Writing MySQL Scripts with PHP and PDO

Paul DuBois
paul@kitebird.com

Document revision: 1.02
Last update: 2013-08-11

PHP makes it easy to write scripts that access databases, enabling you to create dynamic web pages that incorporate database content. PHP includes several specialized database-access interfaces that take the form of separate sets of functions for each database system. There is one set for MySQL, another for InterBase, another for PostgreSQL, and so forth. However, having a different set of functions for each database makes PHP scripts non-portable at the lexical (source code) level. For example, the function for issuing an SQL statement is named `mysql_query()`, `ibase_query()`, or `pg_exec()`, depending on whether you are using MySQL, InterBase, or PostgreSQL.

In PHP 5 and up, you can avoid this problem by using the PHP Data Objects (PDO) extension. PDO supports database access in an engine-independent manner based on a two-level architecture:

- The top level provides an interface that consists of a set of classes and methods that is the same for all database engines supported by PDO. The interface hides engine-specific details so that script writers need not think about which set of functions to use.
- The lower level consists of individual drivers. Each driver supports a particular database engine and translates between the top-level interface seen by script writers and the database-specific interface required by the engine. This provides you the flexibility of using any database for which a driver exists, without having to consider driver-specific details.

This architectural approach has been used successfully with other languages—for example, to develop the DBI (Perl, Ruby), DB-API (Python), and JDBC (Java) database access interfaces. It's also been used with PHP before: PHPLIB, MetaBase, and PEAR DB are older packages that provide a uniform database-independent interface across different engines.

I have written elsewhere about using the PEAR DB module for writing PHP scripts that perform database processing in an engine-independent manner (see ''Resources''). This document is similar but covers PDO instead. The examples use the driver for MySQL.

Preliminary Requirements

PDO uses object-oriented features available only in PHP 5 and up, so you must have PHP 5 or newer installed to use PDO for writing scripts that access MySQL.

PDO uses classes and objects to present an object-oriented interface. This article assumes that you are familiar with PHP's approach to object-oriented programming. If you are not, you may wish to review the ''Classes and Objects'' chapter of the PHP Manual.

Writing PDO Scripts

Scripts that use the PDO interface to access MySQL generally perform the following operations:

1. Connect to the MySQL server by calling `new PDO()` to obtain a database handle object.
2. Use the database handle to issue SQL statements or obtain statement handle objects.
3. Use the database and statement handles to retrieve information returned by the statements.
4. Disconnect from the server when the database handle is no longer needed.

The next sections discuss these operations in more detail.

Connecting to and Disconnecting from the MySQL Server

To establish a connection to a MySQL server, specify a data source name (DSN) containing connection parameters, and optionally the username and password of the MySQL account to use. To connect to the MySQL server on the local host to access the `test` database with a username and password of `testuser` and `testpass`, the connection sequence looks like this:

```
$dbh = new PDO("mysql:host=localhost;dbname=test", "testuser", "testpass");
```

For MySQL, the DSN is a string that indicates the database driver (`mysql`), and optionally the hostname where the server is running and the name of the database to use. Typical syntax for the DSN looks like this:

```
mysql:host=host_name;dbname=db_name
```

The default host is `localhost`. If `dbname` is omitted, no default database is selected.

The MySQL driver also recognizes `port` and `unix_socket` parameters, which specify the TCP/IP port number and Unix socket file pathname, respectively. If you use `unix_socket`, do not specify `host` or `port`.

For other database engines, the driver name is different (for example, `pgsql` for PostgreSQL) and the parameters following the colon might be different as well.

When you invoke the `new PDO()` constructor method to connect to your database server, PDO determines from the DSN which type of database engine you want to use and acesses the low-level driver appropriate for that engine. This is similar to the way that Perl or Ruby DBI scripts reference only the top-level DBI module; the `connect()` method provided by the top-level module looks at the DSN and determines which particular lower-level driver to use.

If `new PDO()` fails, PHP throws an exception. Otherwise, the constructor method returns an object of the `PDO` class. This object is a database handle that you use for interacting with the database server until you close the connection.

