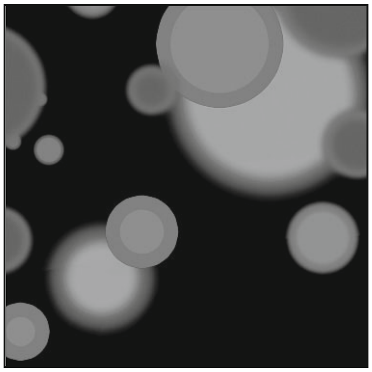


Chapter 11

Connecting to MySQL with PHP and SQL

PHP offers three different ways to connect to and interact with a MySQL database: the original MySQL extension, MySQL Improved (MySQLi), or PHP Data Objects (PDO). Which one you choose is an important decision, because they use incompatible code. You can't mix them in the same script. The original MySQL extension is no longer actively developed and is not recommended for new PHP/MySQL projects. It's not covered in this book.

The PHP documentation describes MySQLi as the preferred option recommended by MySQL for new projects. However, that doesn't mean you should discount PDO. The advantage of PDO is that it's software-neutral. In theory, at least, you can switch your website from MySQL to Microsoft SQL Server or a different database system by changing only a couple of lines of PHP code. In practice, you normally need to rewrite at least some of your SQL queries because each database vendor adds custom functions on top of standard SQL. Still, it's simpler than switching from MySQLi, which works exclusively with MySQL. Switching a MySQLi script to a different database involves rewriting all of the PHP code in addition to any changes needed to the SQL.

If you have no plans to use a database other than MySQL, I recommend that you use MySQLi. It's designed specifically to work with MySQL. Just ignore the sections on PDO. On the other hand, if database flexibility is important to you, choose PDO. Both methods are covered in the remaining chapters of this book.

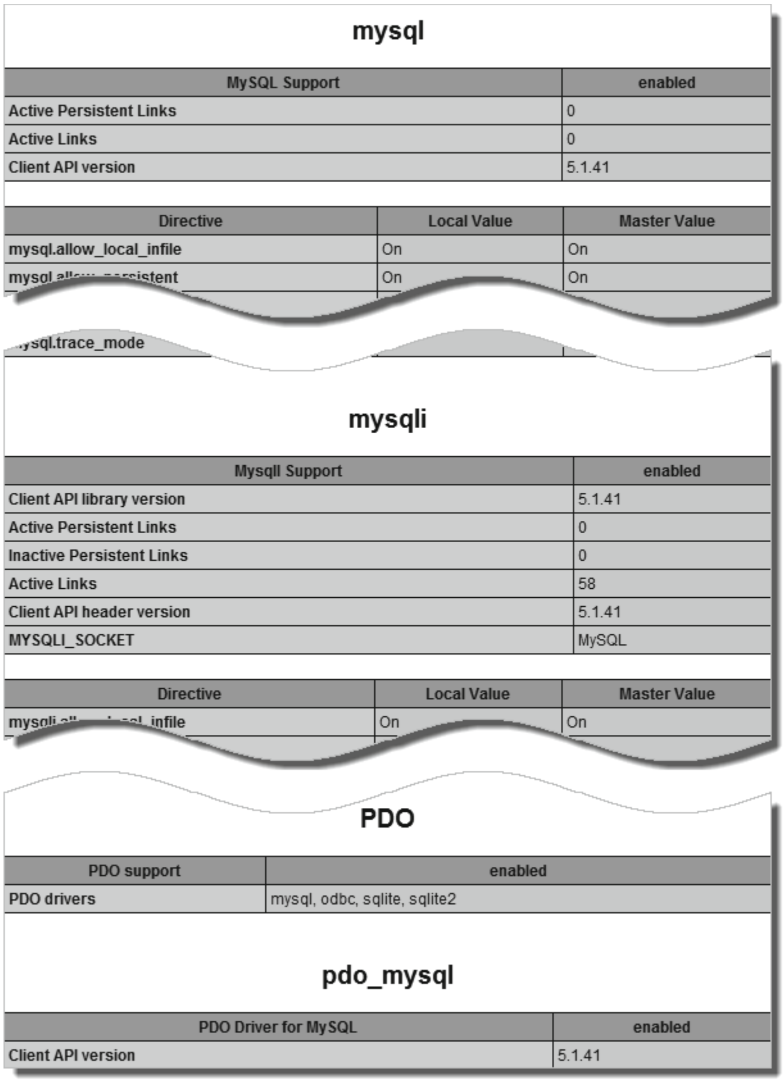
Although PHP connects to the database and stores any results, the database queries need to be written in SQL. This chapter teaches you the basics of retrieving information stored in a table.

In this chapter, you'll learn the following:

- Connecting to MySQL with MySQLi and PDO
- Counting the number of records in a table
- Using `SELECT` queries to retrieve data and display it in a web page
- Keeping data secure with prepared statements and other techniques

Checking your remote server setup

XAMPP and MAMP support all three methods of communicating with MySQL, but you need to check the PHP configuration of your remote server to verify the degree of support it offers. Run `phpinfo()` on your remote server, scroll down the configuration page, and look for the following sections. They're listed alphabetically, so you'll need to scroll down a long way to find them.



mysql

MySQL Support	enabled
Active Persistent Links	0
Active Links	0
Client API version	5.1.41

Directive	Local Value	Master Value
mysql.allow_local_infile	On	On
mysql.allow_persistent	On	On
ysql.trace_mode		

mysqli

MysqlI Support	enabled
Client API library version	5.1.41
Active Persistent Links	0
Inactive Persistent Links	0
Active Links	58
Client API header version	5.1.41
MYSQLI_SOCKET	MySQL

Directive	Local Value	Master Value
mysqli.allow_local_infile	On	On

PDO

PDO support	enabled
PDO drivers	mysql, odbc, sqlite, sqlite2

pdo_mysql

PDO Driver for MySQL	enabled
Client API version	5.1.41

All hosting companies should have the first two sections (**mysql** and **mysqli**). If you plan to use PDO, you not only need to check that PDO is enabled, but you must also make sure **mysql** is listed among the **PDO drivers**.

How PHP communicates with MySQL

Regardless of whether you use MySQLi or PDO, the process always follows this sequence:

1. Connect to MySQL using the hostname, username, password, and database name.
2. Prepare a SQL query.
3. Execute the query and save the result.
4. Extract the data from the result (usually with a loop).

Username and password are straightforward: they're the username and password of the accounts you have just created or the account given to you by your hosting company. But what about hostname? In a local testing environment it's `localhost`. What comes as a surprise is that MySQL often uses `localhost` even on a remote server. This is because in many cases the database server is located on the same server as your website. In other words, the web server that displays your pages and the MySQL server are local to each other. However, if your hosting company has installed MySQL on a separate machine, it will tell you the address to use. The important thing to realize is that the MySQL hostname is not the same as your website domain name.

Let's take a quick look at how you connect to a MySQL server with each of the methods.

Connecting with the MySQL Improved extension

MySQLi has two interfaces: procedural and object-oriented. The procedural interface is designed to ease the transition from the original MySQL functions. Since the object-oriented version is more compact, that's the version adopted here.

To connect to a MySQL server, you create a `mysqli` object by passing four arguments to `new mysqli()`: the hostname, username, password, and the name of the database. This is how you connect to the `phpsols` database:

```
$conn = new mysqli($hostname, $username, $password, 'phpsols');
```

This stores the connection object as `$conn`.

Connecting with PDO

PDO requires a slightly different approach. The most important difference is that, if you're not careful, PDO displays your database username and password onscreen when it can't connect to the database. This is because PDO throws an exception if the connection fails. So, you need to wrap the code in a `try` block, and catch the exception.

To create a connection to the MySQL server, you create a data object by passing the following three arguments to `new PDO()`:

- A string specifying the database type, the hostname, and the name of the database. The string must be presented in the following format:

  `'mysql:host=`*`hostname`*`;dbname=`*`databaseName`*`'`

- The username.

- The user's password.

The code looks like this:

```
try {
  $conn = new PDO("mysql:host=$hostname;dbname=phpsols", $username, $password);
} catch (PDOException $e) {
  echo $e->getMessage();
}
```

Using `echo` to display the message generated by the exception is OK during testing, but when you deploy the script on a live website, you need to redirect the user to an error page, as described in PHP Solution 4-8.

PHP Solution 11-1: Making a reusable database connector

Connecting to a database is a routine chore that needs to be performed in every page from now on. This PHP solution creates a simple function stored in an external file that connects to the database. It's designed mainly for testing the different MySQLi and PDO scripts in the remaining chapters without the need to retype the connection details each time or to switch between different connection files.

1. Create a file called `connection.inc.php` in the `includes` folder, and insert the following code (there's a copy of the completed script in the `ch11` folder):

```
<?php
function dbConnect($usertype, $connectionType = 'mysqli') {
  $host = 'localhost';
  $db = 'phpsols';
  if ($usertype  == 'read') {
    $user = 'psread';
    $pwd = 'K1yOmi$u';
  } elseif ($usertype == 'write') {
    $user = 'pswrite';
    $pwd = '0Ch@Nom1$u';
  } else {
    exit('Unrecognized connection type');
  }
  // Connection code goes here
}
```

   The function takes two arguments: the user type and the type of connection you want. The second argument defaults to `mysqli`. If you want to concentrate on using PDO, set the default value of the second argument to `pdo`.

