PHP 101 (part 9): SQLite My Fire! – Part 1

Hard Choices

If you've been paying attention, you now know how to use PHP's MySQL API to perform queries and process result sets. You might even have started thinking about how to re-program your site to run off a MySQL database. All of this is a Good Thing – it means you're getting comfortable with using PHP's database support to power your applications – but there's still a little further to go.

As you saw in Part Eight, enabling MySQL support in PHP 5.0 is not as simple as it used to be. Instead of supporting MySQL out of the box, PHP now requires you to make all kinds of decisions about versions and libraries before allowing you to hook your scripts up to a MySQL database. If you're lazy (and deep down, we both know you are), you might instead prefer to try a simpler option: the SQLite database engine. Built-in SQLite support is new to PHP 5.0, and offers users a lightweight database system that is fast, efficient and gets the job done. Since it's enabled by default in PHP 5.0, it provides a viable alternative to MySQL; you can use it out of the box, without spending time on version checks and library downloads; just install PHP 5 and start typing. That's why I'm devoting a whole tutorial to it – so get out of bed, make yourself some coffee and let's get started!

Making New Friends

Before getting into the code, let's make sure that you have a clear idea of what SQLite is (and isn't). Unlike MySQL, which operates on a

client-server paradigm, SQLite is a file-based database engine and uses file I/O (input/output) functions to store and read databases from files on disk. It's also much, much smaller than MySQL – the command-line version of SQLite weighs in at under 200 KB – and supports most of the SQL commands you're used to.

This small size shouldn't deceive you, however – according to the official SQLite Web site,

SQLite supports databases up to 2 terabytes in size and is actually faster than MySQL in certain situations. SQLite database files are easily portable, and SQLite databases created on Windows work fine on *NIX platforms and vice-versa.

One of SQLite's more interesting aspects is that it is completely typeless(即變量沒有類型). Fields in an SQLite database need not be associated with a specific type, and even if they are, you can still insert values of different types into them (there is one exception to this rule, but I'll get to that later). This is important, because it means that if you're concerned about values of the wrong type getting into your tables, you need to write code to implement type checking in your application.

Another important difference between MySQL and SQLite lies in their licensing policies: unlike MySQL, SQLite source code is completely public-domain, which means that you can use and distribute it however you choose in both commercial and non-commercial products. Take a look at http://sqlite.org/copyright.html

for more on this.

In order to use SQLite and PHP together, your PHP build must include SQLite. This is enabled by default in both the UNIX and Windows versions

of PHP 5. Read more about this at
http://www.php.net/manual/en/ref.sqlite.php.
If you're a PHP 4.x user, though, don't lose heart – you can still use SQLite,
by manually downloading and installing php_sqlite.dll from
http://snaps.php.net
(Windows) or the latest tarball from http://pecl.php.net/package/SQLite
(UNIX). You don't need to download anything else; the SQLite 'client' is its own
engine.

The Bookworm Turns

As with MySQL, you use regular SQL commands to interact with an
SQLite database. The exact SQL syntax used by SQLite is listed
at http://sqlite.org/lang.html,
but for most operations SQL commands are standard.
Here's an example, which sets up the table I'll be using in this
tutorial:

```
C:\WINDOWS\Desktop\sqlite>sqlite library.db

SQLite version 2.8.15

Enter ".help" for instructions

sqlite> create table books (
   ...> id integer primary key,
```

```
   ...> title varchar(255) not null,

   ...> author varchar(255) not null

   ...>);

sqlite> insert into books (title, author) values ('The Lord Of The Rings', 'J.R.R.
Tolkien');

sqlite> insert into books (title, author) values ('The Murders In The Rue Morgue',
'Edgar Allen Poe');
sqlite> insert into books (title, author) values ('Three Men In A Boat', 'Jerome K.
Jerome');

sqlite> insert into books (title, author) values ('A Study In Scarlet', 'Arthur Conan
Doyle');

sqlite> insert into books (title, author) values ('Alice In Wonderland', 'Lewis
Carroll');
```

```
sqlite> .exit
```

You can enter these commands either interactively or

non-interactively through the SQLite commandline program,

which is available at http://sqlite.org/download.html

as a precompiled binary for Windows and Linux. SQLite 2.* is the

version currently used in both branches of PHP, with SQLite 3.* support

anticipated for PDO and later PHP 5.* releases.