An alternative to putting the connection code directly in your script is to move it into a separate file that you reference from your main script. For example, you could create a file *pdo_testdb_connect.php* that looks like this:

```
<?php
# pdo_testdb_connect.php - function for connecting to the "test" database

function testdb_connect ()
{
  $dbh = new PDO("mysql:host=localhost;dbname=test", "testuser", "testpass");
  return ($dbh);
}
?>
```

Then include the file into your main script and call `testdb_connect()` to connect and obtain the database handle:

```
require_once "pdo_testdb_connect.php";

$dbh = testdb_connect ();
```

This approach makes it easier to use the same connection parameters in several different scripts without writing the values literally into every script; if you need to change a parameter later, just change *pdo_testdb_connect.php*. Using a separate file also enables you to move the code that contains the connection parameters outside of the web server's document tree. That has the benefit of preventing it from being displayed literally if the server becomes misconfigured and starts serving PHP scripts as plain text.

Any of the PHP file-inclusion statements can be used, such as `include` or `require`, but `require_once` prevents errors from occurring if any other files that your script uses also reference *pdo_testdb_connect.php*.

When you're done using the connection, close it by setting the database handle to `NULL`:

```
$dbh = NULL;
```

After that, `$dbh` becomes invalid as a database handle and can no longer be used as such.

If you do not close the connection explicitly, PHP does so when the script terminates.

While the database handle is open and you are using it to issue other PDO calls, you should arrange to handle errors if they occur. You can check for an error after each PDO call, or you can cause exceptions to be thrown. The latter approach is simpler because you need not check for errors explicitly; any error raises an exception that terminates your script. If you enable exceptions, you also have the option of catching them yourself instead of permitting them to terminate your script. By doing this, you can substitute your own error messages for the defaults, perform cleanup operations, and so on.

To enable exceptions, set the PDO error mode as follows after connecting:

```
$dbh->setAttribute (PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
```

That statement is something you could add to the `testdb_connect()` function if you want the error mode to be set automatically whenever you connect.

For more information on dealing with errors, see "Handling Errors."

Issuing Statements

After obtaining a database handle by calling `new PDO()`, use it to execute SQL statements:

- For statements that modify rows and produce no result set, pass the statement string to the database handle `exec()` method, which executes the statement and returns an affected-rows count:

  ```
  $count = $dbh->exec ("some SQL statement");
  ```

- For statements that select rows and produce a result set, invoke the database handle `query()` method, which executes the statement and returns an object of the `PDOStatement` class:

  ```
  $sth = $dbh->query ("some SQL statement");
  ```

  This object is a statement handle that provides access to the result set. It enables you to fetch the result set rows and obtain metadata about them, such as the number of columns.

To illustrate how to handle various types of statements, the following discussion shows how to create and populate a table using `CREATE TABLE` and `INSERT` (statements that return no result set). Then it uses `SELECT` to generate a result set.

Issuing Statements That Return No Result Set

The following code uses the database handle `exec()` method to issue a statement that creates a simple table `animal` with two columns, `name` and `category`:

```
$dbh->exec ("CREATE TABLE animal (name CHAR(40), category CHAR(40))");
```

After the table has been created, it can be populated. The following example invokes the `exec()` method to issue an `INSERT` statement that loads a small data set into the `animal` table:

```
$count = $dbh->exec ("INSERT INTO animal (name, category)
                      VALUES
                        ('snake', 'reptile'),
                        ('frog', 'amphibian'),
                        ('tuna', 'fish'),
                        ('racoon', 'mammal')");
```

`exec()` returns a count to indicate how many rows were affected by the statement. For the preceding `INSERT` statement, the affected-rows count is 4.