   The first two lines inside the function store the name of the host server and database that you want to connect to.

   The `if... elseif` conditional statement checks the value of the first argument and switches between the `psread` and `pswrite` username and password as appropriate.

2. Replace the `Connection code goes here` comment with the following:

```
if ($connectionType == 'mysqli') {
  return new mysqli($host, $user, $pwd, $db) or die ('Cannot open database');
} else {
  try {
    return new PDO("mysql:host=$host;dbname=$db", $user, $pwd);
  } catch (PDOException $e) {
    echo 'Cannot connect to database';
    exit;
  }
}
```

If the second argument is set to `mysqli`, a MySQLi connection object is returned. Otherwise, the function returns a PDO connection. The rather foreboding `die()` simply stops the script from attempting to continue and displays the error message.

To create a MySQLi connection to the `phpsols` database, include `connection.inc.php`, and call the function like this for the `psread` user:

```
$conn = dbConnect('read');
```

For the `pswrite` user, call it like this:

```
$conn = dbConnect('write');
```

To create a PDO connection, add the second argument like this:

```
$conn = dbConnect('read', 'pdo');
$conn = dbConnect('write', 'pdo');
```

Finding the number of results from a query

Counting the number of results from a database query is useful in several ways. It's necessary for creating a navigation system to page through a long set of results (you'll learn how to do that in the next chapter). It's also important for user authentication (covered in Chapter 17). If you get no results from matching a username and password, you know that the login procedure should fail.

MySQLi has a convenient method of finding out the number of results returned by a query. However, PDO doesn't have a direct equivalent.

PHP Solution 11-2: Counting records in a result set (MySQLi)

This PHP solution shows how to submit a SQL query to select all the records in the `images` table, and store the result in a `MySQLi_Result` object. The object's `num_rows` property contains the number of records retrieved by the query.

1. Create a new folder called `mysql` in the `phpsols` site root, and create a new file called `mysqli.php` inside the folder. The page will eventually be used to display a table, so it should have a `DOCTYPE` declaration and an HTML skeleton.

2. Include the connection file in a PHP block above the `DOCTYPE` declaration, and create a connection to MySQL using the account that has read-only privileges like this:

```
require_once('../includes/connection.inc.php');
```

```
// connect to MySQL
$conn = dbConnect('read');
```

3. Next, prepare the SQL query. Add this code immediately after the previous step (but before the closing PHP tag):

```
// prepare the SQL query
$sql = 'SELECT * FROM images';
```

   This means “select everything from the `images` table.” The asterisk (*) is shorthand for “all columns.”

4. Now execute the query by calling the `query()` method on the connection object and passing the SQL query as an argument like this:

```
// submit the query and capture the result
$result = $conn->query($sql) or die(mysqli_error());
```

   The result is stored in a variable, which I have imaginatively named `$result`. If there is a problem, the database server returns an error message, which can be retrieved using `mysqli_error()`. By placing this function between the parentheses of `die()`, the script comes to a halt if there’s a problem and displays the error message.

5. Assuming there’s no problem, `$result` now holds a `MySQLi_Result` object. To get the number of records found by the SQL query, assign the value to a variable like this:

```
// find out how many records were retrieved
$numRows = $result->num_rows;
```

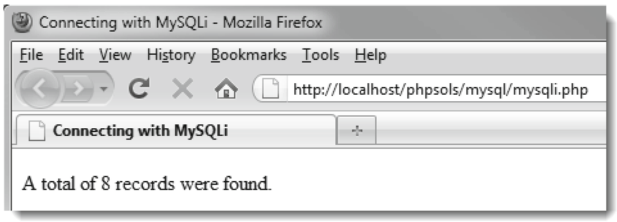
   The complete code above the `DOCTYPE` declaration looks like this:

```
require_once('../includes/connection.inc.php');
// connect to MySQL
$conn = dbConnect('read');
// prepare the SQL query
$sql = 'SELECT * FROM images';
// submit the query and capture the result
$result = $conn->query($sql) or die(mysqli_error());
// find out how many records were retrieved
$numRows = $result->num_rows;
```

6. You can now display the value of `$numRows` in the body of the page like this:

```
<p>A total of <?php echo $numRows; ?> records were found.</p>
```

7. Save `mysqli.php` and load it into a browser. You should see the following result:


Connecting with MySQLi - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/phpsols/mysql/mysqli.php
Connecting with MySQLi

A total of 8 records were found.

Check your code, if necessary, with `mysqli_01.php` in the `ch11` folder.

PHP Solution 11-3: Counting records in a result set (PDO)

PDO doesn't have an equivalent of the MySQLi `num_rows` property. With most databases, you need to execute a SQL query to count the number of items in the table, and then fetch the result. However, you're in luck, because the PDO `rowCount()` method fulfils a dual purpose with MySQL. Normally, the `rowCount()` method reports only the number of rows affected by inserting, updating, or deleting records; but with MySQL, it also reports the number of records found by a `SELECT` query.

1. Create a new file called `pdo.php` in the `mysql` folder. The page will eventually be used to display a table, so it should have a `DOCTYPE` declaration and an HTML skeleton.

2. Include the connection file in a PHP block above the `DOCTYPE` declaration, and create a PDO connection to MySQL using the read-only account like this:

```
require_once('../includes/connection.inc.php');
// connect to MySQL
$conn = dbConnect('read', 'pdo');
```

3. Next, prepare the SQL query:

```
// prepare the SQL query
$sql = 'SELECT * FROM images';
```

   This means "select every record in the `images` table." The asterisk (*) is shorthand for "all columns."

4. Now execute the query and store the result in a variable like this:

```
// submit the query and capture the result
$result = $conn->query($sql);
$error = $conn->errorInfo();
if (isset($error[2])) die($error[2]);
```

   PDO uses `errorInfo()` to build an array of error messages from the database. The third element of the array is created only if something goes wrong. I've stored the result of `$conn->errorInfo()` as `$error`, so you can tell if anything went wrong by using `isset()` to check whether `$error[2]` has been defined. If it has, `die()` brings the script to a halt and displays the error message.

5. To get the number of rows in the result set, call the `rowCount()` method on the `$result` object. The finished code in the PHP block above the `DOCTYPE` declaration looks like this:

```
require_once('../includes/connection.inc.php');
// connect to MySQL
$conn = dbConnect('read', 'pdo');
// prepare the SQL query
$sql = 'SELECT * FROM images';
// submit the query and capture the result
$result = $conn->query($sql);
$error = $conn->errorInfo();
if (isset($error[2])) die($error[2]);
// find out how many records were retrieved
$numRows = $result->rowCount();
```

6. You can now display the value of `$numRows` in the body of the page like this:

   ```
   <p>A total of <?php echo $numRows; ?> records were found.</p>
   ```

7. Save the page, and load it into a browser. You should see the same result as shown in step 5 of PHP Solution 11-2. Check your code, if necessary, with `pdo_01.php`.

In my tests, using `rowCount()` reported the number of items found by a `SELECT` query in MySQL on both Mac OS X and Windows. However, it cannot be guaranteed to work on all databases. If `rowCount()` doesn't work, use the following code instead:

```
// prepare the SQL query
$sql = 'SELECT COUNT(*) FROM images';
// submit the query and capture the result
$result = $conn->query($sql);
$error = $conn->errorInfo();
if (isset($error[2])) die($error[2]);
// find out how many records were retrieved
$numRows = $result->fetchColumn();
// free the database resource
$result->closeCursor();
```

This uses the SQL `COUNT()` function with an asterisk to count all items in the table. There's only one result, so it can be retrieved with the `fetchColumn()` method, which gets the first column from a database result. After storing the result in `$numRows`, you need to call the `closeCursor()` method to free the database resource for any further queries.

Displaying the results of a query

The most common way to display the results of a query is to use a loop in combination with the MySQLi or PDO method to extract the current record into a temporary array.

With MySQLi, use the `fetch_assoc()` method like this:

```
while ($row = $result->fetch_assoc()) {
  // do something with the current record
}
```

PDO handles it slightly differently. You can use the `query()` method directly inside a `foreach` loop to create an array for each record like this:

```
foreach ($conn->query($sql) as $row) {
  // do something with the current record
}
```

In the case of the `images` table, `$row` contains `$row['image_id']`, `$row['filename']`, and `$row['caption']`. Each element is named after the corresponding column in the table.