Extract the downloaded files to a directory of your choice, cd into it from

your shell or DOS box and type ‘sqlite’. You should see the SQLite version

information and the line:

```
Enter ".help" for instructions
```

Read

http://sqlite.org/sqlite.html

for more information on how to use the commandline program.

Once the data has been imported into the database file

`library.db`, run a quick `SELECT`

query to check if everything is working as it should:

```
sqlite> select * from books;
1|The Lord Of The Rings|J.R.R. Tolkien
[redacted]
3|Three Men In A Boat|Jerome K.
Jerome
[redacted]
5|Alice In Wonderland|Lewis Carroll
```

If you saw the same output as above, you’re good to go!

Anatomy Class

Now, use PHP to communicate with SQLite, generate the same result set and format it as an HTML page. Here’s the code:

```
<html>

<head></head>

<body>

<?php

// set path of database file

$db = $_SERVER['DOCUMENT_ROOT']."/../library.db";

// open database file

$handle = sqlite_open($db) or die("Could not open database");

// generate query string

$query = "SELECT * FROM books";

// execute query

$result = sqlite_query($handle, $query) or die("Error in query:

".sqlite_error_string(sqlite_last_error($handle)));
```

```
// if rows exist

if (sqlite_num_rows($result) > 0) {

    // get each row as an array

    // print values

    echo "<table cellpadding=10 border=1>";
        while($row = sqlite_fetch_array($result)) {

        echo "<tr>";

        echo "<td>".$row[0]."</td>";
                echo "<td>".$row[1]."</td>";

        echo "<td>".$row[2]."</td>";
                echo "</tr>";

    }

    echo "</table>";

}
// all done

// close database file
```

```
sqlite_close($handle);
?>

</body>

</html>
```

If all goes well, you should see something like this:

 (原網站上也顯示不出來)

If you remember what you learned in Part Eight,
the PHP script above should be easy to decipher. In case you don't, here's a fast rundown:

1.The ball starts rolling with the `sqlite_open()` function, which
accepts the name of the database file as argument and attempts to open
it. If this database file cannot be found, an empty database file will
be created with the supplied name (assuming the script has write access
to the directory).

```
<?php
$db = $_SERVER['DOCUMENT_ROOT']."/../library.db";
$handle = sqlite_open($db) or die("Could not open database");
?>
```

The database file `library.db` needs to be kept somewhere it can't be accessed through the browser by visitors to your site. That means that you need to create it outside your public_html, www or htdocs directory, in a directory that allows your scripts read/write permissions. Web hosting companies generally will offer a space above your web-visible directory where you can do this.

`$_SERVER['DOCUMENT_ROOT']."/.."` is the directory directly above your web-visible directory.

If successful, the `sqlite_open()` function returns a handle to the file, which is stored in the variable `$handle` and is used for all subsequent communication with the database.

2.The next step is to create and execute the query, with the `sqlite_query()` function.

```
<?php
$query = "SELECT * FROM books";

$result = sqlite_query($handle, $query) or die("Error in query: 
".sqlite_error_string(sqlite_last_error($handle)));
?>
```

This function also needs two parameters: the database handle and the query string. Depending on whether or not the query was successful, the

function returns true or false; in the event of a failure, the `sqlite_error_string()` and `sqlite_last_error()` functions can be used to display the error that took place.

3.If `sqlite_query()` is successful, the result set returned by the query is stored in the variable `$result`. You can retrieve the records in the result set with the `sqlite_fetch_array()` function, which fetches a single row of data as an array called `$row`. Fields in that record are represented as array elements, and can be accessed using standard index notation.

