

MySQL error messages can appear rather cryptic. Chapter 17 contains advice on understanding them and troubleshooting problems with SQL queries.

Displaying the error messages

Now that the checks are complete, you need to build the logic that determines whether the record is inserted in the database. If there are no errors, the new record is inserted into the database, and the user is redirected to the next page. However, if errors are detected, the INSERT command is ignored, and the form needs to be redisplayed with the appropriate error messages.

Building the error detection logic

This section completes the validation process by wrapping the code that inserts the record in a conditional statement to prevent it from being executed if any errors are discovered. You will also add code to the insert form to display any error messages. Continue working with the same file.

1. If no errors have been found, the `$error` array will contain zero elements, which, as you know, PHP treats as `false`. Wrap the remaining section of the Insert Record server behavior code with this conditional statement (the exact location is shown in Figure 15-8):

```
// if no errors, insert the details into the database
if (!$error) {
  // Insert Record server behavior code
}
```

Insert conditional statement here

```
67   // if password OK, encrypt it
68   if ($pwdOK) {
69     $_POST['pwd'] = sha1($_POST['pwd']);
70   }
71
72   // if no errors, insert the details into the database
73   if (!$error) {
74     $insertSQL = sprintf("INSERT INTO users (first_name, family_name, username, pwd, admin_priv) VALUES (%s,
%s, %s, %s, %s)",
75                          GetSQLValueString($_POST['first_name'], "text"),
76                          GetSQLValueString($_POST['family_name'], "text"),
77                          GetSQLValueString($_POST['username'], "text"),
78                          GetSQLValueString($_POST['pwd'], "text"),
79                          GetSQLValueString($_POST['admin_priv'], "text"));
80
81     mysql_select_db($database_connAdmin, $connAdmin);
82     $Result1 = mysql_query($insertSQL, $connAdmin);
83     if (!$Result1 && mysql_errno() == 1062) {
84       $error['username'] = $_POST['username'] . ' is already in use. Please choose a different username.';
85     } elseif (mysql_error()) {
86       $error['dbError'] = 'Sorry, there was a problem with the database. Please try later.';
87     } else {
88       $insertGoTo = "list_users.php";
89       if (isset($_SERVER['QUERY_STRING'])) {
90         $insertGoTo .= (strpos($insertGoTo, '?')) ? "&" : "?";
91         $insertGoTo .= $_SERVER['QUERY_STRING'];
92       }
93       header(sprintf("Location: %s", $insertGoTo));
94     }
95   }
96 }
97 ?>
98 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
```

Figure 15-8. The conditional statement prevents the record from being inserted if any errors are found.

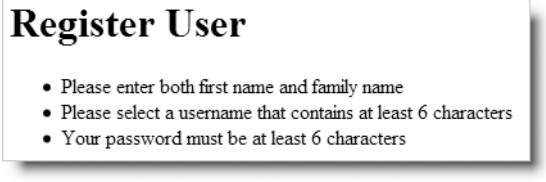
The negation operator (an exclamation mark) gives you the reverse meaning of a value. So if `$error` is an empty array, this test equates to `true`, and the Insert Record server behavior is executed. If errors are found, the test equates to `false`, and the server behavior is ignored.

2. Scroll down to the page heading (around line 106) just below the `<body>` tag, and insert the following code block between the heading and the opening `<form>` tag:

```
<h1>Register user </h1>
<?php
if (isset($error)) {
  echo '<ul>';
  foreach ($error as $alert) {
    echo "<li class='warning'>$alert</li>\n";
  }
  echo '</ul>';
}
?>
<form action="<?php echo $editFormAction; ?>" method="post" ➥
name="form1" id="newUser">
```

This begins by checking whether the `$error` array exists, because it's created only when the form is submitted. If it doesn't exist, the whole block is ignored. If it does exist, a `foreach` loop iterates through the array and assigns each element to the temporary variable `$alert`, which is used to display the error messages as a bulleted list. (See Chapter 10 if you need to refresh your memory about `foreach` loops.)

3. Save `register_user.php`, and load it into a browser. Click the Insert record button without filling in any fields. The page should reload and display the following warnings:



4. Now try filling in all fields, but with a username that is already registered. This time, you should see something similar to this:

If you have any problems, check your code against `register_user_04.php` in `examples/ch15`. The page contains no style rules, but if you add a `warning` class, you could make the error messages stand out in bold, red text.

This has improved the insert form considerably, but imagine the frustration of being forced to fill in all the details again because of a mistake in just one field. What you really need is a server behavior to provide the same solution you used in the contact form in Chapter 11. There isn't one, but you can make it yourself.

Building custom server behaviors

One reason for the great success of Dreamweaver is that, in addition to its massive range of features, it's also extensible. You can build your own server behaviors to take the tedium out of repetitive tasks.

To redisplay the contents of a text field after a form has been submitted, all you need to do is insert a PHP conditional statement between the quotes of the `<input>` element's `value` attribute like this:

```
value="<?php if (isset($_POST['field'])) {echo htmlentities( ➥
$_POST['field'], ENT_COMPAT, UTF-8);} ?>"
```

This checks whether the `$_POST` array element exists. If it does, it's passed to `htmlentities()` to avoid any problems with quotes, and the resulting output is inserted into the `value` attribute using `echo`. It's very similar to the snippet you created in Chapter 11. Apart from *field*, the code never changes. This consistency makes it ideal for creating a new server behavior, which involves the following steps:

1. Create a unique name for each block of code that the server behavior will insert into your page. The Server Behavior Builder generates this automatically for you.
2. Type the code into the Server Behavior Builder, replacing any changeable values with Dreamweaver parameters. The parameters act as placeholders until you insert the actual value through a dialog box when the server behavior is applied.
3. Tell Dreamweaver where to insert the code.
4. Design the server behavior dialog box.

Creating a Sticky Text Field server behavior

These instructions show you how to create your own server behavior to insert a conditional statement in the `value` attribute of a text field to preserve user input in any page. You must have a PHP page open in the Document window before you start.

1. In the Server Behaviors panel, click the plus button, and select New Server Behavior. In the dialog box that opens, make sure that Document type is set to PHP MySQL. Type Sticky Text Field in the Name field, and click OK.
2. This opens the Server Behavior Builder dialog box. Click the plus button next to Code blocks to insert. Dreamweaver suggests a name for the new code block based

on the name of the new server behavior. Click OK to accept it. Dreamweaver fills in the remaining fields of the Server Behavior Builder, as shown in Figure 15-9.

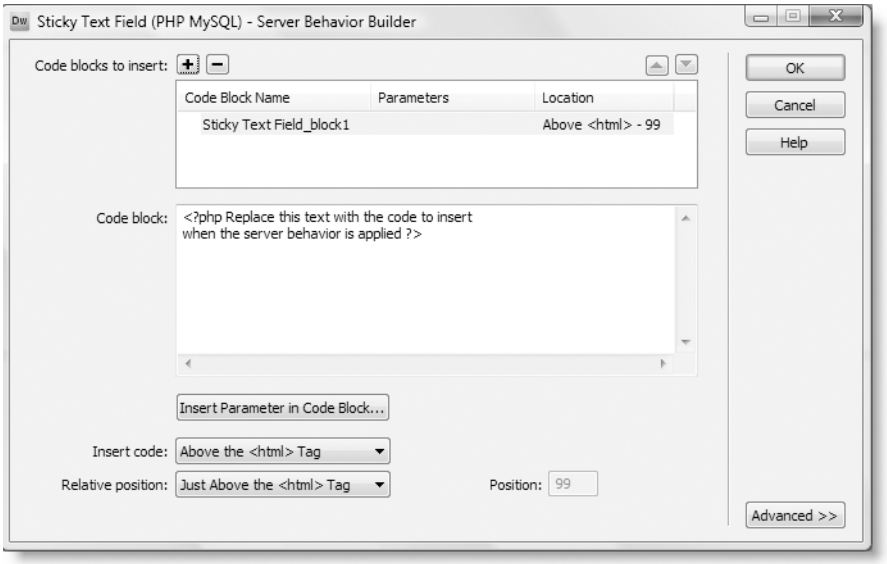


Figure 15-9. The Server Behavior Builder makes it easy to create your own server behaviors.