PHP Solution 11-4: Displaying the images table using MySQLi

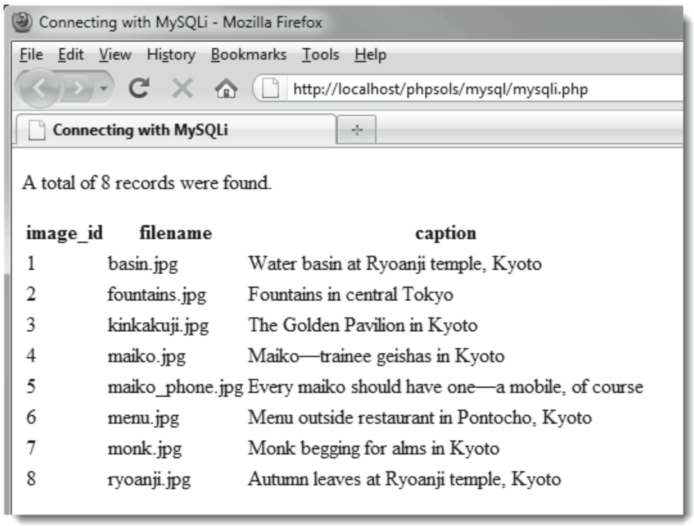
This PHP solution shows how to loop through a `MySQLi_Result` object to display the results of a `SELECT` query. Continue using the file from PHP Solution 11-2.

1. Add the following table to the main body of `mysqli.php` (the PHP code that displays the result set is highlighted in bold):

   ```
   <table>
     <tr>
       <th>image_id</th>
       <th>filename</th>
       <th>caption</th>
     </tr>
     <?php while ($row = $result->fetch_assoc()) { ?>
     <tr>
       <td><?php echo $row['image_id']; ?></td>
       <td><?php echo $row['filename']; ?></td>
       <td><?php echo $row['caption']; ?></td>
     </tr>
     <?php } ?>
   </table>
   ```

   The `while` loop iterates through the database result, using the `fetch_assoc()` method to extract each record into `$row`. Each element of `$row` is displayed in a table cell. The loop continues until `fetch_assoc()` comes to the end of the result set.

2. Save `mysqli.php`, and view it in a browser. You should see the contents of the `images` table displayed as shown in the following screenshot:


Connecting with MySQLi - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/phpsols/mysql/mysqli.php
Connecting with MySQLi

A total of 8 records were found.

image_id	filename	caption
1	basin.jpg	Water basin at Ryoanji temple, Kyoto
2	fountains.jpg	Fountains in central Tokyo
3	kinkakuji.jpg	The Golden Pavilion in Kyoto
4	maiko.jpg	Maiko—trainee geishas in Kyoto
5	maiko_phone.jpg	Every maiko should have one—a mobile, of course
6	menu.jpg	Menu outside restaurant in Pontocho, Kyoto
7	monk.jpg	Monk begging for alms in Kyoto
8	ryoanji.jpg	Autumn leaves at Ryoanji temple, Kyoto

You can compare your code, if necessary with `mysql_02.php` in the `ch11` folder.

PHP Solution 11-5: Displaying the images table using PDO

Instead of a `while` loop with `fetch_assoc()`, PDO uses the `query()` method in a `foreach` loop.

Continue working with `pdo.php`, the same file as in PHP Solution 11-3.

1. Insert the following table in the body of `pdo.php` (the PHP code that displays the result set is displayed in bold):

```
<table>
  <tr>
    <th>image_id</th>
    <th>filename</th>
    <th>caption</th>
  </tr>
  <?php foreach ($conn->query($getDetails) as $row) { ?>
  <tr>
    <td><?php echo $row['image_id']; ?></td>
    <td><?php echo $row['filename']; ?></td>
    <td><?php echo $row['caption']; ?></td>
  </tr>
  <?php } ?>
</table>
```

2. Save the page, and view it in a browser. It should look like the screenshot in PHP Solution 11-4. You can compare your code against `pdo_02.php` in the `ch11` folder.

MySQL connection crib sheet

Tables 11-2 and 11-3 summarize the basic details of connection and database query for MySQLi and PDO. Some commands will be used in later chapters, but are included here for ease of reference.

Table 11-2. Connection to MySQL with the MySQL Improved object-oriented interface

Action	Usage	Comments
Connect	`$conn = new mysqli($h,$u,$p,$d);`	All arguments optional; first four always needed in practice: hostname, username, password, database name. Creates connection object.
Choose DB	`$conn->select_db('dbName');`	Use to select a different database.
Submit query	`$result = $conn->query($sql);`	Returns result object.
Count results	`$numRows = $result->num_rows;`	Returns number of rows in result object.
Release DB resources	`$result->free_result();`	Frees up connection to allow new query.
Extract record	`$row = $result->fetch_assoc();`	Extracts current row from result object as associative array.
Extract record	`$row = $result->fetch_row();`	Extracts current row from result object as indexed (numbered) array.

Table 11-3. Connection to MySQL with PDO

Action	Usage	Comments
Connect	`$conn = new PDO($DSN,$u,$p);`	In practice, requires three arguments: data source name (DSN), username, password. Must be wrapped in try/catch block.
Submit query	`$result = $conn->query($sql);`	Can also be used inside `foreach` loop to extract each record.
Count results	`$numRows = $result->rowCount()`	Should work with MySQL, but use `SELECT COUNT(*) FROM` *`table_name`* for other databases.

Action	Usage	Comments
Get single result	`$item = $result->fetchColumn();`	Gets first column in first record of result. To get result from other columns, use column number (from 0) as argument.
Get next record	`$row = $result->fetch();`	Gets next row from result set as associative array.
Release DB resources	`$result->closeCursor();`	Frees up connection to allow new query.
Extract records	`foreach($conn->query($sql) as $row) {`	Extracts current row from result set as associative array.

When using PDO with MySQL, the data source name (DSN) is a string that takes the following format:

```
'mysql:host=hostname;dbname=databaseName'
```

If you need to specify a different port from the MySQL default (3306), use the following format, substituting the actual port number:

```
'mysql:host=hostname;port=3307;dbname=databaseName'
```

Using SQL to interact with a database

As you have just seen, PHP connects to the database, sends the query, and receives the results; but the query itself needs to be written in SQL. Although SQL is a common standard, there are many dialects of SQL. Each database vendor, including MySQL, has added extensions to the standard language. These improve efficiency and functionality, but are usually incompatible with other databases. The SQL in this book works with MySQL 4.1 or later, but it won't necessarily transfer to Microsoft SQL Server, Oracle, or another database.

Writing SQL queries

SQL syntax doesn't have many rules, and all of them are quite simple.

SQL is case-insensitive

The query that retrieves all records from the `images` table looks like this:

```
SELECT * FROM images
```

The words in uppercase are SQL keywords. This is purely a convention. The following are all equally correct:

```
SELECT * FROM images
select * from images
```

```
SeLEcT * fRoM images
```

Although SQL keywords are case-insensitive, the same *doesn't* apply to database column names. The advantage of using uppercase for keywords is that it makes SQL queries easier to read. You're free to choose whichever style suits you best, but the ransom-note style of the last example is probably best avoided.

Whitespace is ignored

This allows you to spread SQL queries over several lines for increased readability. The one place where whitespace is *not* allowed is between a function name and the opening parenthesis. The following generates an error:

```
SELECT COUNT (*) FROM images  /* BAD EXAMPLE */
```

The space needs to be closed up like this:

```
SELECT COUNT(*) FROM images  /* CORRECT */
```

As you probably gathered from these examples, you can add comments to SQL queries by putting them between `/*` and `*/`.

Strings must be quoted

All strings must be quoted in a SQL query. It doesn't matter whether you use single or double quotes, as long as they are in matching pairs. However, it's normally better to use MySQLi or PDO prepared statements, as explained later in this chapter.

Handling numbers

As a general rule, numbers should not be quoted, as anything in quotes is a string. However, MySQL accepts numbers enclosed in quotes and treats them as their numeric equivalent. Be careful to distinguish between a real number and any other data type made up of numbers. For instance, a date is made up of numbers but should be enclosed in quotes and stored in a date-related column type. Similarly, telephone numbers should be enclosed in quotes and stored in a text-related column type.

SQL queries normally end with a semicolon, which is an instruction to the database to execute the query. When using PHP, the semicolon must be omitted from the SQL. Consequently, standalone examples of SQL are presented throughout this book without a concluding semicolon.

Refining the data retrieved by a SELECT query

The only SQL query you have run so far retrieves all records from the `images` table. Much of the time, you want to be more selective.

Selecting specific columns

Using an asterisk to select all columns is a convenient shortcut, but you should normally specify only those columns you need. List the column names separated by commas after the `SELECT` keyword. For example, this query selects only the `filename` and `caption` fields for each record:

```
SELECT filename, caption FROM images
```

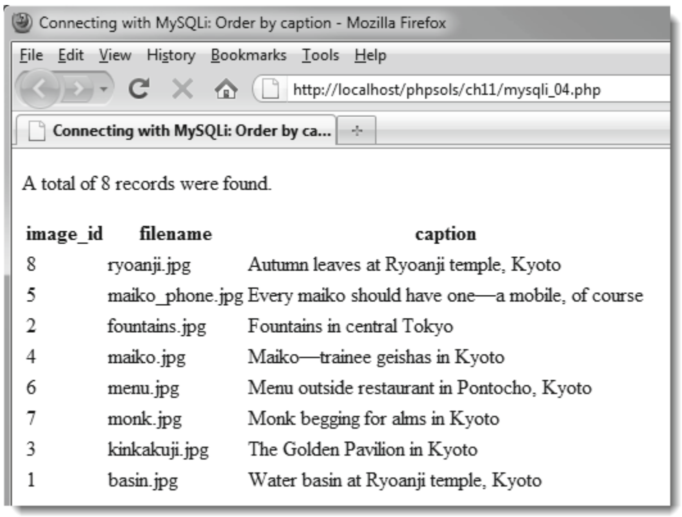
You can test this in `mysqli_03.php` and `pdo_03.php` in the `ch11` folder.