Issuing Statements That Return a Result Set

Now that the table exists and contains a few records, `SELECT` can be used to retrieve rows from it. To issue statements that return a result set, use the database handle `query()` method:

```
$sth = $dbh->query ("SELECT name, category FROM animal");
printf ("Number of columns in result set: %d\n", $sth->columnCount ());
$count = 0;
while ($row = $sth->fetch ())
{
  printf ("Name: %s, Category: %s\n", $row[0], $row[1]);
  $count++;
}
printf ("Number of rows in result set: %d\n", $count);
```

A successful `query()` call returns a `PDOStatement` statement-handle object that is used for all operations on the result set. Some of the information available from a `PDOStatement` object includes the row contents and the number of columns in the result set:

- The `fetch()` method returns each row in succession, or `FALSE` when there are no more rows.
- The `columnCount()` methods returns the number of columns in the result set.

Note: A statement handle also has a `rowCount()` method, but for statements that return a result set, it cannot be assumed to reliably return the number of rows. Instead, fetch the rows and count them, as shown in the preceding example.

Other Ways To Fetch Result Set Rows

`fetch()` accepts an optional fetch-mode argument indicating what type of value to return. This section describes some common mode values. Assume in each case that the following query has just been issued to produce a result set:

```
$sth = $dbh->query ("SELECT name, category FROM animal");
```

- `PDO::FETCH_NUM`

  Return each row of the result set as an array containing elements that correspond to the columns named in the `SELECT` statement and that are accessed by numeric indices beginning at 0:

```
while ($row = $sth->fetch (PDO::FETCH_NUM))
  printf ("Name: %s, Category: %s\n", $row[0], $row[1]);
```

- `PDO::FETCH_ASSOC`

  Return each row as an array containing elements that are accessed by column name:

  ```
  while ($row = $sth->fetch (PDO::FETCH_ASSOC))
    printf ("Name: %s, Category: %s\n", $row["name"], $row["category"]);
  ```

- `PDO::FETCH_BOTH`

  Return each row as an array containing elements that can be accessed either by numeric index or by column name:

  ```
  while ($row = $sth->fetch (PDO::FETCH_BOTH))
  {
    printf ("Name: %s, Category: %s\n", $row[0], $row[1]);
    printf ("Name: %s, Category: %s\n", $row["name"], $row["category"]);
  }
  ```

- `PDO::FETCH_OBJ`

  Return each row as an object. In this case, you access column values as object properties that have the same names as columns in the result set:

  ```
  while ($row = $sth->fetch (PDO::FETCH_OBJ))
    printf ("Name: %s, Category: %s\n", $row->name, $row->category);
  ```

If you invoke `fetch()` with no argument, the default fetch mode is `PDO::FETCH_BOTH` unless you change the default before fetching the rows:

- The `query()` method accepts an optional fetch-mode argument following the statement string:

  ```
  $sth = $dbh->query ("SELECT name, category FROM animal", PDO::FETCH_OBJ);
  while ($row = $sth->fetch ())
    printf ("Name: %s, Category: %s\n", $row->name, $row->category);
  ```

- Statement handles have a `setFetchMode()` method to set the mode for subsequent `fetch()` calls:

  ```
  $sth->setFetchMode (PDO::FETCH_OBJ);
  while ($row = $sth->fetch ())
    printf ("Name: %s, Category: %s\n", $row->name, $row->category);
  ```

Another way to fetch results is to bind variables to the result set columns with `bindColumn()`. Then you fetch each row using the `PDO::FETCH_BOUND` fetch mode. PDO stores the column values in the variables, and `fetch()` returns `TRUE` instead of a row value while rows remain in the result set:

```
$sth = $dbh->query ("SELECT name, category FROM animal");
$sth->bindColumn (1, $name);
$sth->bindColumn (2, $category);
while ($sth->fetch (PDO::FETCH_BOUND))
  printf ("Name: %s, Category: %s\n", $name, $category);
```

Using Prepared Statements

`exec()` and `query()` are `PDO` object methods: You use them with a database handle and they execute a statement immediately and return its result. It is also possible to prepare a statement for execution without executing it immediately. The `prepare()` method takes an SQL statement as its argument and returns a `PDOStatement` statement-handle object. The statement handle has an `execute()` method that executes the statement:

```
$sth = $dbh->prepare ($stmt);
$sth->execute ();
```

Following the `execute()` call, other statement-handle methods provide information about the statement result:

- For a statement that modifies rows, invoke `rowCount()` to get the rows-affected count:

```
$sth = $dbh->prepare ("DELETE FROM animal WHERE category = 'mammal'");
$sth->execute ();
printf ("Number of rows affected: %d\n", $sth->rowCount ());
```

- For a statement that produces a result set, the `fetch()` method retrieves them and the `columnCount()` method indicates how many columns there are. To determine how many rows there are, count them as you fetch them. (As mentioned previously, `rowCount()` returns a row count, but should be used only for statements that modify rows.)

```
$sth = $dbh->prepare ("SELECT name, category FROM animal");
$sth->execute ();
printf ("Number of columns in result set: %d\n", $sth->columnCount ());
$count = 0;
while ($row = $sth->fetch ())
{
  printf ("Name: %s, Category: %s\n", $row[0], $row[1]);
  $count++;
}
printf ("Number of rows in result set: %d\n", $count);
```

If you are not sure whether a given SQL statement modifies or returns nows, the statement handle itself enables you to determine the proper mode of processing. See "Determining the Type of a Statement."

As just shown, prepared statements appear to offer no advantage over `exec()` and `query()` because using them introduces an extra step into statement processing. But there are indeed some benefits to them:

- Prepared statements can be parameterized with placeholders that indicate where data values should appear. You can bind specific values to these placeholders and PDO takes care of any quoting or escaping issues for values that contain special characters. "Placeholders and Quoting" discusses these topics further.

- Separating statement preparation from execution can be more efficient for statements to be executed multiple times because the preparation phase need be done only once. For example, if you need to insert a bunch of rows, you can prepare an `INSERT` statement once and then execute it repeatedly, binding successive row values to it for each execution.

Placeholders and Quoting

A prepared statement can contain placeholders to indicate where data values should appear. After you prepare the statement, bind specific values to the placeholders (either before or at statement-execution time), and PDO substitutes the values into the statement before sending it to the database server.

PDO supports named and positional placeholders:

- A named placeholder consists of a name preceded by a colon. After you prepare the statement, use `bindValue()` to provide a value for each placeholder, and then execute the statement. To insert another row, bind new values to the placeholders and invoke `execute()` again:

```
$sth = $dbh->prepare ("INSERT INTO animal (name, category)
                       VALUES (:name, :cat)");
$sth->bindValue (":name", "ant");
$sth->bindValue (":cat", "insect");
$sth->execute ();
$sth->bindValue (":name", "snail");
$sth->bindValue (":cat", "gastropod");
$sth->execute ();
```

As an alternative to binding the data values before calling `execute()`, you can pass the values directly to `execute()` using an array that associates placeholder names with the values:

```
$sth->execute (array (":name" => "black widow", ":cat" => "spider"));
```

- Positional placeholders are characters within the statement string. You can bind the values prior to calling `execute()`, similar to the previous example, or pass an array of values directly to `execute()`:

```
$sth = $dbh->prepare ("INSERT INTO animal (name, category)
                       VALUES (?, ?)");
# use bindValue() to bind data values
$sth->bindValue (1, "ant");
$sth->bindValue (2, "insect");
$sth->execute ();
# pass values directly to execute() as an array
$sth->execute (array ("snail", "gastropod"));
```

Positional placeholder numbers begin with 1.

An alternative to `bindValue()` is `bindParam()`, which adds a level of indirection to value-binding. Instead of passing a data value as the second argument to `bindParam()`, pass a variable to associate the variable with the placeholder. To supply a value for the placeholder, assign a value to the variable:

```
$sth = $dbh->prepare ("INSERT INTO animal (name, category)
                       VALUES (?, ?)");
$sth->bindParam (1, $name);
$sth->bindParam (2, $category);
$name = "ant";
$category = "insect";
$sth->execute ();
$name = "snail";
$category = "gastropod";
$sth->execute ();
```

The preceding examples use `INSERT` statements, but placeholder techniques are applicable to any type of statement, such as `UPDATE` or `SELECT`.