3. The Code block area in the center is where you insert the PHP code that you want to appear on the page. The value of *field* will change every time, so you need to replace it with a parameter. Parameter names must not contain any spaces, but they are used to label the server behavior dialog box, so it's a good idea to choose a descriptive name, such as `FieldName`. To insert a parameter, click the Insert Parameter in Code Block button at the appropriate point in the code, type the name in the dialog box, and click OK. Dreamweaver places it in the code with two @ characters on either side. You can also type the parameters in the code block directly yourself. Whichever method you use, replace the dummy text in the Code block area with this:

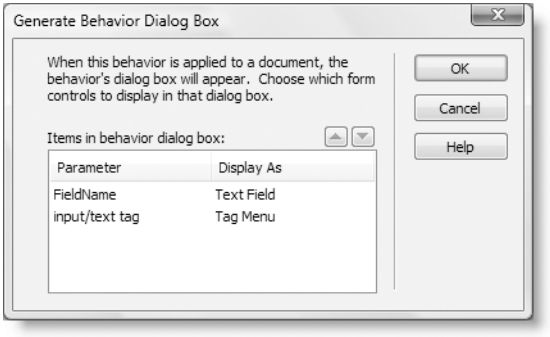
```
<?php if (isset($_POST['@@FieldName@@'])) {
echo htmlentities($_POST['@@FieldName@@'], ENT_COMPAT, 'UTF-8');} ?>
```

> *I am using the optional second and third arguments to* `htmlentities()`*, as described in Chapter 11. If you want to encode single quotes or are using a different encoding from Dreamweaver's default UTF-8, change the second and third arguments to suit your own requirements (see Tables 11-1 and 11-2 for the available options).*

4. As soon as you add any parameters in the Code block area, the label on the OK button changes to Next, but first you need to tell Dreamweaver where you want the code to appear in the page. It needs to be applied to the value attribute of `<input>` tags, so select Relative to a Specific Tag from the Insert code drop-down menu.
5. This reveals two more drop-down menus. Select input/text for Tag, and select As the Value of an Attribute for Relative position.
6. This triggers the appearance of another drop-down menu labeled Attribute. Select value. The bottom section of the Server Behavior Builder should now look like this:

This specifies that the code you entered in step 3 should be applied as the value attribute of a text field. Click Next at the top right of the Server Behavior Builder dialog box.

7. To be able to use your new server behavior, you need to create a dialog box where you can enter the values that will be substituted for the parameters. Dreamweaver does most of the work for you, and on this occasion, the suggestions in the Generate Behavior Dialog Box dialog box are fine, so just click OK.



Creating a server behavior for Sticky Text Areas

The server behavior you have just built works only with text fields, so it's worth building another to handle text areas. Unlike text fields, text areas don't have a value attribute.

1. Repeat steps 1 and 2 of the previous section, only this time call the new server behavior Sticky Text Area.
2. In step 3 of the previous section, enter the following code in the Code block area:

```
<?php if (isset($_POST['@@TextArea@@'])) {echo ➥
htmlentities($_POST['@@TextArea@@'], ENT_COMPAT, 'UTF-8');} ?>
```

I have split the code over two lines because of printing constraints, but you should enter the code all on a single line to avoid adding any whitespace between the `<textarea>` tags when this code is executed. Although the value is inserted directly between the tags as plain text, it's still a good idea to use `htmlentities()` to prevent malicious users from attempting to embed executable script, such as JavaScript, in your page.

3. Fill in the bottom section of the Server Behavior Builder, as shown in the following screenshot. This places the content of the `$_POST` variable between the opening and closing `<textarea>` tags.

4. Click Next, and accept the defaults suggested for the server behavior dialog box.

Both server behaviors will be available in all PHP sites from the menu in the Server Behaviors panel.

Completing the user registration form

Now that you have built your own server behaviors, you can complete `register_user.php`. What remains to be done is to redisplay the user's input if any errors are detected by the server-side validation. In the case of the text fields, this is done by the Sticky Text Field server behavior that you have just built. However, the radio buttons need to be handled differently. First, let's deal with the text fields.

Preserving user input in text fields

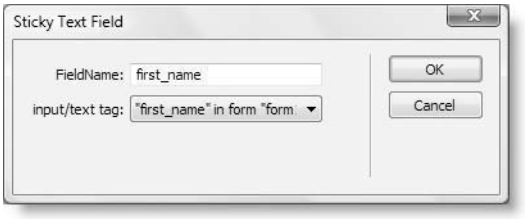
Applying the Sticky Text Field server behavior to each text field ensures that data already inserted won't be lost through the failure of any validation test.

Applying the Sticky Text Field server behavior

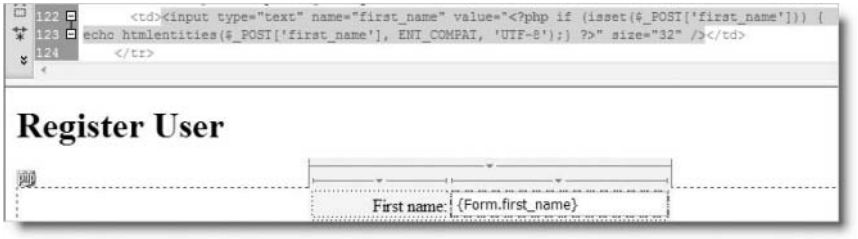
This section shows you how to use the Sticky Text Field server behavior. Continue working with `register_user.php` from earlier in the chapter.

1. In Design view, select the `first_name` text field. Click the plus button in the Server Behaviors panel. The new server behaviors are now listed. Select Sticky Text Field.
2. The Sticky Text Field dialog box appears. If you have selected the `first_name` text field correctly, the input/text tag field should automatically select first_name. If it's

not selected, activate the drop-down menu to select it. Type the field's name in FieldName, as shown here, and click OK:



3. Dreamweaver inserts a dynamic content placeholder inside the text field in Design view. Open Split view, and as the next screenshot shows, the conditional statement you created in the Code block area of the Server Behavior Builder has been inserted, but `@@FieldName@@` has been replaced by the actual name of the field:



4. Apply the Sticky Text Field server behavior to the `family_name` and `username` fields. Dreamweaver doesn't include password fields in the drop-down menu, so you can't apply the server behavior to them. In any case, the password is encrypted by `sha1()`, so you shouldn't attempt to redisplay it.
5. All instances of Sticky Text Field are now listed in the Server Behaviors panel. If you ever need to edit one, highlight it and double-click, or use the minus (–) button to remove it cleanly from your code.
6. Save `register_user.php`, and load it into a browser. Test it by entering an incomplete set of details. This time, the content of text fields is preserved. Check your code, if necessary, against `register_user_05.php` in `examples/ch15`.

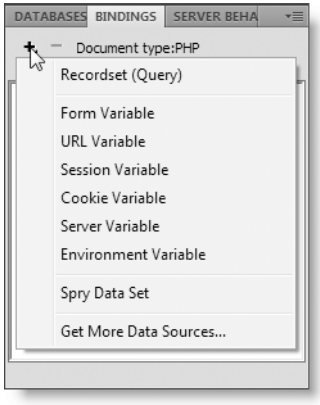
Applying a dynamic value to a radio group

The Administrator radio buttons still don't respond to the changes. We'll fix that next. Dreamweaver lets you bind the value of radio buttons to a dynamic value, such as from a recordset or a variable. You can type the variable directly into the dialog box, but Dreamweaver also lets you define superglobal variables, such as from the `$_POST` and `$_GET` arrays, for use throughout the site.

Making the radio buttons sticky

In this section, you'll define the $_POST variable that contains the value of the selected radio button and apply it to the radio button group so that it displays the value selected by the user when an error is detected. Continue working with register_user.php from the previous section.