Each time you call `sqlite_fetch_array()`, the next record in the result set is returned. This makes `sqlite_fetch_array()` very suitable for use in a `while()` loop, in much the same way as `mysql_fetch_row()` was used earlier.

```
<?php
if (sqlite_num_rows($result) > 0) {

   echo "<table cellpadding=10 border=1>";

   while($row = sqlite_fetch_array($result)) {
            echo "<tr>";

      echo "<td>".$row[0]."</td>";
```

```
            echo "<td>".$row[1]."</td>";

        echo "<td>".$row[2]."</td>";
            echo "</tr>";

    }

    echo "</table>";

}
?>
```

The number of records returned by the query can be retrieved with the `sqlite_num_rows()` function. Or, if what you're really interested in is the number of fields in the result set, use the `sqlite_num_fields()` function instead. Of course, these are only applicable with queries that actually return records; it doesn't really make sense to use them with `INSERT`, `UPDATE` or `DELETE` queries.

4.Once you're done, it's a good idea to close the database handle and return the used memory to the system, with a call to `sqlite_close()`:

```
<?php
sqlite_close($handle);
?>
```

In PHP 5 you can also use the SQLite API in an object-oriented way,

wherein each of the functions above becomes a method of the

`SQLiteDatabase()` object. Take a look at this next listing,

which is equivalent to the one above:

```
<html>

<head></head>

<body>
<?php

// set path of database file

$file = $_SERVER['DOCUMENT_ROOT']."/../library.db";
// create database object

$db = new SQLiteDatabase($file) or die("Could not open database");
// generate query string

$query = "SELECT *  FROM books";
// execute query

// return result object

$result = $db->query($query) or die("Error in query");
// if rows exist
```

```
if ($result->numRows() > 0) {

    // get each row as an array

    // print values

    echo "<table cellpadding=10 border=1>";
        while($row = $result->fetch()) {

        echo "<tr>";

        echo "<td>".$row[0]."</td>";
                echo "<td>".$row[1]."</td>";

        echo "<td>".$row[2]."</td>";
                echo "</tr>";

    }

    echo "</table>";

}
// all done

// destroy database object
```

```
unset($db);
?>

</body>

</html>
```

Here, the `new` keyword is used to instantiate an object of the class `SQLiteDatabase()` by passing the object constructor the name of the database file. If the database file does not already exist, a new database file is created. The resulting object, stored in `$db`, then exposes methods and properties to perform queries. Every query returns an instance of the class `SQLiteResult()`, which in turn exposes methods for fetching and processing records.

If you look closely at the two scripts above, you'll see the numerous similarities between the procedural function names and the object method names. While the correspondence between the two is not perfect, it's usually close enough to make it possible to guess the one if you know the other.

Different Strokes

As with the MySQL API, PHP's SQLite API offers you more than one way to skin a cat. For example, you can retrieve each row as an object with the `sqlite_fetch_object()` method, and access field values

by using the field names as object properties. Here’s an example:

```
<html>
<head></head>

<body>

<?php

// set path of database file

$db = $_SERVER['DOCUMENT_ROOT']."/../library.db";
// open database file
$handle = sqlite_open($db) or die("Could not open database");
// generate query string

$query = "SELECT * FROM books";
// execute query

$result = sqlite_query($handle, $query) or die("Error in query:
".sqlite_error_string(sqlite_last_error($handle)));
// if rows exist

if (sqlite_num_rows($result) > 0) {

    // get each row as an object
```

```
    // print field values as object properties

    echo "<table cellpadding=10 border=1>";
        while($obj = sqlite_fetch_object($result)) {

        echo "<tr>";

        echo "<td>".$obj->id."</td>";
                echo "<td>".$obj->title."</td>";

        echo "<td>".$obj->author."</td>";
                echo "</tr>";

    }

    echo "</table>";

}
// all done

// close database file

sqlite_close($handle);
?>
```

```
</body>
```

```
</html>
```

Another option is to retrieve the complete result set in one fell swoop with the `sqlite_fetch_all()` function. This function retrieves the complete set of records as an array of arrays; each element of the outer array represents a record, and is itself structured as an array whose elements represent fields in that record.