One of the benefits of using placeholders is that PDO handles any quoting or escaping of special characters or `NULL` values. For example, if you bind the string `"a'b'c"` to a placeholder, PDO inserts `"'a\'b\'c'"` into the statement. To bind the SQL `NULL` value to a placeholder, bind the PHP `NULL` value. In this case, PDO inserts the word "NULL" into the statement without surrounding quotes. (Were quotes to be added, the value inserted into the statement would be the string `"'NULL'"`, which is incorrect.)

PDO also provides a database handle `quote()` method to which you can pass a string and receive back a quoted string with special characters escaped. However, I find this method deficient. For example, if you pass it `NULL`, it returns an empty string, which if inserted into a statement string does not correspond to the SQL `NULL` value. Use `quote()` with care if you use it.

Determining the Type of a Statement

When you issue a statement using a database handle, you must know whether the statement modifies rows or produces a result set, so that you can invoke whichever of `exec()` or `query()` is appropriate. However, under certain circumstances, you might not know the statement type, such as when you write a script to execute arbitrary statements that it reads from a file. To handle such cases, use `prepare()` with the database handle to get a statement handle and `execute()` to execute the statement. Then check the statement's column count:

- If `columnCount()` is zero, the statement did not produce a result set. Instead, it modified rows and you can invoke `rowCount()` to determine the number of affected rows.

- If `columnCount()` is greater than zero, the statement produced a result set and you can fetch the rows. To determine how many rows there are, count them as you fetch them.

The following example determines whether a statement modifies rows or produces a result set, and then processes it accordingly:

```
$sth = $dbh->prepare ($stmt);
$sth->execute ();
if ($sth->columnCount () == 0)
{
  # there is no result set, so the statement modifies rows
  printf ("Number of rows affected: %d\n", $sth->rowCount ());
}
else
{
  # there is a result set
  printf ("Number of columns in result set: %d\n", $sth->columnCount ());
  $count = 0;
  while ($row = $sth->fetch (PDO::FETCH_NUM))
  {
    # display column values separated by commas
    print (join (", ", $row) . "\n");
    $count++;
  }
  printf ("Number of rows in result set: %d\n", $count);
}
```

Handling Errors

When you invoke `new PDO()` to create a database handle, occurrance of an error causes a `PDOException` to be thrown. If you don't catch the exception, PHP terminates your script. To handle the exception yourself, use a `try` block to perform the connection attempt and a `catch` block to catch any error that occurs:

```
try
{
  $dbh = new PDO("mysql:host=localhost;dbname=test", "testuser", "testpass");
}
catch (PDOException $e)
{
  print ("Could not connect to server.\n");
  print ("getMessage(): " . $e->getMessage () . "\n");
}
```

A `PDOException` is an extension of the PHP `Exception` class, so it has `getCode()` and `getMessage()` methods that return an error code and descriptive message, respectively. (However, I find that `getCode()` always returns 0 for connection errors and is meaningful only for PDO exceptions that occur after the connection has been established.)

After you successfully obtain a database handle, PDO handles subsequent calls that use it according to the PDO error mode. There are three modes:

- `PDO::ERRMODE_SILENT`

  When an error occurs in silent or warning mode for a given object method, PDO sets up error information that you can access when the method returns. This is the default error mode.

- `PDO::ERRMODE_WARNING`

  This is like silent mode but PDO also displays a warning message in addition to setting up error information when an error occurs.

- `PDO::ERRMODE_EXCEPTION`

  PDO sets up error information when an error occurs and throws a `PDOException`.

PDO sets error information for the object to which the error applies, regardless of the error mode. This information is available via the object's `errorCode()` and `errorInfo()` methods. `errorCode()` returns an SQLSTATE value (a five-character string). `errorInfo()` returns a three-element array containing the SQLSTATE value, and a driver-specific error code and error message. For MySQL, the driver-specific values are a numeric error code and a descriptive error message.