Changing the order of results

To control the sort order, add an `ORDER BY` clause with the name(s) of the column(s) in order of precedence. Separate multiple columns by commas. The following query sorts the captions from the `images` table in alphabetical order (the code is in `mysqli_04.php` and `pdo_04.php`):

```
$sql = 'SELECT * FROM images ORDER BY caption';
```

The semicolon indicates the end of the PHP statement. It is not part of the SQL query.

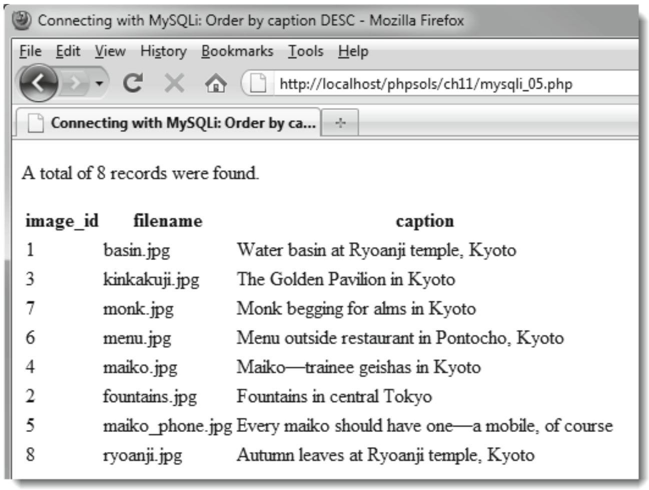

Connecting with MySQLi: Order by caption - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/phpsols/ch11/mysqli_04.php
Connecting with MySQLi: Order by ca...

A total of 8 records were found.

image_id	filename	caption
8	ryoanji.jpg	Autumn leaves at Ryoanji temple, Kyoto
5	maiko_phone.jpg	Every maiko should have one—a mobile, of course
2	fountains.jpg	Fountains in central Tokyo
4	maiko.jpg	Maiko—trainee geishas in Kyoto
6	menu.jpg	Menu outside restaurant in Pontocho, Kyoto
7	monk.jpg	Monk begging for alms in Kyoto
3	kinkakuji.jpg	The Golden Pavilion in Kyoto
1	basin.jpg	Water basin at Ryoanji temple, Kyoto

To reverse the sort order, add the `DESC` (for “descending”) keyword like this:

```
$sql = 'SELECT * FROM images ORDER BY caption DESC';
```


Connecting with MySQLi: Order by caption DESC - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/phpsols/ch11/mysqli_05.php
Connecting with MySQLi: Order by ca...

A total of 8 records were found.

image_id	filename	caption
1	basin.jpg	Water basin at Ryoanji temple, Kyoto
3	kinkakuji.jpg	The Golden Pavilion in Kyoto
7	monk.jpg	Monk begging for alms in Kyoto
6	menu.jpg	Menu outside restaurant in Pontocho, Kyoto
4	maiko.jpg	Maiko—trainee geishas in Kyoto
2	fountains.jpg	Fountains in central Tokyo
5	maiko_phone.jpg	Every maiko should have one—a mobile, of course
8	ryoanji.jpg	Autumn leaves at Ryoanji temple, Kyoto

There is also an ASC (for “ascending”) keyword. It’s the default sort order, so is normally omitted.

However, specifying ASC increases clarity when columns in the same table are sorted in a different order. For example, if you publish multiple articles every day, you could use the following query to display titles in alphabetical order, but ordered by the date of publication with the most recent ones first:

```
SELECT * FROM articles
ORDER BY published DESC, title ASC
```

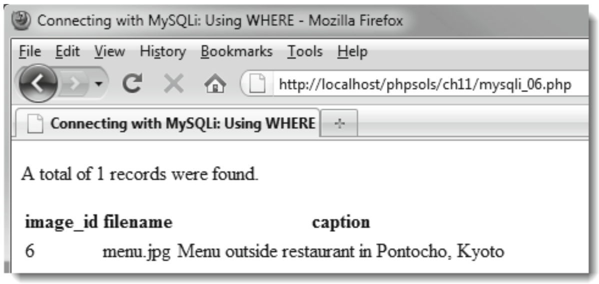
Searching for specific values

To search for specific values, add a WHERE clause to the SELECT query. The WHERE clause follows the name of the table. For example, the query in mysqli_06.php and pdo_06.php looks like this:

```
$sql = 'SELECT * FROM images
        WHERE image_id = 6';
```

Note that SQL uses a single equal sign to test for equality, unlike PHP, which uses two.

It produces the following result:


Connecting with MySQLi: Using WHERE - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/phpsols/ch11/mysqli_06.php
Connecting with MySQLi: Using WHERE

A total of 1 records were found.

image_id	filename	caption
6	menu.jpg	Menu outside restaurant in Pontocho, Kyoto

In addition to testing for equality, a `WHERE` clause can use comparison operators, such as greater than (>) and less than (<). Rather than go through all the options now, I'll introduce others as needed. Chapter 13 has a comprehensive roundup of the four main SQL commands: `SELECT`, `INSERT`, `UPDATE`, and `DELETE`, including a list of the main comparison operators used with `WHERE`.

If used in combination with `ORDER BY`, the `WHERE` clause must come first. For example (the code is in `mysqli_07.php` and `pdo_07.php`):

```
$sql = 'SELECT * FROM images
        WHERE image_id > 6
        ORDER BY caption DESC';
```

This selects the two images that have an `image_id` greater than 6 and sorts them by their captions in reverse order.

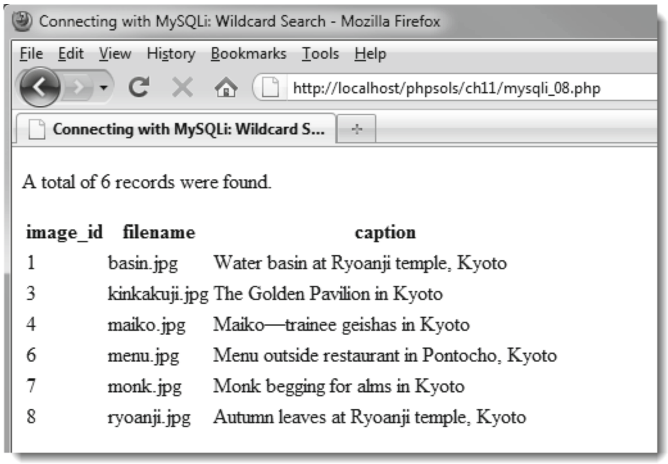
Searching for text with wildcard characters

In SQL, the percentage sign (%) is a wildcard character that matches anything or nothing. It's used in a `WHERE` clause in conjunction with the `LIKE` keyword.

The query in `mysqli_08.php` and `pdo_08.php` looks like this:

```
$sql = 'SELECT * FROM images
        WHERE caption LIKE "%Kyoto%"';
```

It searches for all records in the `images` table where the `caption` column contains "Kyoto," and produces the following result:


Connecting with MySQLi: Wildcard Search - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/phpsols/ch11/mysqli_08.php
Connecting with MySQLi: Wildcard S...

A total of 6 records were found.

image_id	filename	caption
1	basin.jpg	Water basin at Ryoanji temple, Kyoto
3	kinkakuji.jpg	The Golden Pavilion in Kyoto
4	maiko.jpg	Maiko—trainee geishas in Kyoto
6	menu.jpg	Menu outside restaurant in Pontocho, Kyoto
7	monk.jpg	Monk begging for alms in Kyoto
8	ryoanji.jpg	Autumn leaves at Ryoanji temple, Kyoto

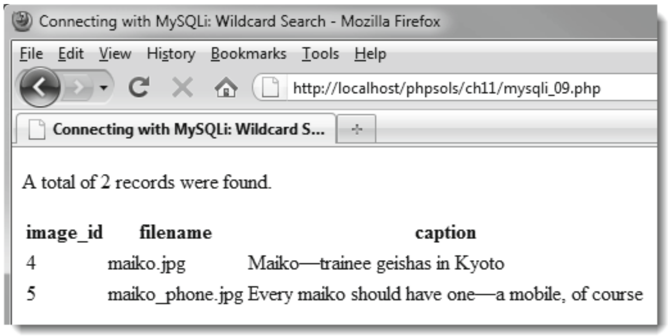
As the preceding screenshot shows, it finds six records out of the eight in the `images` table. All the captions end with "Kyoto," so the wildcard character at the end is matching nothing, whereas the wildcard at the beginning matches the rest of each caption.

If you omit the leading wildcard (`"Kyoto%"`), the query searches for captions that begin with "Kyoto." None of them does, so you get no results from the search.