Here’s an example, which might make this clearer:

```
<html>
```

```
<head></head>
```

```
<body>
```

```
<?php
```

```
// set path of database file
```

```
$db = $_SERVER['DOCUMENT_ROOT']."/../library.db";
// open database file

$handle = sqlite_open($db) or die("Could not open database");
```

```
// generate query string

$query = "SELECT * FROM books";
// execute query

$result = sqlite_query($handle, $query) or die("Error in query:
".sqlite_error_string(sqlite_last_error($handle)));
// get the complete result set as a series of nested arrays

$data = sqlite_fetch_all($result);
// all done

// close database file

sqlite_close($handle);
// check the array to see if it contains at least one record
if (sizeof($data) > 0) {

    echo "<table cellpadding=10 border=1>";

    // iterate over outer array (rows)

    // print values for each element of inner array (columns)

    foreach ($data as $row) {
            echo "<tr>";
```

```
        echo "<td>".$row[0]."</td>";
                echo "<td>".$row[1]."</td>";

        echo "<td>".$row[2]."</td>";
                echo "</tr>";

    }

    echo "</table>";

}

?>

</body>

</html>
```

In all the previous examples, the database remained open while the result set was processed, because records were retrieved one after another with the `sqlite_fetch_array()` or `sqlite_fetch_object()` functions. The example above is unique in that the database can be closed before the result set array is processed. This is because the entire result set is retrieved at once and stored in the `$data` array, so there really isn't any need to leave the database

open while processing it.

If your result set contains only a single field, use the `sqlite_fetch_single()` function, which retrieves the value of the first field of a row. The PHP manual puts it best when it says “this is the most optimal way to retrieve data when you are only interested in the values from a single column of data.” Take a look:

```
<html>

<head></head>

<body>

<?php

// set path of database file

$db = $_SERVER['DOCUMENT_ROOT']."/../library.db";
// open database file

$handle = sqlite_open($db) or die("Could not open database");
// generate query string

// this query returns only a single record with a single field
```

```
$query = "SELECT author FROM books WHERE title = 'A Study In Scarlet'";
// execute query

$result = sqlite_query($handle, $query) or die("Error in query:
".sqlite_error_string(sqlite_last_error($handle)));
// if a row exists

if (sqlite_num_rows($result) > 0) {

    // get the value of the first field of the first row

    echo sqlite_fetch_single($result);

}
// all done

// close database file

sqlite_close($handle);

?>

</body>

</html>
```

You can even use the `sqlite_fetch_single()` function in combination with a `while()` loop to iterate over a result set containing many records but a single field. Notice also my usage of the `sqlite_has_more()` function, to check if the next row exists or not.

```
<html>

<head></head>

<body>

<?php

// set path of database file

$db = $_SERVER['DOCUMENT_ROOT']."/../library.db";
// open database file

$handle = sqlite_open($db) or die("Could not open database");
// generate query string

$query = "SELECT DISTINCT author FROM books";
// execute query

$result = sqlite_query($handle, $query) or die("Error in query:
```

```
".sqlite_error_string(sqlite_last_error($handle)));
// if rows exist

if (sqlite_num_rows($result) > 0) {

    echo "<table cellpadding=10 border=1>";

    // check for more rows

    while (sqlite_has_more($result)) {
                // get first field from each row

        // print values

        $row = sqlite_fetch_single($result);

        echo "<tr>";

        echo "<td>".$row."</td>";
                echo "</tr>";

    }

    echo "</table>";

}
// all done
```

```
// close database file

sqlite_close($handle);
?>

</body>

</html>
```

You can, of course, do the same thing using object notation in PHP 5. However, you need to know that `sqlite_has_more()` is one function that really doesn't translate to its object method name; in an OO script, you would need to call `$result->valid();`.

This script is the OO equivalent of the one above:

```
<html>

<head></head>

<body>

<?php
```

```
// set path of database file

$file = $_SERVER['DOCUMENT_ROOT']."/../library.db";
// create database object

$db = new SQLiteDatabase($file) or die("Could not open database");
// generate query string

$query = "SELECT DISTINCT author FROM books";
// execute query

$result = $db->query($query) or die("Error in query");
// if rows exist

if ($result->numRows() > 0) {
        echo "<table cellpadding=10 border=1>";

    // check for more rows

    while ($result->valid()) {

        // get first field from each row

        // print values

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

```
            echo "<tr>";

        echo "<td>".$row."</td>";
            echo "</tr>";

    }

    echo "</table>";

}
// all done

// destroy database object

unset($db);
?>

</body>

</html>
```