To handle errors in silent mode, you must check the result of each PDO call. The following example shows how to test for errors during an operation that uses a database handle, `$dbh`, and a statement handle, `$sth` (you would not necessarily print all the available information as the example does):

```
if (!($sth = $dbh->prepare ("INSERT INTO no_such_table")))
{
  print ("Could not prepare statement.\n");
  print ("errorCode: " . $dbh->errorCode () . "\n");
  print ("errorInfo: " . join (", ", $dbh->errorInfo ()) . "\n");
}
else if (!$sth->execute ())
{
  print ("Could not execute statement.\n");
  print ("errorCode: " . $sth->errorCode () . "\n");
  print ("errorInfo: " . join (", ", $sth->errorInfo ()) . "\n");
}
```

Testing the result of every call can become messy quickly. Another way to deal with failures is to set the error handling mode so that any error raises an exception:

```
$dbh->setAttribute (PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
```

In this case, you can assume that if you invoke a method and it returns, it succeeded. You can either leave exceptions uncaught or catch and handle them yourself. If you leave them uncaught, exceptions cause PHP to print a backtrace and terminate your script. To catch exceptions, perform PDO operations using a `try/catch` construct. The `try` block contains the operations and the `catch` block handles an execption if one occurs.

```
try
{
  $sth = $dbh->prepare ("INSERT INTO no_such_table");
  $sth->execute ();
}
catch (PDOException $e)
{
  print ("The statement failed.\n");
  print ("getCode: ". $e->getCode () . "\n");
  print ("getMessage: ". $e->getMessage () . "\n");
}
```

By using `try` and `catch`, you can substitute your own error messages if you like, perform cleanup operations, and so on.

As shown in the preceding example, the `try` block can contain operations on multiple handles. However, if an exception occurs in that case, you won't be able to use the handle-specific `errorCode()` or `errorInfo()` methods in the `catch` block very easily because you won't know which handle caused the error. You'll need to use the information available from the exception methods, as shown.

Using Transactions

In MySQL, some storage engines are transactional (including InnoDB, the default storage engine as of MySQL 5.5). A transactional engine enables you to perform an operation and then commit it permanently if it succeeded or roll it back to cancel its effects if an error occurred. PDO provides a mechanism for performing transactions that is based on the following database-handle methods:

- To start a transaction, invoke `beginTransaction()` to disable autocommit mode so that database changes do not take effect immediately.
- To commit a successful transaction or roll back an unsuccessful one, invoke `commit()` or `rollback()`, respectively.

The easiest way to use these methods is to enable PDO exceptions and use `try` and `catch` to handle errors:

```
$dbh->setAttribute (PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
try
{
  $dbh->beginTransaction ();           # start the transaction
  # ... perform database operation ...
  $dbh->commit ();                     # success
}
catch (PDOException $e)
{
  print ("Transaction failed: " . $e->getMessage () . "\n");
  $dbh->rollback ();                   # failure
}
```

For additional paranoia, you can place the `rollback()` call within a nested `try/catch` construct so that if `rollback()` itself fails and raises another exception, the script doesn't get terminated.

Resources

- The home sites for MySQL and MySQL documentation are:

  ```
  http://www.mysql.com/
  http://dev.mysql.com/doc
  ```

- The home site for PHP is:

  ```
  http://www.php.net/
  ```

- The book *MySQL* discusses at length how to use PDO for PHP programming. See Chapter 9, "Writing Programs using PHP." Appendix I provides a reference for PDO classes and methods. The Web site for this book has sample code for several PDO applications that you can examine:

  ```
  http://www.kitebird.com/mysql-book/
  ```

  (The first and second editions cover the interface provided by the native PHP MySQL functions, the third edition covers the PEAR DB interface. Editions from the fourth up cover PDO.)

- Other documents similar to the one you are reading are available that show how to access MySQL using interfaces for other programming languages:

  ```
  http://www.kitebird.com/articles/
  ```

Revision History

- 1.00—Original version.
- 1.01, 2008-05-07—Removed my mistaken statement that the PDO driver for MySQL requires the `mysqli` extension. It does not. The driver uses `libmysqlclient` directly.
- 1.02, 2013-08-11—Minor revisions and updates.