The query in `mysqli_09.php` and `pdo_09.php` looks like this:

```
$sql = 'SELECT * FROM images
        WHERE caption LIKE "%maiko%"';
```

It produces the following result:


Connecting with MySQLi: Wildcard Search - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/phpsols/ch11/mysqli_09.php
Connecting with MySQLi: Wildcard S...

A total of 2 records were found.

image_id	filename	caption
4	maiko.jpg	Maiko—trainee geishas in Kyoto
5	maiko_phone.jpg	Every maiko should have one—a mobile, of course

The query spells "maiko" all in lowercase, but the query also finds it with an initial capital. Wildcard searches with `LIKE` are case-insensitive.

To perform a case-sensitive search, you need to add the `BINARY` keyword like this (the code is in `mysqli_10.php` and `pdo_10.php`):

```
$sql = 'SELECT * FROM images
        WHERE caption LIKE BINARY "%maiko%"';
```

All the examples you have seen so far have been hard-coded, but most of the time, the values used in SQL queries need to come from user input. Unless you're careful, this puts you at risk from a malicious exploit know as SQL injection. The rest of this chapter explains the danger and how to avoid it.

Understanding the danger of SQL injection

SQL injection is very similar to the email header injection I warned you about in Chapter 5. An injection attack tries to insert spurious conditions into a SQL query in an attempt to expose or corrupt your data. The meaning of the following query should be easy to understand:

```
SELECT * FROM users WHERE username = 'xyz' AND pwd = 'abc'
```

It's the basic pattern for a login application. If the query finds a record where `username` is `xyz` and `pwd` is `abc`, you know that a correct combination of username and password have been submitted, so the login succeeds. All an attacker needs to do is inject an extra condition like this:

```
SELECT * FROM users WHERE username = 'xyz' AND pwd = 'abc' OR 1 = 1
```

The `OR` means only one of the conditions needs to be true, so the login succeeds even without a correct username and password. SQL injection relies on quotes and other control characters not being properly escaped when part of the query is derived from a variable or user input.

There are several strategies you can adopt to prevent SQL injection, depending on the situation:

- If the variable is an integer (for example, the primary key of a record), use `is_numeric()` and the `(int)` casting operator to ensure it's safe to insert in the query.

- If you are using MySQLi, pass each variable to the `real_escape_string()` method before inserting it in the query.
- The PDO equivalent of `real_escape_string()` is the `quote()` method, but it doesn't work with all databases. The PDO documentation advises against using `quote()`, strongly recommending the use of prepared statements instead.
- Use a **prepared statement**. In a prepared statement, placeholders in the SQL query represent values that come from user input. The PHP code automatically wraps strings in quotes, and escapes embedded quotes and other control characters. The syntax is different for MySQLi and PDO.
- None of the preceding strategies is suitable for column names, which must not be enclosed in quotes. To use a variable for column names, create an array of acceptable values, and check that the submitted value is in the array before inserting it into the query.

With the exception of `quote()`, let's take a look at using each of these techniques.

PHP Solution 11-6: Inserting an integer from user input into a query

This PHP solution shows how to sanitize a variable from user input to make sure it contains only an integer before inserting the value in a SQL query. The technique is the same for both MySQLi and PDO.

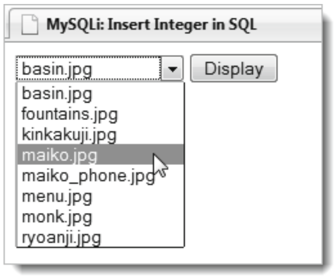
1. Copy either `mysqli_integer_01.php` or `pdo_integer_01.php` from the `ch11` folder to the `mysql` folder. Each file contains a SQL query that selects the `image_id` and `filename` columns from the `images` table. In the body of the page, there's a form with a drop-down menu which is populated by a loop that runs through the results of the SQL query. The MySQLi version looks like this:

```
<form action="" method="get" id="form1">
  <select name="image_id" id="image_id">
    <?php while ($row = $images->fetch_assoc()) { ?>
    <option value="<?php echo $row['image_id']; ?>"
    <?php if (isset($_GET['image_id']) && $_GET['image_id'] == ➥
      $row['image_id']) {
      echo 'selected';
    } ?>
    ><?php echo $row['filename']; ?></option>
    <?php } ?>
  </select>
  <input type="submit" name="go" id="go" value="Display">
</form>
```

The form uses the `get` method and assigns the `image_id` to the `value` attribute of the `<option>` tags. If `$_GET['image_id']` has the same value as `$row['image_id']`, the current `image_id` is the same as passed through the page's query string, so the `selected` attribute is added to the opening `<option>` tag. The value of `$row['filename']` is inserted between the opening and closing `<option>` tags.

The PDO version is identical apart from the fact that it runs the query directly in a `foreach` loop.

2. If you load the page into a browser, you'll see a drop-down menu that lists the files in the `images` folder like this:



3. Insert the following code immediately after the closing `</form>` tag. The code is the same for both MySQLi and PDO, apart from one line.

```
<?php
if (isset($_GET['image_id'])) {
  if (!is_numeric($_GET['image_id'])) {
    $image_id = 1;
  } else {
    $image_id = (int) $_GET['image_id'];
  }
  $sql = "SELECT filename, caption FROM images
         WHERE image_id = $image_id";
  $result = $conn->query($sql);
  $row = $result->fetch_assoc();
?>
<figure><img src="../images/<?php echo $row['filename']; ?>">
  <figcaption><?php echo $row['caption']; ?></figcaption>
</figure>
<?php } ?>
```

The conditional statement checks whether `image_id` has been sent through the `$_GET` array. If it has, the next conditional statement uses the logical Not operator with `is_numeric()` to check whether it's not numeric. The `is_numeric()` function applies a strict test, accepting only numbers or numeric strings. It doesn't attempt to convert the value to a number if it begins with a digit.

If the value submitted through the query string isn't numeric, a default value is assigned to a new variable called `$image_id`. However, if `$_GET['image_id']` is numeric, it's assigned to `$image_id` using the `(int)` casting operator. Using the casting operator is an extra precaution in case someone tries to probe your script for error messages by submitting a floating point number.

Since you know `$image_id` is an integer, it's safe to insert directly in the SQL query. Because it's a number, it doesn't need to be wrapped in quotes, but the string assigned to `$sql` needs to use double quotes to ensure the value of `$image_id` is inserted into the query.

The new query is submitted to MySQL by the `query()` method, and the result is stored in `$row`. Finally, `$row['filename']` and `$row['caption']` are used to display the image and its caption in the page.

4. If you are using the PDO version, locate this line:

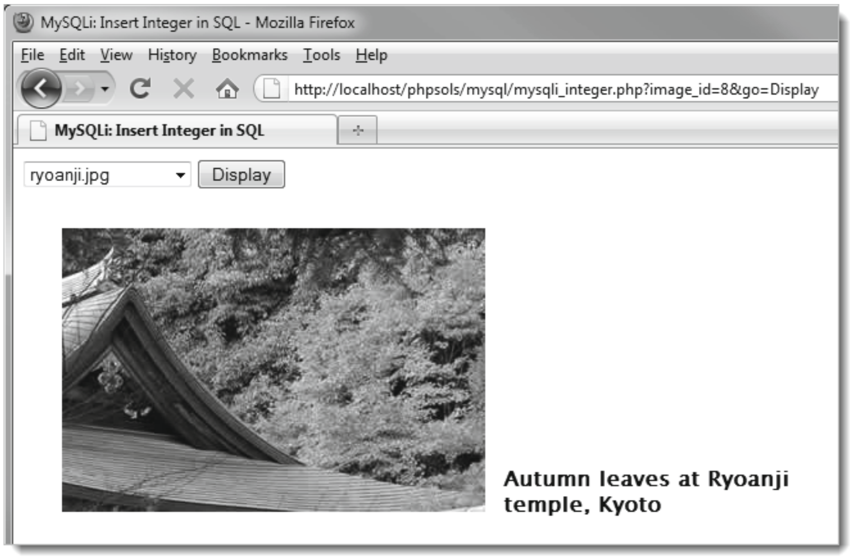
```
$row = $result->fetch_assoc();
```

Change it to this:

```
$row = $result->fetch();
```

5. Save the page, and load it into a browser. When the page first loads, only the drop-down menu is displayed.

6. Select a filename from the drop-down menu, and click **Display**. The image of your choice should be displayed, as shown in the following screenshot:


MySQLi: Insert Integer in SQL - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/phpsols/mysql/mysqli_integer.php?image_id=8&go=Display
MySQLi: Insert Integer in SQL
ryoanji.jpg
Display

Autumn leaves at Ryoanji temple, Kyoto

7. If you encounter problems, check your code against `mysqli_integer_02.php` or `pdo_integer_02.php` in the `ch11` folder.

8. Edit the query string in the browser, changing the value of `image_id` to a string or a string that begins with a number. You should see `basin.jpg`, which has `image_id` 1.