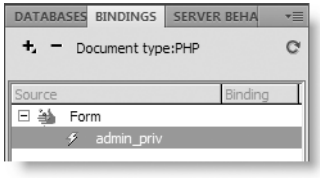
1. When any errors are detected, you need checked="checked" to be inserted in the tag of the radio button that the user selected. Since the radio group is called admin_priv, the value you want is contained in $_POST['admin_priv']. Although you can type this directly into the Dynamic Radio Group dialog box, Dreamweaver lets you define $_POST, $_GET, and other superglobal variables in the Bindings panel.

   In the Bindings panel, click the plus button to display the menu shown alongside.

   Dreamweaver uses generic names because the same menu applies to other server-side languages. As explained earlier, Form Variable refers to the $_POST array, and URL Variable refers to the $_GET array. You want to define a $_POST variable, so click Form Variable.

2. Type admin_priv in the Name field of the Form Variable dialog box, and click OK. The new dynamic variable is now listed in the Bindings panel like this:

3. Select one of the radio buttons in Design view, and click the Dynamic button in the Property inspector.

4. The admin_priv radio group will be automatically selected in the Dynamic Radio Group dialog box and grayed out, because the Record Insertion Form Wizard bound the value of the radio group to n. Change the binding by clicking the lightning bolt icon to the right of the Select value equal to field. Then choose admin_priv from the Dynamic Data panel (click the tiny plus sign or triangle alongside Form if you can't see admin_priv). Click OK twice to close both panels.

5. The problem with binding the value of the radio button group to $_POST['admin_priv'] is that this variable doesn't exist when the registration form first loads. As a result, neither radio button is selected. If PHP error reporting is set to its highest level, this displays unsightly error messages. And even if the display of errors is turned off, you're still left without a default radio button checked, which could lead to the user forgetting to select one and generating another error. So, this needs to be fixed—and it involves another journey into Code view.

In Design view, highlight one of the radio buttons so that you can easily locate the relevant code, and switch to Code view. The radio button code looks like this:

```
147           <td><input type="radio" name="admin_priv" value="y"  <?php if (!(strcmp($_POST['admin_priv'],"y")))
{echo "checked=\"checked\"";} ?> />
148             Yes</td>
149           </tr>
150           <tr>
151             <td><input type="radio" name="admin_priv" value="n"  <?php if (!(strcmp($_POST['admin_priv'],"n")))
{echo "checked=\"checked\"";} ?> />
152             No</td>
```

Dreamweaver uses a rather unusual PHP function called `strcmp()` to check whether `$_POST['admin_priv']` is `y` or `n`. The function takes two arguments and returns `0` if they're exactly the same. Since `0` equates to `false`, the negation operator (`!`) converts it to `true`. If you find the logic difficult to follow, just take my word for it—it works.

6. You need to check whether the form has been submitted. Although the `POST` array is always set, it will be empty if the form hasn't been submitted. And as you should know by now, an empty array equates to `false`. Amend the beginning of both sections of radio button code (shown on lines 147 and 151 in the preceding screenshot) like this:

   ```
   <input <?php if ($_POST && !(strcmp($_POST['admin_priv'],
   ```

7. Save the page, and load it into your browser. The radio buttons should now be back to normal. The only problem is that you don't have a default checked value when the page first loads. In one respect, it shouldn't be a problem, because you set a default value when defining the `users` table earlier. Unfortunately, Dreamweaver server behaviors treat unset values as `NULL`, causing your form to fail because `admin_priv` was defined as "not null."

8. Change the code for the `No` radio button shown on line 151 in the preceding screenshot like this (the change made in step 6 is also shown in bold):

   ```
   <input <?php if (($_POST && !(strcmp($_POST['admin_priv'],"n"))) ➥
   || !$_POST) {echo "checked=\"checked\"";} ?> name="admin_priv" ➥
   type="radio" value="n" />
   ```

   I have enclosed the original test (as adapted in step 6) in an extra pair of parentheses to ensure that it's treated as a single unit. Then I added a second test:

   ```
   || !$_POST
   ```

   This tests whether the `$_POST` array is empty. The result is this (in pseudocode):

   ```
   if ((the form has been sent AND admin_priv is "n")
   OR the form has not been sent) {mark the button "checked"}
   ```

9. Just one thing remains to be tidied up. If your PHP configuration has magic quotes turned on (and many hosting companies seem to use this setting), your sticky text fields will end up with backslashes escaping apostrophes in users' names. So, scroll

down to the section of code that displays the error messages, and insert a new line just before the closing curly brace. Open the Snippets panel, and insert the POST stripslashes snippet that you installed in the PHP-DWCS4 folder in Chapter 11. The amended code at the top of the body of the page should now look like this:

```
<?php
if (isset($error) && $error) {
  echo '<ul>';
  foreach ($error as $alert) {
    echo "<li class='warning'>$alert</li>\n";
  }
  echo '</ul>';
  // remove escape characters from POST array
  if (PHP_VERSION < 6 && get_magic_quotes_gpc()) {
    function stripslashes_deep($value) {
      $value = is_array($value) ? array_map('stripslashes_deep', ➥
        $value) : stripslashes($value);
      return $value;
    }
    $_POST = array_map('stripslashes_deep', $_POST);
  }
}
?>
```

10. Save register_user.php. You now have a user registration form that performs all the necessary checks before entering a new record into your database, but all the input fields will still be populated if an error is detected.

    Check your code, if necessary, against register_user_06.php in examples/ch15.

Building server-side validation into a simple user registration form has taken a lot of effort. You could have used the version from the previous chapter right away, but before long, you would have ended up with a lot of unusable data in your database, not to mention the frustration of users when an input error results in all their data being wiped from the screen. The more time you spend refining the forms that interact with your database, the more time you will save in the long run.

Applying server-side validation to the update form

The validation tests required by the update form are the same as those for the insert form, so there's considerably less new script involved. However, you need to take the following points into consideration:

- The password has been encrypted, so it can no longer be displayed in the update form. The code needs to be amended so that the password is updated only if a

value is inserted into the form. If the password fields are left empty, the original password is retained.

- When the update form first loads, it populates the form fields with values from the database, but you need to preserve any changes if the server-side validation detects errors when the form is submitted. This means adapting the Sticky Text Field server behavior to work with an update form.

Right, let's get to work.

Merging the validation and update code

Much of the work involved in adapting the code created by the Record Update Form Wizard can be done by copying and pasting the server-side validation code from the insert form.

Adapting update_user.php

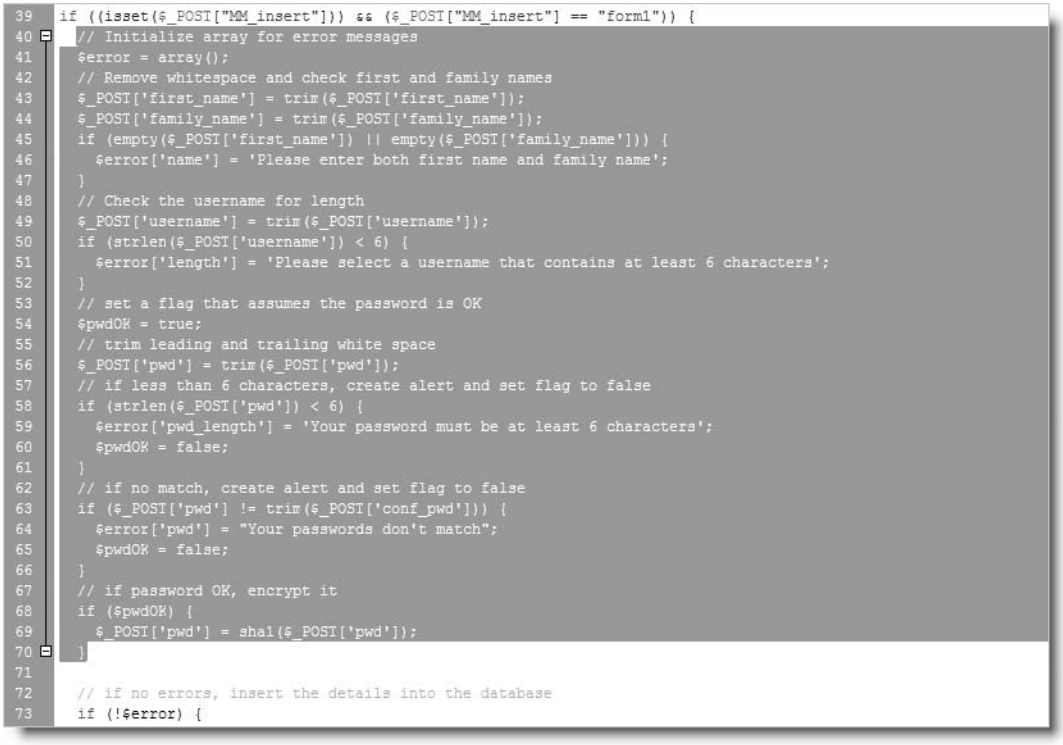
These instructions show how to apply the same validation tests to `update_user.php`. You can use your own version from the previous chapter. Alternatively, copy `update_user_start.php` from `examples/ch15` to `workfiles/ch15`, and rename it `update_user.php`. Continue working with the amended version of `register_user.php` from the preceding section. However, if you want to start with a clean copy, use `register_user_06.php` in `examples/ch15`.

