Writing Scripts with PHP's PEAR DB Module

Paul DuBois
paul@kitebird.com

Document revision: 1.02
Last update: 2005-12-30

As a web programming language, one of PHP's strengths traditionally has been to make it easy to write scripts that access databases so that you can create dynamic web pages that incorporate database content. This is important when you want to provide visitors with information that is always up-to-date, without hand tweaking a lot of static HTML pages. However, although PHP is easy to use, it includes no general-purpose database access interface. Instead it has a number of specialized ones that take the form of separate sets of functions for each database system. There is one set for MySQL, another for InterBase, and another for PostgreSQL—and others as well.

This wide range of support for different database engines help make PHP popular because it means essentially that no matter which database you use, PHP probably supports it. On the other hand, having a different set of functions for each database also makes PHP scripts non-portable at the lexical (source code) level. For example, the function for issuing a SQL statement is named `mysql_query()`, `ibase_query()`, or `pg_exec()`, depending on whether you are using MySQL, InterBase, or PostgreSQL. This necessitates a round of messy script editing to change function names if you want to use your scripts with a different database engine, or if you obtain scripts from someone who doesn't use the same engine you do.

In PHP 4 and up, this problem is addressed by means of a database module included in PEAR (the PHP Extension and Add-on Repository). The PEAR DB module supports database access based on a two-level architecture:

- The top level provides an abstract interface that hides database-specific details and thus is the same for all databases supported by PEAR DB. 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 abstract 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 and MetaBase are two packages that provide an abstract database interface. However, PEAR is included with PHP distributions and installed by default, so if you have a recent version of PHP, you already have PEAR and can begin using it.

PEAR DB uses classes and objects to present an object-oriented interface, and 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.

The PEAR DB architecture implements database support primarily through two files that are used for all database engines, and a third that is chosen on an engine-specific basis:

- The primary ''overseer'' file is *DB.php*. It implements the `DB` class that creates database connection objects, and also contains some utility routines.
- *DB/common.php* implements the `DB_common` class that forms the basis for database access. This file

contains common code that implements default database-access methods and additional utility routines. The default methods are overridden as necessary on a driver-specific basis.

- The third file contains the driver and is selected according to the database you're using. Each driver file has a name like *DB/driver.php* and implements a class named `DB_driver` that inherits the base class `DB_common`. For MySQL, there are actually two drivers, each of which implements support for one of the two PHP extensions, `mysql` and `mysqli` (''MySQL improved''). *DB/mysql.php* implements a class `DB_mysql` that inherits `DB_common` and extends it to provide methods tailored for accessing MySQL servers. This driver accesses the `mysql_xxx()` PHP functions. *DB/mysql.php* is available as of PHP 5. It is similar but implements a class `DB_mysqli` that accesses the `mysqli_xxx()` PHP functions.

You should be able to find these files under the installation directory for your PEAR hierarchy. Typically a script references only *DB.php*, to gain access to the `DB` class. Then, when you invoke the `connect()` method of the `DB` class to connect to your database server, the class determines which type of server you're using and reads in the proper driver file. The driver file in turn pulls in *DB/common.php*. 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 determines which particular lower-level driver you need.

Preliminary Requirements

To use PEAR DB for writing scripts that access MySQL, the following requirements must be satisfied:

- You must have PHP 4 or newer installed. In particular, you should have version 4.0.4 or newer. PEAR does not work with PHP 3, and is not bundled with earlier PHP 4 distributions. PHP includes a program called `pear` that enables you to check which PEAR modules are installed and install missing ones. For example, to list the installed modules, and then install the DB module if it is missing, execute these commands:

  ```
  % pear lib
  % pear install DB
  ```

  You might need to execute the second command as `root`.

  If you install PHP 5, you can use either the original `mysql` PEAR DB driver, or the `mysqli` ''MySQL improved'' driver. `mysqli` can use the binary client/server protocol for prepared statements that was introduced in MySQL 4.1.

- Your version of PHP must include the client library for MySQL or your scripts won't be able to connect to your MySQL server. The client library must be from MySQL 4.1.2 or newer if you want to use the `mysqli` driver.

- The PHP initialization file (*php.ini*) should be set up so that the `include_path` variable specifies the pathname of the directory where the PEAR files are installed. For example, if the PEAR files are installed in */usr/local/lib/php* under Unix, you'd set `include_path` like this:

  ```
  include_path = "/usr/local/lib/php"
  ```

  If the PEAR files are installed for Windows under *C:\php\pear*, you'd set `include_path` like this:

  ```
  include_path = "C:\php\pear"
  ```

  `include_path` can name other directories as well. If you want its value to specify multiple directories, separate the directory pathnames by colons for Unix and by semicolons for Windows.