9. Try a floating point number between 1.0 and 8.9. The relevant image is displayed normally.

10. Try a number outside the range of 1–8. No error messages are displayed because there's nothing wrong with the query. It's simply looking for a value that doesn't exist. In this example, it doesn't matter, but you should normally check the number of rows returned by the query, using the `num_rows` property with MySQLi or the `rowCount()` method with PDO.

11. Change the code like this for MySQLi:

```
  $result = $conn->query($sql);
  if ($result->num_rows) {
    $row = $result->fetch_assoc();
  ?>
<figure><img src="../images/<?php echo $row['filename']; ?>">
  <figcaption><?php echo $row['caption']; ?></figcaption>
</figure>
<?php } else { ?>
<p>Image not found</p>
<?php }
}?>
```

For PDO, use `$result->rowCount()` in place of `$result->num_rows`.

If no rows are returned by the query, 0 is treated by PHP as implicitly `false`, so the condition fails, and the `else` clause is executed instead.

12. Test the page again. When you select an image from the drop-down menu, it displays normally as before. But if you try entering an out-of-range value in the query string, you see the following message instead:

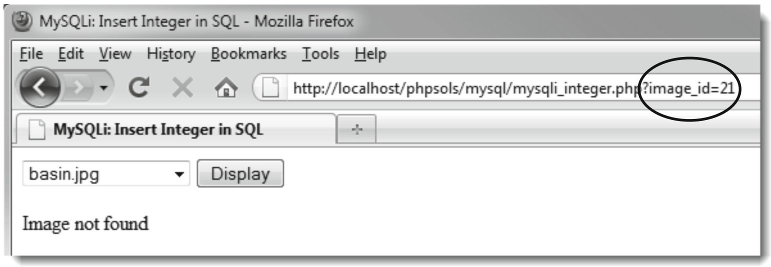

MySQLi: Insert Integer in SQL - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/phpsols/mysql/mysqli_integer.php?image_id=21
MySQLi: Insert Integer in SQL
basin.jpg
Display

Image not found

The amended code is in `mysqli_integer_03.php` and `pdo_integer_03.php` in the `ch11` folder.

PHP Solution 11-7: Inserting a string with real_escape_string()

This PHP solution works only with MySQLi. It shows how to insert a value from a search form into a SQL query using the `real_escape_string()` method. If you have used the original MySQL extension before, it does the same as the `mysql_real_escape_string()` function. In addition to handling single and double quotes, it also escapes other control characters, such as newlines and carriage returns. Although the functionality is the same, you must use the MySQLi version. You can't use `mysql_real_escape_string()` with MySQLi.

1. Copy `mysqli_real_escape_01.php` from the `ch11` folder, and save it in the `mysql` folder as `mysql_real_escape.php`. The file contains a search form and a table for displaying the results.

2. Add the following code in a PHP block above the `DOCTYPE` declaration:

```
if (isset($_GET['go'])) {
  require_once('../includes/connection.inc.php');
```

```
    $conn = dbConnect('read');
    $searchterm = '%' . $conn->real_escape_string($_GET['search']) . '%';
  }
```

3. This includes the connection file and establishes a MySQLi connection for the read-only user account if the form has been submitted. Then, the value of `$_GET['search']` is passed to the connection object's `real_escape_string()` method to make it safe to incorporate into a SQL query, and the `%` wildcard character is concatenated to both ends before the result is assigned to `$searchterm`. So, if the value submitted through the search form is "hello," `$searchterm` becomes `%hello%`.

4. Add the `SELECT` query on the next line (before the closing curly brace):

```
$sql = "SELECT * FROM images WHERE caption LIKE '$searchterm'";
```

The whole query is wrapped in double quotes so that the value of `$searchterm` is incorporated. However, `$searchterm` contains a string, so it also needs to be wrapped in quotes. To avoid a clash, use single quotes around `$searchterm`.

5. Execute the query, and get the number of rows returned. The complete code in the PHP block above the `DOCTYPE` declaration looks like this:

```
if (isset($_GET['go'])) {
  require_once('../includes/connection.inc.php');
  $conn = dbConnect('read');
  $searchterm = '%' . $conn->real_escape_string($_GET['search']) . '%';
  $sql = "SELECT * FROM images WHERE caption LIKE '$searchterm'";
  $result = $conn->query($sql) or die($conn->error);
  $numRows = $result->num_rows;
}
```

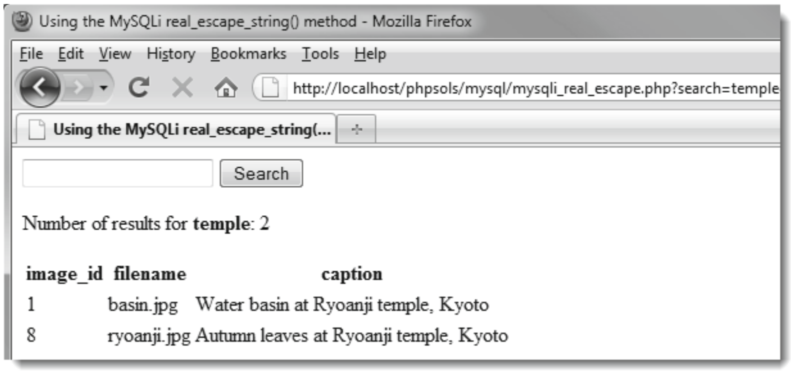
6. Add the PHP code to the body of the page to display the results:

```
<?php if (isset($numRows)) { ?>
<p>Number of results for <b><?php echo htmlentities($_GET['search'], ➥
   ENT_COMPAT, 'utf-8'); ?></b>: <?php echo $numRows; ?></p>
<?php if ($numRows) { ?>
<table>
  <tr>
    <th scope="col">image_id</th>
    <th scope="col">filename</th>
    <th scope="col">caption</th>
  </tr>
  <?php while ($row = $result->fetch_assoc()) { ?>
  <tr>
    <td><?php echo $row['image_id']; ?></td>
    <td><?php echo $row['filename']; ?></td>
    <td><?php echo $row['caption']; ?></td>
  </tr>
  <?php } ?>
</table>
<?php }
} ?>
```

The first conditional statement is wrapped around the paragraph and table, preventing them from being displayed if `$numRows` doesn't exist, which happens when the page is first loaded. If the form has been submitted, `$numRows` will have been set, so the search term is redisplayed using `htmlentities()` (see Chapter 5), and the value of `$numRows` reports the number of matches.

If the query returns no results, `$numRows` is 0, which is treated as `false`, so the table is not displayed. If `$numRows` contains anything other than 0, the table is displayed, and the `while` loop displays the results of the query.

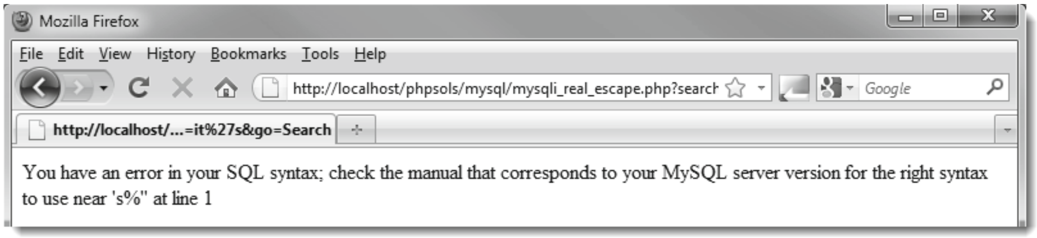
7. Save the page, and load it into a browser. Enter some text in the search field, and click **Search**. The number of results is displayed, together with any captions that contain the search term, as shown in the following screenshot:


Using the MySQLi real_escape_string() method - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/phpsols/mysql/mysqli_real_escape.php?search=temple
Using the MySQLi real_escape_string(...
Search

Number of results for **temple**: 2

image_id	filename	caption
1	basin.jpg	Water basin at Ryoanji temple, Kyoto
8	ryoanji.jpg	Autumn leaves at Ryoanji temple, Kyoto

If you don't use `real_escape_string()` or a prepared statement, the search form still works most of the time. But if the search term includes an apostrophe or quotation marks, your page will fail to load correctly, and a SQL syntax error will be displayed like this:


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/phpsols/mysql/mysqli_real_escape.php?search
Google
http://localhost/...=it%27s&go=Search

```
You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax
to use near 's%'' at line 1
```

Worse, it leaves your database wide open to malicious attack.

Although `real_escape_string()` escapes quotes and other control characters in the submitted value, you still need to wrap strings in quotes in the SQL query. The `LIKE` keyword must always be followed by a string, even if the search term is limited to numbers.

Embedding variables in MySQLi prepared statements

Instead of incorporating variables directly in the SQL query, you use question marks as placeholders like this:

```
$sql = 'SELECT image_id, filename, caption FROM images WHERE caption LIKE ?';
```

Using a MySQLi prepared statement involves the following steps:

1. Initialize the statement.
2. Pass the SQL query to the statement to make sure it's valid.
3. Bind the variable(s) to the query.
4. Bind results to variables (optional).
5. Execute the statement.
6. Store the result (optional).
7. Fetch the result(s).