1. Open both `register_user.php` and `update_user.php` in Code view.
2. In `update_user.php`, locate the conditional statement that controls the update server behavior, and insert a couple of blank lines, as shown in the following screenshot. This is where you will paste the validation script from `register_user.php`.

Paste validation script here

```
39  if ((isset($_POST["MM_update"])) && ($_POST["MM_update"] == "form1")) {
40
41
42    $updateSQL = sprintf("UPDATE users SET first_name=%s, family_name=%s, username=%s, pwd=%s, admin_priv=%s
    WHERE user_id=%s",
43                         GetSQLValueString($_POST['first_name'], "text"),
```

3. Switch to `register_user.php`, and copy the validation script shown highlighted in Figure 15-10.
4. Paste the code into `update_user.php` in the location indicated in the screenshot in step 2.



```
if ((isset($_POST["MM_insert"])) && ($_POST["MM_insert"] == "form1")) {
  // Initialize array for error messages
  $error = array();
  // Remove whitespace and check first and family names
  $_POST['first_name'] = trim($_POST['first_name']);
  $_POST['family_name'] = trim($_POST['family_name']);
  if (empty($_POST['first_name']) || empty($_POST['family_name'])) {
    $error['name'] = 'Please enter both first name and family name';
  }
  // Check the username for length
  $_POST['username'] = trim($_POST['username']);
  if (strlen($_POST['username']) < 6) {
    $error['length'] = 'Please select a username that contains at least 6 characters';
  }
  // set a flag that assumes the password is OK
  $pwdOK = true;
  // trim leading and trailing white space
  $_POST['pwd'] = trim($_POST['pwd']);
  // if less than 6 characters, create alert and set flag to false
  if (strlen($_POST['pwd']) < 6) {
    $error['pwd_length'] = 'Your password must be at least 6 characters';
    $pwdOK = false;
  }
  // if no match, create alert and set flag to false
  if ($_POST['pwd'] != trim($_POST['conf_pwd'])) {
    $error['pwd'] = "Your passwords don't match";
    $pwdOK = false;
  }
  // if password OK, encrypt it
  if ($pwdOK) {
    $_POST['pwd'] = sha1($_POST['pwd']);
  }

  // if no errors, insert the details into the database
  if (!$error) {
```

Figure 15-10. Most of the validation script can be copied and pasted into the update page.

5. There's just one change you need to make to the validation script you have pasted into `update_user.php`. When a user's record is being updated, you want either to preserve the same password or to set a new one. The simplest way to handle this is to decide that if `pwd` is left blank, the existing password will be maintained. Otherwise, the password needs to be checked and encrypted as before.

Amend the password validation code as follows (new code shown in bold):

```
$_POST['pwd'] = trim($_POST['pwd']);
// if password field is empty, use existing password
if (empty($_POST['pwd'])) {
  $_POST['pwd'] = $row_getUser['pwd'];
} else {
  // otherwise, conduct normal checks
  // if less than 6 characters, create alert and set flag to false
  if (strlen($_POST['pwd']) < 6) {
    $error['pwd_length'] = 'Your password must be at least 6➥
 characters';
    $pwdOK = false;
  }
```

```
    // if no match, create alert and set flag to false
    if ($_POST['pwd'] != trim($_POST['conf_pwd'])) {
      $error['pwd'] = 'Your passwords don\'t match';
      $pwdOK = false;
    }
    // if new password OK, encrypt it
    if ($pwdOK) {
      $_POST['pwd'] = sha1($_POST['pwd']);
    }
  }
```

This checks whether `$_POST['pwd']` is empty. If it is, the value of the existing password is taken from the `getUser` recordset and assigned to `$_POST['pwd']`. Because the existing password is already encrypted, there is no need to pass it to `sha1()`. If `$_POST['pwd']` isn't empty, the `else` clause executes the checks inherited from `register_user.php`.

6. You now need to prevent the update query from being executed if there are any errors. This involves wrapping the section of code immediately below the validation script in a conditional statement in the same way as in `register_user.php`. Figure 15-11 shows where to insert the code.

Figure 15-11. The conditional statement prevents the update code from being run if there are validation errors.

7. You also need to make the same changes as before to the code that runs the update query to catch any database errors and prevent the page from being redirected if any are found. Remove `or die(mysql_error())` shown on line 89 of Figure 15-11, and amend the code on lines 89–96 like this:

```
$Result1 = mysql_query($updateSQL, $connAdmin);
if (!$Result1 && mysql_errno() == 1062) {
  $error['username'] = $_POST['username'] . ' is already in use. ➥
    Please choose a different username.';
} elseif (mysql_error()) {
  $error['dbError'] = 'Sorry, there was a problem with the ➥
    database. Please try later.';
} else {
  $updateGoTo = "list_users.php";
  if (isset($_SERVER['QUERY_STRING'])) {
    $updateGoTo .= (strpos($updateGoTo, '?')) ? "&" : "?";
    $updateGoTo .= $_SERVER['QUERY_STRING'];
  }
  header(sprintf("Location: %s", $updateGoTo));
}
}
```

   You can copy and paste the first two conditions from `register_user.php`, because they are identical. Don't forget to add the closing curly brace after the code that redirects to the next page.

8. That deals with the changes to the validation script in Code view, but the update form doesn't have the password confirmation field. You also need to add some text to inform the user to leave the password fields blank if the same password is to be kept.

   So, switch to Design view, and add (leave blank if unchanged) to the Password label.

9. The original update form showed the password in plain text, so select the `pwd` field, and change the Type radio button from Single line to Password in the Property inspector.

10. Create a new table row between Password and Administrator. Type Confirm password as the label in the left cell, and insert a text field in the right cell. Name the text field `conf_pwd`, and set Type to Password in the Property inspector.

11. The change you made to the password validation in step 6 compares `$_POST['pwd']` with `$row_getUser['pwd']`. However, as I explained at the beginning of the chapter, Dreamweaver always inserts the code for recordsets immediately above the `DOCTYPE` declaration. Consequently, `$row_getUser['pwd']` won't have been created unless you move the recordset script to an earlier position.