  Whenever you modify the *php.ini* file, you should restart Apache if you're using PHP as an Apache module (rather than as a standalone program). Otherwise, PHP won't notice the changes.

If you are an end user and do not have permission to modify `include_path`, contract your administrator

and request that the PEAR installation directory be added to `include_path` if necessary.

Writing PEAR DB Scripts

Scripts that use the PEAR DB interface to access MySQL generally perform the following steps:

- Reference the *DB.php* file to gain access to the PEAR DB module.
- Connect to the MySQL server by calling `connect()` to obtain a connection object.
- Use the connection object to issue SQL statements and obtain result objects
- Use the result objects to retrieve information returned by the statements.
- Disconnect from the server when the connection object is no longer needed.

The next sections discuss this process in more detail.

Referencing the PEAR DB Source

Before using any PEAR DB calls, your script must pull in the *DB.php* file. Assuming that your `include_path` setting names the PEAR installation directory, you can refer to the file like this:

```
require_once "DB.php";
```

Any of the 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 *DB.php*.

Connecting to the MySQL Server

To establish a connection to the MySQL server, you must specify a data source name (DSN) containing connection parameters. The DSN is a URL-style string that indicates the database driver (which is `mysql` for the MySQL extension or `mysqli` for the ''MySQL improved'' extension), the hostname where the server is running, the user name and password for your MySQL account, and the name of the database you want to use. Typical syntax for the DSN looks like this:

```
mysqli://user_name:password@host_name/db_name
```

This article uses the `mysqli` driver for examples. Substitute `mysql` for `mysqli` if you want to use the older `mysql` driver instead.

The DSN is passed to the `connect()` method of the `DB` class. For example, to connect to the MySQL server on the local host to access the `test` database with a user name and password of `testuser` and `testpass`, the connection sequence can be written like this:

```
$dsn = "mysqli://testuser:testpass@localhost/test";
$conn =& DB::connect ($dsn);
if (DB::isError ($conn))
  die ("Cannot connect: " . $conn->getMessage () . "\n");
```

If `connect()` fails, `$conn` refers to an error object that you can use for printing a message before exiting. If `connect()` succeeds, `$conn` refers to a connection object you can use for issuing statements until you close the connection. Be sure to check the result of the `connect()` call. If the return value represents an error and you try to use it to issue statements, you'll just get more errors. (An alternative approach to error handling is to tell PEAR to terminate your script automatically when a PEAR error occurs. This is discussed in ''More on Error Handling.'')

Another way to specify connection parameters is to put them in a separate file that you reference from your main script. For example, you can create a file *testdb_params.php* that looks like this:

```
<?php
# parameters for connecting to the "test" database
$driver = "mysqli";
$user = "testuser";
$password = "testpass";
$host = "localhost";
$db = "test";
# DSN constructed from parameters
$dsn = "mysqli://testuser:testpass@localhost/test";
?>
```

Then you can include the file into your main script and use the `$dsn` variable constructed from the connection parameter variables like this:

```
require_once "testdb_params.php";

$conn =& DB::connect ($dsn);
if (DB::isError ($conn))
  die ("Cannot connect: " . $conn->getMessage () . "\n");
```

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 sometime, just change *testdb_params.php*. It also enables you to move the parameter file outside of the web server's document tree, which prevents its contents from being displayed literally if the server becomes misconfigured and starts serving PHP scripts as plain text.

Issuing Statements

After obtaining a connection object by calling `connect()`, you can use it to issue SQL statements by passing a statement string to the object's `query()` method:

```
$stmt = "some SQL statement";
$result =& $conn->query ($stmt);
```

The return value, `$result`, can take three forms:

- If an error occurs, `DB::isError($result)` will be true. When this happens, you do nothing further with `$result` other than perhaps to use it for printing an error message.
- If the string is a statement such as `INSERT` or `UPDATE` that manipulates data rather than returning a result set, `$result` will be `DB_OK` for success. In this case, you can call `$conn->affectedRows()` to find out how many rows the statement changed.
- If the string is a statement such as `SELECT` that generates a result set (a set of rows), `$result` is a result set object when the statement succeeeds. You can use the object to determine the number of rows and columns in the result set and to fetch the rows. When you're done with the result set, dispose of it by calling `$result->free()`.