To initialize the prepared statement, call the `stmt_init()` method on the database connection, and store it in a variable like this:

```
$stmt = $conn->stmt_init();
```

You then pass the SQL query to `$stmt->prepare()`. This checks that you haven't used question mark placeholders in the wrong place, and that when everything is put together, the query is valid SQL. If there are any mistakes, `$stmt->prepare()` returns `false`, so you need to enclose the next steps in a conditional statement to ensure they run only if everything is still OK.

Error messages can be accessed by using `$stmt->error`.

Binding the parameters means replacing the question marks with the actual values held in the variables. This is what protects your database from SQL injection. You pass the variables to `$stmt->bind_param()` in the same order as you want them inserted into the SQL query, together with a first argument specifying the data type of each variable, again in the same order as the variables. The data type must be specified by one of the following four characters:

- `b`: Binary (such as an image, Word document, or PDF file)
- `d`: Double (floating point number)
- `i`: Integer (whole number)
- `s`: String (text)

The number of variables passed to `$stmt->bind_param()` must be exactly the same as the number of question mark placeholders. For example, to pass a single value as a string, use this:

```
$stmt->bind_param('s', $_GET['words']);
```

To pass two values, the `SELECT` query needs two question marks as placeholders, and both variables need to be bound with `bind_param()` like this:

```
$sql = 'SELECT * FROM products WHERE price < ? AND type = ?';
$stmt = $conn->stmt_init();
```

```
$stmt->prepare($sql);
$stmt->bind_param('ds', $_GET['price'], $_GET['type']);
```

The first argument to `bind_param()`,`'ds'`, specifies `$_GET['price']` as a floating point number, and `$_GET['type']` as a string.

Optionally, you can bind the results of a `SELECT` query to variables with the `bind_result()` method. This avoids the need to extract each row and access the results as `$row['column_name']`. To bind the results, you must name each column specifically in the `SELECT` query. List the variables you want to use in the same order, and pass them as arguments to `bind_result()`. To bind the results of the query at the beginning of this section, use this:

```
$stmt->bind_result($image_id, $filename, $caption);
```

This allows you to access the results directly as `$image_id`, `$filename`, and `$caption`.

Once the statement has been prepared, you call `$stmt->execute()`, and the result is stored in `$stmt`.

To access the `num_rows` property, you must first store the result like this:

```
$stmt->store_result();
$numRows = $stmt->num_rows;
```

Using `store_result()` is optional, but if you don't use it, `num_rows` returns 0.

To loop through the results of a `SELECT` query executed with a prepared statement, use the `fetch()` method. If you have bound the results to variables, do it like this:

```
while ($stmt->fetch()) {
  // display the bound variables for each row
}
```

If you don't bind the result to variables, use `$row = $stmt->fetch()`, and access each variable as `$row['column_name']`.

When you have finished with a result, you can free the memory by using the `free_result()` method. The `close()` method frees the memory used by the prepared statement.

PHP Solution 11-8: Using a MySQLi prepared statement in a search

This PHP solution shows how to use a MySQLi prepared statement with a `SELECT` query and demonstrates binding the result to named variables.

1. Copy `mysql_prepared_01.php` from the `ch11` folder and save it in the `mysql` folder as `mysql_prepared.php`. It contains the same search form and results table as used in PHP Solution 11-7.

2. In a PHP code block above the `DOCTYPE` declaration, create a conditional statement to include `connection.inc.php` and create a MySQL read-only connection when the search form is submitted. The code looks like this:

```
if (isset($_GET['go'])) {
  require_once('../includes/connection.inc.php');
  $conn = dbConnect('read');
}
```

3. Next, add the SQL query inside the conditional statement. The query needs to name the three columns you want to retrieve from the `images` table. Use a question mark as the placeholder for the search term like this:

```
$sql = 'SELECT image_id, filename, caption FROM images
        WHERE caption LIKE ?';
```

4. Before passing the user-submitted search term to the `bind_param()` method, you need to add the wildcard characters to it and assign it to a new variable like this:

```
$searchterm = '%'. $_GET['search'] .'%';
```

5. You can now create the prepared statement. The finished code in the PHP block above the `DOCTYPE` declaration looks like this:

```
if (isset($_GET['go'])) {
  require_once('../includes/connection.inc.php');
  $conn = dbConnect('read');
  $sql = 'SELECT image_id, filename, caption FROM images
          WHERE caption LIKE ?';
  $searchterm = '%'. $_GET['search'] .'%';
  $stmt = $conn->stmt_init();
  if ($stmt->prepare($sql)) {
    $stmt->bind_param('s', $searchterm);
    $stmt->bind_result($image_id, $filename, $caption);
    $stmt->execute();
    $stmt->store_result();
    $numRows = $stmt->num_rows;
  } else {
    echo $stmt->error;
  }
}
```

This initializes the prepared statement and assigns it to `$stmt`. The SQL query is then passed to the `prepare()` method, which checks the validity of the query's syntax. If there's a problem with the syntax, the `else` block displays the error message. If the syntax is OK, the rest of the script inside the conditional statement is executed.

The code is wrapped in a conditional statement for testing purposes only. If there's an error with your prepared statement, `echo $stmt->error;` displays a MySQL error message to help identify the problem. In a live website, you should remove the conditional statement, and call `$stmt->prepare($sql);` directly

The first line inside the conditional statement binds `$searchterm` to the `SELECT` query, replacing the question mark placeholder. The first argument tells the prepared statement to treat it as a string.

The next line binds the results of the `SELECT` query to `$image_id`, `$filename`, and `$caption`. These need to be in the same order as in the query. I have named the variables after the columns they represent, but you can use any variables you want.

Then the prepared statement is executed and the result stored. Note that the result is stored in the `$stmt` object. You don't assign it to a variable.

Assigning `$stmt->store_result()` to a variable doesn't store the database result. It records only whether the result was successfully stored in the `$stmt` object.

Finally, the number of rows retrieved by the query is stored in `$numRows`.

6. Add the following code after the search form to display the result:

```
<?php if (isset($numRows)) { ?>
<p>Number of results for <b><?php echo htmlentities($_GET['search'], ➥
  ENT_COMPAT, 'utf-8'); ?></b>: <?php echo $numRows; ?></p>
<?php if ($numRows) { ?>
<table>
  <tr>
    <th scope="col">image_id</th>
    <th scope="col">filename</th>
    <th scope="col">caption</th>
  </tr>
  <?php while ($stmt->fetch()) { ?>
  <tr>
    <td><?php echo $image_id; ?></td>
    <td><?php echo $filename; ?></td>
    <td><?php echo $caption; ?></td>
  </tr>
  <?php } ?>
</table>
<?php }
} ?>
```

Most of this code is the same as in PHP Solution 11-7. The difference lies in the `while` loop that displays the results. Instead of using the `fetch_assoc()` method on a result object and storing the result in `$row`, it simply calls the `fetch()` method on the prepared statement. There's no need to store the current record as `$row`, because the values from each column have been bound to `$image_id`, `$filename`, and `$caption`.

You can compare your code with `mysqli_prepared_02.php` in the `ch11` folder.

Embedding variables in PDO prepared statements

Whereas MySQLi always uses question marks as placeholders in prepared statements, PDO offers several options. I'll describe the two most useful: question marks and named placeholders.

Question mark placeholders Instead of embedding variables in the SQL query, you replace them with question marks like this:

```
$sql = 'SELECT image_id, filename, caption FROM images WHERE caption LIKE ?';
```

This is identical to MySQLi. However, the way that you bind the values of the variables to the placeholders is completely different. It involves just two steps, as follows:

1. Prepare the statement to make sure the SQL is valid.
2. Execute the statement by passing the variables to it as an array.

Assuming you have created a PDO connection called `$conn`, the PHP code looks like this:

```
// prepare statement
$stmt = $conn->prepare($sql);
// execute query by passing array of variables
$stmt->execute(array($_GET['words']));
```

The first line of code prepares the statement and stores it as `$stmt`. The second line binds the values of the variable(s) and executes the statement all in one go. The variables must be in the same order as the placeholders. Even if there is only one placeholder, the variable must be passed to `execute()` as an array. The result of the query is stored in `$stmt`.

Named placeholders Instead of embedding variables in the SQL query, you replace them with named placeholders beginning with a colon like this:

```
$sql = 'SELECT image_id, filename, caption FROM images WHERE caption LIKE :search';
```

With named placeholders, you can either bind the values individually or pass an associative array to `execute()`. When binding the values individually, the PHP code looks like this:

```
$stmt = $conn->prepare($sql);
// bind the parameters and execute the statement
$stmt->bindParam(':search', $_GET['words'], PDO::PARAM_STR);
$stmt->execute();
```

You pass three arguments to `$stmt->bindParam()`: the name of the placeholder, the variable that you want to use as its value, and a constant specifying the data type. The main constants are as follows:

- `PDO::PARAM_INT`: Integer (whole number)
- `PDO::PARAM_LOB`: Binary (such as an image, Word document, or PDF file)
- `PDO::PARAM_STR`: String (text)

There isn't a constant for floating point numbers, but the third argument is optional, so you can just leave it out. Alternatively, use `PDO::PARAM_STR`. This wraps the value in quotes, but MySQL converts it back to a floating point number.