Cut the recordset code shown on lines 105–113 of the following screenshot, and paste it in the position indicated (I used Code Collapse to hide most of the validation script).

```
34  $editFormAction = $_SERVER['PHP_SELF'];
35  if (isset($_SERVER['QUERY_STRING'])) {
36    $editFormAction .= "?" . htmlentities($_SERVER['QUERY_STRING']);
37  }
38
39  if ((isset($_POST["MM_update"])) && ($_POST["MM_update"] == "form1")) {
40    // Initialize array for error messages
41    $error = array();
42  // Remo...
100       header(sprintf("Location: %s", $updateGoTo));
101     }
102   }
103 }
104
105 $colname_getUser = "-1";
106 if (isset($_GET['user_id'])) {
107   $colname_getUser = $_GET['user_id'];
108 }
109 mysql_select_db($database_connAdmin, $connAdmin);
110 $query_getUser = sprintf("SELECT user_id, username, pwd, first_name, family_name, admin_priv FROM users WHERE
    user_id = %s", GetSQLValueString($colname_getUser, "int"));
111 $getUser = mysql_query($query_getUser, $connAdmin) or die(mysql_error());
112 $row_getUser = mysql_fetch_assoc($getUser);
113 $totalRows_getUser = mysql_num_rows($getUser);
114 ?>
115 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
```

12. Save the page, and leave it open for the next section. There have been a lot of important changes, so check your code against `update_user_01.php` in `examples/ch15`.

The final set of changes you need to make to the update page involves removing the existing code that binds the values from the database to the input fields and replacing it with code that not only displays the values retrieved from the database but also preserves the user's input if there are any errors when the update form is submitted. The Sticky Text Field server behavior won't work in these circumstances, but it's easy to adapt.

Adapting the Sticky Text Field server behavior

As you have already seen, it's only when the form has been submitted—and errors detected—that the Sticky Text Field code executes. So if the `$_POST` variables haven't been set, you know the form hasn't been submitted and that you need to display the values stored in the database instead.

Dreamweaver always uses the following naming convention to refer to the results of a recordset: `$row_`*`RecordsetName`*`['`*`FieldName`*`']`. So, all that's needed is to add an `else` clause to the existing code:

```
<?php if (isset($_POST['field'])) {
  echo htmlentities($_POST['field'], ENT_COMPAT, 'UTF-8');
} else {
  echo htmlentities($row_RecordsetName['FieldName'], ENT_COMPAT, ➥
    'UTF-8');
} ?>
```

Most of the settings are identical to the Sticky Text Field server behavior that you built earlier, so you can use the existing server behavior to create the new one.

1. Make sure you have a PHP page open, and click the plus button in the Server Behaviors panel. Select New Server Behavior.
2. Name the new server behavior Sticky Edit Field, and place a check mark in the box labeled Copy existing server behavior. This will populate a drop-down menu with the names of server behaviors you have already built (unfortunately, the dialog box won't let you base a new server behavior on one of Dreamweaver's). Select Sticky Text Field, and click OK.
3. Edit the contents of the Code block area like this:

```
<?php if (isset($_POST['@@FieldName@@'])) {
  echo htmlentities($_POST['@@FieldName@@'], ENT_COMPAT, 'UTF-8');
} else {
  echo htmlentities($row_@@RecordsetName@@['@@FieldName@@'], ➥
    ENT_COMPAT, 'UTF-8');
} ?>
```

   Dreamweaver will use the new parameter—`@@RecordsetName@@`—in combination with `@@FieldName@@` to build a variable like `$row_getUser['family_name']`.

   > *Sometimes Dreamweaver prevents you from using the same parameter name in more than one server behavior. If that happens, change both instances of* `@@FieldName@@` *to* `@@Field@@`.

4. Click Next. Dreamweaver warns you that the server behavior's HTML file already exists and asks whether you want to overwrite it. The HTML file is actually a copy, so there's no problem overwriting it. It controls the server behavior's dialog box, which needs to be redesigned, so the answer is Yes.
5. In the Generate Behavior Dialog Box dialog box, reset Display as for RecordsetName by clicking to the right of the existing value and selecting Recordset Menu. Set FieldName to Recordset Field Menu, and reorder the items as shown here. Click OK.

To create a similar server behavior for text areas, name it Sticky Edit Area, and select Sticky Text Area in step 2. The code block in step 3 is identical for both Sticky Edit Area and Sticky Text Area.

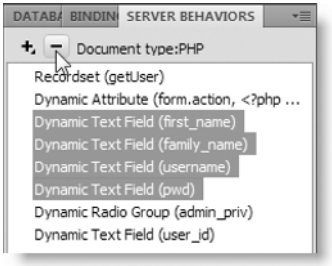
Binding the field values to the update form

Now that you have the Sticky Edit Field server behavior, you can bind the results of the getUser recordset to the form fields so that the existing values are ready for editing but will be replaced by the user's input if the update process fails for any reason. The text fields are quite easy, but the radio button group needs special handling.

Completing the update form

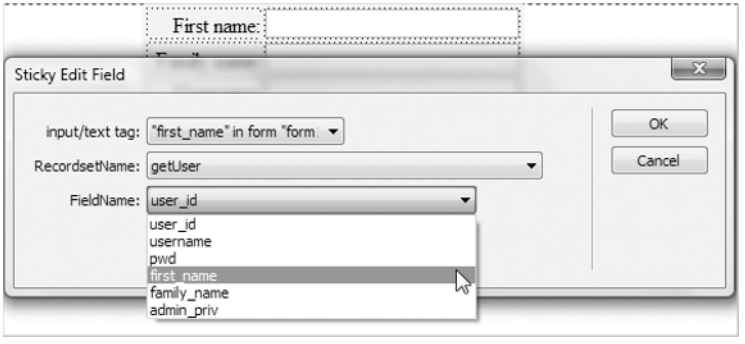
These instructions show how to apply the Sticky Edit Field server behavior and adapt the code in the radio button group. Continue working with update_user.php from before.

1. Before you can apply the Sticky Edit Field server behavior, you need to remove the existing code from the form fields. In the Server Behaviors panel, Shift-click to select the Dynamic Text Field entries for first_name, family_name, username, and pwd. Then click the minus button, as shown in the screenshot alongside, to remove them cleanly from the update form.

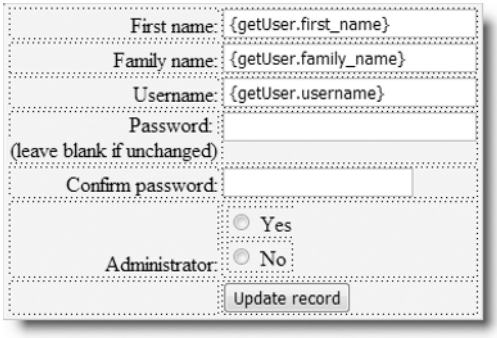


2. In Design view, select the first_name field, click the plus button in the Server Behaviors panel, and select Sticky Edit Field.

   Since getUser is the only recordset on this page, it's selected automatically in the Sticky Edit Field dialog box, but make sure you choose the right one if you use this server behavior on a page that has two or more recordsets. Select the field's name from the FieldName drop-down menu, as shown here:



3. Apply the Sticky Edit Field server behavior in the same way to the family_name and username fields. In Design view, the form should end up looking like the following screenshot, with dynamic text placeholders in the first three fields.



The dynamic text placeholders in the three fields look the same as before in Design view, but if you inspect the underlying code in Split view, you'll see that Dreamweaver has inserted the code you used to build the Sticky Edit Field server behavior.

4. The radio buttons present an interesting challenge. When the page first loads, you want the value stored in the database for `admin_priv` to be selected; but if the form is submitted with errors and the value of `admin_priv` has been changed, you want the new value to be shown.

   Select one of the radio buttons in Design view to help locate the code for the radio group; then switch to Code view to actually see it. The code looks like this:

```
160           <td><input type="radio" name="admin_priv" value="y" <?php if (!(strcmp(htmlentities($row_getUser[
    'admin_priv'], ENT_COMPAT, 'utf-8'),"y"))) {echo "checked=\"checked\"";} ?> />
161             Yes</td>
162         </tr>
163         <tr>
164           <td><input type="radio" name="admin_priv" value="n" <?php if (!(strcmp(htmlentities($row_getUser[
    'admin_priv'], ENT_COMPAT, 'utf-8'),"n"))) {echo "checked=\"checked\"";} ?> />
165             No</td>
```

   Let's first map out in terms of pseudocode what needs to happen inside the Yes radio button's `<input>` tag. The logic goes like this:

```
if (the form has NOT been submitted
   AND the value of admin_priv in the database is "y") {
     mark the button "checked"
} elseif (the form has been submitted
          AND the form value of admin_priv is "y") {
    mark the button "checked"
}
```

   You can create this code by copying and pasting the existing conditional statements and making a few changes. It's not difficult, but you need to follow the next steps carefully.