If you pass the variables as an associative array, you can't specify the data type. The PHP code for the same example using an associative array looks like this:

```
// prepare statement
$stmt = $conn->prepare($sql);
// execute query by passing array of variables
$stmt->execute(array(':search' => $_GET['words']));
```

In both cases, the result of the query is stored in `$stmt`.

Error messages can be accessed in the same way as with a PDO connection. However, instead of calling the `errorInfo()` method on the connection object, use it on the PDO statement like this:

```
$error = $stmt->errorInfo();
if (isset($error[2])) {
  echo $error[2];
}
```

To bind the results of a `SELECT` query to variables, each column needs to bound separately using the `bindColumn()` method before calling `execute()`. The `bindColumn()` method takes two arguments. The first argument can be either the name of the column or its number counting from 1. The number comes from its position in the `SELECT` query, not the order it appears in the database table. So, to bind the result from the `filename` column to `$filename`, either of the following is acceptable:

```
$stmt->bindColumn('filename', $filename);
$stmt->bindColumn(2, $filename);
```

PHP Solution 11-9: Using a PDO prepared statement in a search

This PHP solution shows how to embed the user-submitted value from a search form into a SELECT query with a PDO prepared statement. It uses the same search form as the MySQLi versions in PHP Solutions 11-7 and 11-8.

1. Copy `pdo_prepared_01.php` from the `ch11` folder, and save it in the `mysql` folder as `pdo_prepared.php`.
2. Add the following code in a PHP block above the `DOCTYPE` declaration:

```
if (isset($_GET['go'])) {
  require_once('../includes/connection.inc.php');
  $conn = dbConnect('read', 'pdo');
  $sql = 'SELECT image_id, filename, caption FROM images
          WHERE caption LIKE :search';
  $searchterm = '%'. $_GET['search'] .'%';
  $stmt = $conn->prepare($sql);
  $stmt->bindParam(':search', $searchterm, PDO::PARAM_STR);
  $stmt->bindColumn('image_id', $image_id);
  $stmt->bindColumn('filename', $filename);
  $stmt->bindColumn(3, $caption);
  $stmt->execute();
  $numRows = $stmt->rowCount();
}
```

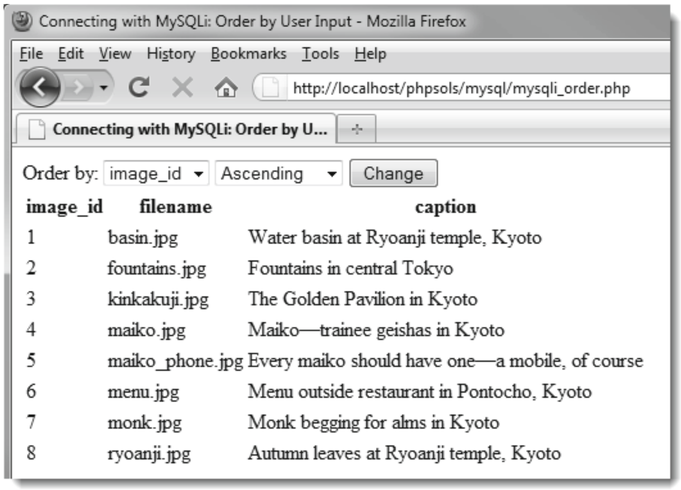
When the form is submitted, this includes the connection file and creates a PDO connection to MySQL. The prepared statement uses `:search` as a named parameter in place of the user-submitted value. Like MySQLi prepared statements, you need to add the `%` wildcard characters to the search term before binding it to the prepared statement with `bindParam()`. The results are bound to `$image_id`, `$filename`, and `$caption`. The first two use the column names, but the `caption` column is referred to by its position in the `SELECT` query.

3. The code that displays the results is identical to step 6 in PHP Solution 11-8. The finished file is in `pdo_prepared_02.php` in the `ch11` folder.

PHP Solution 11-10: Changing column options through user input

This PHP solution shows how to change the name of SQL keywords in a `SELECT` query through user input. SQL keywords cannot be wrapped in quotes, so using prepared statements or the MySQLi `real_escape_string()` method won't work. Instead, you need to ensure that the user input matches an array of expected values. If no match is found, use a default value instead. The technique is identical for MySQLi and PDO.

1. Copy either `mysqli_order_01.php` or `pdo_order_01.php` from the `ch11` folder, and save it in the `mysql` folder. Both versions select all records from the `images` table and display the results in table. The pages also contain a form that allows the user to select the name of a column to sort the results in either ascending or descending order. In their initial state, the form is inactive. The pages display the details sorted by `image_id` in ascending order like this:


Connecting with MySQLi: Order by User Input - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/phpsols/mysql/mysqli_order.php
Connecting with MySQLi: Order by U...
Change

Order by: image_id Ascending

image_id	filename	caption
1	basin.jpg	Water basin at Ryoanji temple, Kyoto
2	fountains.jpg	Fountains in central Tokyo
3	kinkakuji.jpg	The Golden Pavilion in Kyoto
4	maiko.jpg	Maiko—trainee geishas in Kyoto
5	maiko_phone.jpg	Every maiko should have one—a mobile, of course
6	menu.jpg	Menu outside restaurant in Pontocho, Kyoto
7	monk.jpg	Monk begging for alms in Kyoto
8	ryoanji.jpg	Autumn leaves at Ryoanji temple, Kyoto

2. Amend the code in the PHP block above the `DOCTYPE` declaration like this (the following listing shows the MySQLi version, but the changes highlighted in bold type are the same for PDO):

```
require_once('../includes/connection.inc.php');
// connect to MySQL
$conn = dbConnect('read');
// set default values
$col = 'image_id';
$dir = 'ASC';
// create arrays of permitted values
$columns = array('image_id', 'filename', 'caption');
$direction = array('ASC', 'DESC');
// if the form has been submitted, use only expected values
if (isset($_GET['column']) && in_array($_GET['column'], $columns)) {
  $col = $_GET['column'];
}
if (isset($_GET['direction']) && in_array($_GET['direction'], $direction)) {
  $dir = $_GET['direction'];
}
// prepare the SQL query using sanitized variables
$sql = "SELECT * FROM images
        ORDER BY $col $dir";
// submit the query and capture the result
$result = $conn->query($sql) or die(mysqli_error());
```

The new code defines two variables, `$col` and `$dir`, that are embedded directly in the `SELECT` query. Because they have been assigned default values, the query displays the results sorted by the `image_id` column in ascending order when the page first loads.

Two arrays, `$columns` and `$direction`, then define permitted values: the column names, and the `ASC` and `DESC` keywords. These arrays are used by the conditional statements that check the `$_GET` array for `column` and `direction`. The submitted values are reassigned to `$col` and `$dir` only if they match a value in the `$columns` and `$direction` arrays respectively. This prevents any attempt to inject illegal values into the SQL query.

3. Edit the `<option>` tags in the drop-down menus so they display the selected values for `$col` and `$dir` like this:

```
<select name="column" id="column">
  <option <?php if ($col == 'image_id') echo 'selected'; ?>>image_id</option>
  <option <?php if ($col == 'filename') echo 'selected'; ?>>filename</option>
  <option <?php if ($col == 'caption') echo 'selected'; ?>>caption</option>
</select>
<select name="direction" id="direction">
  <option value="ASC" <?php if ($dir == 'ASC') echo 'selected'; ?>> ➥
    Ascending</option>
  <option value="DESC" <?php if ($dir == 'DESC') echo 'selected'; ?>> ➥
    Descending</option>
</select>
```

4. Save the page, and test it in a browser. You can change the sort order of the display by selecting the values in the drop-down menus and clicking **Change**. However, if you try to inject an illegal value through the query string, the page uses the default values of `$col` and `$dir` to display the results sorted by `image_id` in ascending order.

   You can check your code against `mysqli_order_02.php` and `pdo_order_02.php` in the `ch11` folder.

Chapter review

PHP provides three methods of communicating with MySQL:

- **The original MySQL extension, which is no longer actively maintained**: It should not be used for new projects. If you need to maintain an existing site, you can easily recognize whether it uses the original MySQL extension, because all functions begin with `mysql_`. For help using it, consult the first edition of this book or use the online documentation at `http://docs.php.net/manual/en/book.mysql.php`.
- **The MySQL Improved (MySQLi) extension**: This is recommended for all new MySQL projects. It requires PHP 5.0 and MySQL 4.1 or higher. It's more efficient, and has the added safety of prepared statements.
- **The PHP Data Objects (PDO) abstraction layer, which is software-neutral**: You should choose this option if your projects are likely to need to be adapted to use other databases.

Although PHP communicates with the database and stores the results, queries need to be written in SQL, the standard language used to query a relational database. This chapter showed how to retrieve information stored in a database table using a `SELECT` statement, refining the search with a `WHERE` clause, and changing the sort order with `ORDER BY`. You also learned several techniques to protect queries from SQL injection, including prepared statements, which use placeholders instead of embedding variables directly in a query.

In the next chapter, you'll put this knowledge to practical use creating an online photo gallery.