5. When the page first loads, the form hasn't been submitted, so the `$_POST` array will have zero elements (and therefore equate to `false`). This means the first check can be performed by inserting `!$_POST` into the conditional statement like this:

```
if (!$_POST && !(strcmp(htmlentities($row_getUser['admin_priv'], ➥
ENT_COMPAT, 'utf-8'),"y"))) {echo "checked=\"checked\"";}
```

6. You now need to deal with the alternative scenario. Begin by copying the amended conditional statement and pasting it immediately after the closing curly brace. So, now you have two identical conditional statements.
7. You want the second statement to run only if the first one fails, so change the second `if` to `elseif`.
8. In the alternative scenario, you want `$_POST` to be `true`, so remove the negative operator from in front of `$_POST`.
9. You also want the value of `admin_priv` to come from the form input, rather than the database, so change `$row_getUser['admin_priv']` to `$_POST['admin_priv']`.
10. Repeat steps 5–9 for the No button. The completed radio button code looks like this:

```
  <td><input type="radio" name="admin_priv" value="y"
<?php if (!$_POST && !(strcmp(htmlentities($row_getUser['admin_priv'], ➥
ENT_COMPAT, 'utf-8'),"y"))) {echo "checked=\"checked\"";}
elseif ($_POST && !(strcmp(htmlentities($_POST['admin_priv'], ➥
ENT_COMPAT, 'utf-8'),"y"))) {echo "checked=\"checked\"";} ?> />
Yes</td>
</tr>
<tr>
  <td><input type="radio" name="admin_priv" value="n"
<?php if (!$_POST && !(strcmp(htmlentities($row_getUser['admin_priv'], ➥
ENT_COMPAT, 'utf-8'),"n"))) {echo "checked=\"checked\"";}
elseif ($_POST && !(strcmp(htmlentities($_POST['admin_priv'], ➥
ENT_COMPAT, 'utf-8'),"n"))) {echo "checked=\"checked\"";} ?> />
No</td>
```

11. One more thing, and you're done. Copy the code that displays the error messages from `register_user.php` (shown on lines 107–123 of the following screenshot), and paste it just above the update form in `update_user.php`.

```
106  <h1>Register User</h1>
107  <?php
108  if (isset($error)) {
109    echo '<ul>';
110    foreach ($error as $alert) {
111      echo "<li class='warning'>$alert</li>\n";
112    }
113    echo '</ul>';
114    // remove escape characters from POST array
115    if (PHP_VERSION < 6 && get_magic_quotes_gpc()) {
116      function stripslashes_deep($value) {
117        $value = is_array($value) ? array_map('stripslashes_deep', $value) : stripslashes($value);
118        return $value;
119        }
120      $_POST = array_map('stripslashes_deep', $_POST);
121    }
122  }
123  ?>
124  <form action="<?php echo $editFormAction; ?>" method="post" name="form1" id="form1">
```

12. Save `update_user.php`. Compare your code with `update_user_02.php` in `examples/ch15` if you have any problems.

You can now update existing records by loading `list_users.php` into a browser and clicking the EDIT link alongside the username of the account you want to change. Adapting the update form has also required considerable effort. It's a pity that Dreamweaver doesn't offer more help in the way of server-side validation, but if you value your data, you need to customize the code that Dreamweaver creates for you.

You might want to take a break at this stage, but now that you have a simple user registration system, you can use it to password protect various parts of your website. You'll be relieved to know that Dreamweaver's user authentication server behaviors don't need anywhere near the same level of customization. They rely on the use of PHP sessions, so before showing you how to build a login system, let's take a quick look at sessions and what they're for.

What sessions are and how they work

The Web is a brilliant illusion. When you visit a well-designed website, you get a great feeling of continuity, as though flipping through the pages of a book or a magazine. Everything fits together as a coherent entity. The reality is quite different. Each part of an individual page is stored and handled separately by the web server. Apart from needing to know where to send the relevant files, the server has no interest in who you are, nor is it interested in the PHP script it has just executed. PHP **garbage collection** (yes, that's what it's actually called) destroys variables and other resources used by a script as soon as they're no longer required. But it's not like garbage collection at your home, where it's taken away, say, once a week. With PHP, it's instant: the server memory is freed up for the next task. Even variables in the `$_POST` and `$_GET` arrays persist only while being passed from one page to the next. Unless the information is stored in some other way, such as a hidden form field, it's lost.

To get around these problems, PHP (in common with other server-side languages) uses **sessions**. A session ensures continuity by storing a random identifier on the web server and on the visitor's computer (as a cookie). Because the identifier is unique to each visitor, all the information stored in session variables is directly related to that visitor and cannot be seen by anyone else.

> *The security offered by sessions is adequate for most user authentication, but it is not 100-percent foolproof. For credit card and other financial transactions, you should use an SSL connection verified by a digital certificate. To learn more about this and other aspects of building security into your PHP sites,* Pro PHP Security *by Chris Snyder and Michael Southwell (Apress, ISBN: 978-1-59059-508-4) is essential reading. Although aimed at readers with an intermediate to advanced knowledge of PHP, it contains a lot of practical advice of value to all skill levels.*

Creating PHP sessions

Creating a session is easy. Just put this command in every PHP page that you want to use in a session:

```
session_start();
```

Once you call that command, the page has access to the visitor's session variables. This command should be called only once in each page, and it must be called before the PHP script generates any output, so the ideal position is immediately after the opening PHP tag. If any output is generated before the call to `session_start()`, the command fails, and the session won't be activated for that page. Even a single blank space, newline character, or byte-order mark is considered output. This is the same issue that affects the `header()` function, if any output is generated before you call the function. The solution is the same and was described in "Avoiding the 'Headers already sent' error" in Chapter 12.

Creating and destroying session variables

You create a session variable by adding it to the `$_SESSION` superglobal array in the same way you would assign an ordinary variable. Say you want to store a visitor's name and display a greeting. If the name is submitted in a login form as `$_POST['name']`, you assign it like this:

```
$_SESSION['name'] = $_POST['name'];
```

`$_SESSION['name']` can now be used in any page that begins with `session_start()`. Because session variables are stored on the server, you should get rid of them as soon as they are no longer required by your script or application. Unset a session variable like this:

```
unset($_SESSION['name']);
```

To unset *all* session variables—for instance, when you're logging someone out—set the `$_SESSION` superglobal array to an empty array, like this:

```
$_SESSION = array();
```

> *Do not be tempted to try* `unset($_SESSION)`. *It not only clears the current session but also prevents any further sessions from being stored.*

Destroying a session

By itself, unsetting all the session variables effectively prevents any of the information from being reused, but you should also destroy the session with the following command:

```
session_destroy();
```

By destroying a session like this, there is no risk of an unauthorized person gaining access either to a restricted part of the site or to any information exchanged during the session. However, a visitor may forget to log out, so it's not always possible to guarantee that the `session_destroy()` command will be triggered, which is why it's so important not to store sensitive information in a session variable.

> *You may find the deprecated functions* `session_register()` *and* `session_unregister()` *in old scripts. Use* `$_SESSION['variable_name']` *and* `unset($_SESSION['variable_name'])` *instead.*

Checking that sessions are enabled

Sessions should be enabled by default in PHP. A quick way to check is to load `session1.php` in `examples/ch15` into a browser. Type your name in the text field, and click the Submit button. When `session2.php` loads, you should see your name and a link to the next page. Click the link. If `session3.php` displays your name and a confirmation that sessions are working, your setup is fine. Click the link to page 2 to destroy the session.

If you don't see the confirmation on the third page, create a PHP page containing the single line of code `<?php phpinfo(); ?>` to display details of your PHP configuration. Make sure that `session.save_path` points to a valid folder that the web server can write to. Also make sure that a software firewall or other security system is not blocking access to the folder specified in `session.save_path`.

Registering and authenticating users

As you have just seen, session variables enable you to keep track of a visitor. If you can identify visitors, you can also determine whether they have the right to view certain pages. Dreamweaver has four user authentication server behaviors, as follows:

- **Log In User:** This queries a database to check whether a user is registered and has provided the correct password. You can also check whether a user belongs to a particular group to distinguish between, say, administrators and ordinary users.
- **Restrict Access to Page:** This prevents visitors from viewing a page unless they have logged in and (optionally) have the correct group privileges. Anyone not logged in is sent to the login page but can be automatically redirected to the originally selected page after login.
- **Log Out User:** This brings the current session to an end and prevents the user from returning to any restricted page without first logging back in again.
- **Check New Username:** This checks whether a particular username is already in use. I don't recommend using it, because it's rather badly designed. Using a unique index and testing for MySQL error 1062, as described earlier in this chapter, is more user-friendly.

Creating a login system

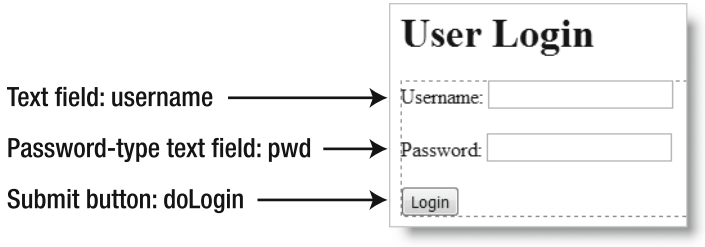
Now that you have a way of registering users, you need to create a way for them to log in to restricted areas of your site. Building the login system is a lot simpler than building the registration system.

> *The login system uses encrypted passwords. You must encrypt the passwords of records that were created with the forms from the previous chapter before server-side validation was added. Do this by clicking the* EDIT *link in* `list_users.php` *and reentering the password in the update form.*

Creating the login page

The first element of a login system is the form where registered users enter their username and password. To keep things simple, the following instructions use a dedicated login page, but you can embed the login form on any public page of a site.

1. Create a PHP page called `login.php` in `workfiles/ch15`. Lay out the page with a form, two text fields, and a submit button, as shown here. Since you'll be applying a server behavior, there is no need to set the `action` or `method` attributes of the form.



2. The Log In User server behavior expects you to designate two pages: one that the user will be taken to if the login is successful and another if it fails. Create one page called `success.php`, and enter some content to indicate that the login was successful. Call the other page `loginfail.php`, and insert a message telling the user that the login failed, together with a link back to `login.php`.
3. Make sure `login.php` is the active page in the Dreamweaver workspace. Click the plus button in the Server Behaviors panel, and select User Authentication ➤ Log In User. (You can also apply the server behavior from the Data tab of the Insert bar or from the Data Objects submenu of the Insert menu.)

4. The Log In User dialog box has a lot of options, but their meaning should be obvious, at least for the first two sections. Select the connAdmin connection, the users table, and the username and password columns, using the settings shown alongside.

The third section asks you to specify which pages to send the user to, depending on whether the login succeeds or fails. Between the text fields for the filenames is a check box labeled Go to previous URL (if it exists). This works in conjunction with the Restrict Access to Page server behavior that you will use shortly. If someone tries to access a restricted page without first logging in, the user is redirected to the login page. If you select this option, after a successful login, the user will be taken directly to the page that originally refused access. Unless you always want users to view a specific page when first logging in, this is quite a user-friendly option.

The final section of the dialog box allows you to specify whether access should be restricted on the basis of username and password (the default) or whether you also want to specify an access level. The access level must be stored in one of your database columns. For this login page, set Get level from to admin_priv. Click OK to apply the server behavior.

5. A drawback with the Dreamweaver Log In User server behavior is that it has no option for handling encrypted passwords, so you need to make a minor adjustment by hand. Open Code view, and place your cursor immediately to the right of the opening PHP tag on line 2. Press Enter/Return to insert a new line, and type the following code:

```
if (isset($_POST['pwd'])) { $_POST['pwd'] = sha1($_POST['pwd']); }
```

This checks whether the form has been submitted, and it uses sha1() to encrypt the password. I have reassigned the value back to $_POST['pwd'] so that Dreamweaver continues to recognize the server behavior; this way, you can still edit it through the Server Behaviors panel. Although Dreamweaver doesn't object to you placing the line of code here, it will automatically remove it if you ever decide to remove the server behavior.

> *It's important to realize that you're not decrypting the version of the password stored in the database. You can't—the* sha1() *function performs one-way encryption. You verify the user's password by encrypting it again and comparing the two encrypted versions.*

6. Save login.php. You can check your code against login.php in examples/ch15.

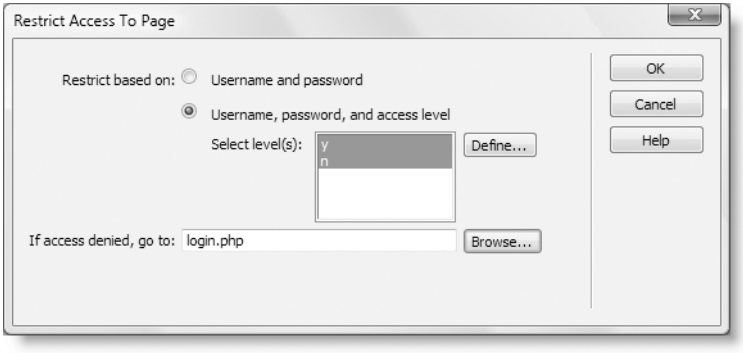
Restricting access to individual pages

Now that you have a means of logging in registered users, you can protect sensitive pages in your site. When working with PHP sessions, there is no way of protecting an entire folder. Sessions work on a page-by-page basis, so you need to protect each page individually.

1. Open `success.php`. Click the plus button in the Server Behaviors panel, and select User Authentication ➤ Restrict Access to Page.
2. In the Restrict Access to Page dialog box, select the radio button to restrict access based on Username, password, and access level. Then click the Define button.
3. The Define Access Levels dialog box lets you specify acceptable values. What may come as a bit of a surprise is that it's not the column name that Dreamweaver is interested in but the value retrieved from the column. Consequently, it's not admin_priv that you enter here but y or n.

   As you might have noticed, although Dreamweaver gives you the option to specify different access levels, the Log In User server behavior sends all successful logins to the same page. If you have different login pages for each type of user, this is fine; you select the appropriate value. So, for an administrator's login page, just enter y in the Name field, and click the plus button to register it in the Access levels area.

   However, if you want to use the same login form for everyone, you need to register all access levels for the first page and then use PHP conditional logic to distinguish between different types of users. So, for `success.php`, also enter n in the Name field, and click the plus button to register it. Then click OK.
4. After defining the access levels, hold down the Shift key, and select them all in the Select level(s) field. Then, either browse to login.php, or type the filename directly in the field labeled If access denied, go to. The dialog box should look like this:



5. Click OK to apply the server behavior, and save `success.php`.
6. Try to view the page in a browser. Instead of `success.php`, you should see `login.php`. You have been denied access and taken to the login page instead.
7. Enter a username and password that you registered earlier, and click Log in. You should be taken to `success.php`. You can check your code against `success_01.php` in `examples/ch15`.

When developing pages that will be part of a restricted area, I find it best to leave the application of this server behavior to the very last. Testing pages becomes an exercise in frustration if you need to be constantly logging in and out.

I'll come back to the question of how to deal with different access levels, but first, let's look at logging out.

Logging out users

The Dreamweaver Log Out User server behavior is quick and easy to apply. It automatically inserts a logout link in your page, so you need to position your cursor at the point you want the link to be created.

1. Press Enter/Return to create a new paragraph in `success.php`.
2. Click the plus button in the Server Behaviors panel, and select User Authentication ➤ Log Out User.
3. The Log Out User dialog box gives you the option to log out when a link is clicked or when the page loads. In this case, you want the default option, which is to log out when a link is clicked and to create a new logout link. Browse to `login.php`, or type the filename directly into the field labeled When done, go to. Click OK.
4. Save `success.php`, and load the page into a browser. Click the Log out link, and you will be taken back to the login page. Type the URL of `success.php` in the browser address bar, and you will be taken back to the login page until you log in again. You can check your code against `success_02.php` in `examples/ch15`.

Displaying different content depending on access levels

As I mentioned earlier, PHP sessions are the technology that lies behind the user authentication server behaviors. The Log In User server behavior creates the following two session variables that control access to restricted pages:

- `$_SESSION['MM_Username']`: This stores the user's username.
- `$_SESSION['MM_UserGroup']`: This stores the user's access level.

You can use these in a variety of ways. The simplest, and perhaps most important, use is to present different content on the first page after logging in. The following exercises are based on `success.php` but can be used with any page that begins with `session_start()` after a user has logged in.

> *The following instructions assume you have created at least one administrator and an ordinary user in the* `users` *table.*

1. In `success.php`, insert two paragraphs: one indicating that it's for administrators, the other indicating that it's for non-administrators. The actual content is unimportant.

2. Switch to Code view, and add the PHP code highlighted in bold around the two paragraphs like this:

```
<?php if ($_SESSION['MM_UserGroup'] == 'y') { ?>
<p>Content and links for administators</p>
<?php } else { ?>
<p>Content and links for non-administrators</p>
<?php } ?>
```

   This is simple PHP conditional logic. If the value of `$_SESSION['MM_UserGroup']` is y, display the HTML inside the first set of curly braces. If it's not, show the other material. There's only one paragraph in each conditional block, but you can put as much as you want.

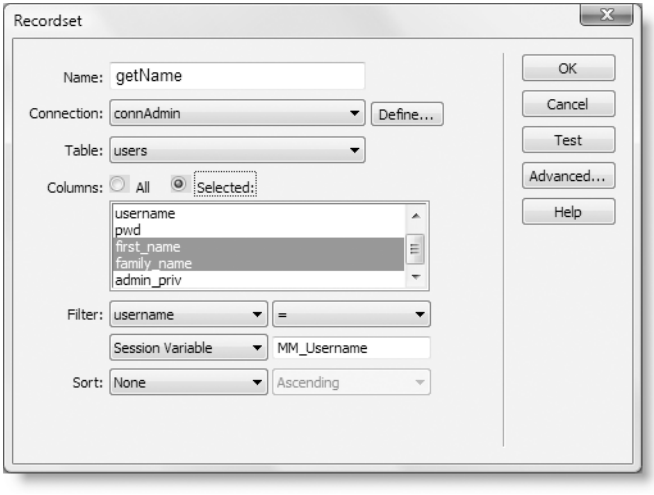
3. Save the page, and log in as an administrator. You'll see only the first paragraph. Log out and log back in as an ordinary user. This time you'll see the second paragraph. You can compare your code with `success_03.php` in `examples/ch15`.

Any content that you want to be seen by both groups should go outside this PHP conditional statement. (In `success_03.php`, you'll see that the page heading and the log out link are common to both groups.) By using this sort of branching logic in the first page, you can restrict access to subsequent pages according to the specific access level. So, the links in the first section would point to pages that only administrators are permitted to see.

Greeting users by name

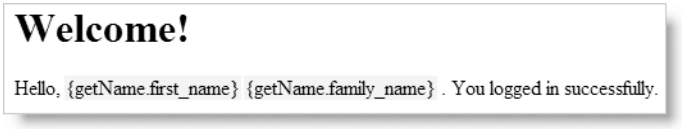
Since the user's username is stored in `$_SESSION['MM_Username']`, you could use that to display a greeting, but it's much friendlier to use the person's real name. All that's needed is a simple recordset.

1. In `success.php`, create a recordset using the following settings in Simple mode:



By setting Filter to username = Session Variable MM_Username, the recordset retrieves the values of the first_name and family_name columns for the currently logged in user.

2. Open the Bindings panel, and drag the first_name and family_name dynamic text placeholders into the page like this:



When the page loads, the dynamic text placeholders will be replaced by the values drawn from the recordset. You can check your code against success_04.php.

Of course, if you want other details about the user, such as user_id, amend the settings in the Recordset dialog box to retrieve all the columns you need.

Creating your own $_SESSION variables from user details

To avoid the need to create a recordset on every page, store these details as $_SESSION variables. The code needs to be inserted *after* the recordset code, which Dreamweaver places immediately above the DOCTYPE declaration. The pattern Dreamweaver uses for recordset results looks like this:

```
$row_recordsetName['fieldName']
```

So, to create $_SESSION variables from first_name and family_name in session.php, you would add the following code immediately before the closing PHP tag above the DOCTYPE declaration:

```
$_SESSION['first_name'] = $row_getName['first_name'];
$_SESSION['family_name'] = $row_getName['family_name'];
```

You're not restricted to using the same element names for the variables. You could do this instead:

```
$_SESSION['full_name'] = $row_getName['first_name'].' '. ➥
$row_getName['family_name'];
```

You can see this code in action in success_05.php in examples/ch15.

Redirecting to a personal page after login

You might want to provide users with their own personal page or folder after logging in. This is very easy to do, particularly if you base the name of the personal name or folder on the username. Before the Log In User server behavior creates the session variables, it stores the submitted username as $loginUsername, so you can use this variable to redirect users to pages or folders based on their username.

If the name of the personal page is in the form *username*.php, enter the following in the Log In User dialog box in the field labeled If login succeeds, go to (see step 4 of "Creating the login page"):

```
$loginUsername.php
```

If the personal page is in a folder named after the username, use the following:

```
$loginUsername/index.php
```

This assumes that the folder is a subfolder of the folder where the login page is located. If the username is dpowers, these values would redirect the user to dpowers.php and dpowers/index.php, respectively.

Encrypting and decrypting passwords

These are common questions: What happens when a user forgets his or her password? How can I send a reminder? If you encrypt passwords using sha1(), as described in this chapter, you can't. The sha1() algorithm is one-way; you can't decrypt it. Although this sounds like a disadvantage, it actually ensures a considerable level of security. Since the password cannot be decrypted, even a corrupt system administrator has no way of discovering another person's password. The downside is that you can't send out password reminders.

If a password is forgotten, you need to verify the user's identity and issue a new password. You can also create a form for users to change their own passwords after logging in. It's simply a question of using $_SESSION['MM_Username'] as the filter for the Update Record server behavior. Don't worry if you feel that's currently beyond your capability. In the next chapter, you'll learn about the four basic SQL commands that are the key to database management.

However, it is possible to store passwords using two-way encryption. For more information, see my book *PHP Solutions: Dynamic Web Design Made Easy* (friends of ED, ISBN: 978-1-59059-731-6) and the MySQL documentation at http://dev.mysql.com/doc/refman/5.0/en/encryption-functions.html.

Chapter review

If you're beginning to wobble because of the constant need to dive into Code view, take heart. This has been a tough chapter. The danger with Dreamweaver server behaviors is they make it very easy to create record insertion and update forms, giving you a false sense of achievement. If you're just creating a dynamic website as a hobby, you might be happy with minimum checks on what's inserted into your database. But even if it's a hobby, do you really want to waste your time on a database that gets filled with unusable data? And if you're doing it professionally, you simply can't afford to.

PHP is like the electricity or kitchen knives in your home: handled properly, it's very safe; handled irresponsibly, it can do a lot of damage. Get to know what the code you're putting into your pages is doing. The more hands-on experience you get, the easier it becomes. A lot of PHP coding is simple logic: if this, do one thing; else do something different.

Take a well-earned rest. In the next chapter, we'll delve into the mysteries of SQL, the language used to communicate with most databases, and joining records from two or more tables.