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. In general, the statement processing sequence is as follows:

- Invoke `query()` to execute the statement.
- Check whether the statement succeeded.
- If the statement failed, report the error.

- If the statement succeeded, retrieve any additional information you expect to receive, such as a row count or the contents of any rows returned.

Issuing Statements That Return No Result Set

The following code issues a statement to create a simple table `animal` with two columns, `name` and `category`:

```
$result =& $conn->query ("CREATE TABLE animal
                          (name CHAR(40), category CHAR(40))");
if (DB::isError ($result))
  die ("CREATE TABLE failed: " . $result->getMessage () . "\n");
```

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

```
$result =& $conn->query ("INSERT INTO animal (name, category)
                          VALUES
                            ('snake', 'reptile'),
                            ('frog', 'amphibian'),
                            ('tuna', 'fish'),
                            ('racoon', 'mammal')");
if (DB::isError ($result))
  die ("INSERT failed: " . $result->getMessage () . "\n");
```

To determine how many rows were affected by a successful data-manipulation statement, use the `affectedRows()` method of your connection object:

```
printf ("Number of rows inserted: %d\n", $conn->affectedRows ());
```

For the preceding `INSERT` statement, `affectedRows()` returns the value 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, as shown below:

```
$result =& $conn->query ("SELECT name, category FROM animal");
if (DB::isError ($result))
  die ("SELECT failed: " . $result->getMessage () . "\n");
printf ("Result set contains %d rows and %d columns\n",
        $result->numRows (), $result->numCols ());
while ($row =& $result->fetchRow ())
  printf ("%s, %s\n", $row[0], $row[1]);
$result->free ();
```

A successful `query()` call returns an object, `$result`, that is used for all operations on the result set. Information that is available from `$result` includes the row and column count of the result set, and the contents of those rows. When you no longer need the result set, dispose of it by calling `free()`. After that, `$result` becomes invalid and no longer can be used to access the result set.

Other Ways To Fetch Result Set Rows

`fetchRow()` accepts an optional argument indicating what type of value to return. By default, `fetchRow()` returns the next 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. This behavior is the same as if you had invoked `fetchRow()` with a `DB_FETCHMODE_ORDERED` argument:

```
$row =& $result->fetchRow (DB_FETCHMODE_ORDERED);
```

`fetchRow()` can return an associative array instead, which enables you to refer to array elements by

column name. To invoke `fetchRow()` this way, pass it an argument of `DB_FETCHMODE_ASSOC`:

```
while ($row =& $result->fetchRow (DB_FETCHMODE_ASSOC))
  printf ("%s, %s\n", $row["name"], $row["category"]);
```

To fetch rows as objects, use the `DB_FETCHMODE_OBJECT` mode. In this case, you access column values as object properties:

```
while ($obj =& $result->fetchRow (DB_FETCHMODE_OBJECT))
  printf ("%s, %s\n", $obj->name, $obj->category);
```

If you find yourself overriding the default fetch mode by passing an argument to `fetchRow()` each time you invoke it, you may want to reset the default mode by calling `setFetchMode()`. For example, rather than doing this:

```
$result =& $conn->query ($stmt1);
while ($row =& $result->fetchRow (DB_FETCHMODE_ASSOC)) ...
...
$result =& $conn->query ($stmt2);
while ($row =& $result->fetchRow (DB_FETCHMODE_ASSOC)) ...
...
etc.
```

You can do this instead:

```
$conn->setFetchMode (DB_FETCHMODE_ASSOC);
$result =& $conn->query ($stmt1);
while ($row =& $result->fetchRow ()) ...
...
$result =& $conn->query ($stmt2);
while ($row =& $result->fetchRow ()) ...
...
etc.
```

Determining the Type of a Statement

Typically when you issue a statement, you'll know whether to expect a result set from it. However, under certain circumstances, this may not be true, such as when you write a script to execute arbitrary statements that it reads from a file. To determine whether a statement returns a result set so that you can process it properly, check the statement string with `isManip()`. This is a `DB` class method that returns true if the statement manipulates (changes) rows, and false if it retrieves rows:

```
if (DB::isManip ($stmt))
{
  # the statement manipulates data; no result set is expected
}
else
{
  # the statement retrieves data; a result set is expected
}
```

Disconnecting from the Server

When you're done using the connection, close it:

```
$conn->disconnect ();
```

After invoking `disconnect()`, `$conn` becomes invalid as a connection object and can no longer be used as such.

More on Error Handling

PEAR offers script writers control over the handling of PEAR errors. By default, PEAR calls return error objects. This enables you to do whatever you want with the error information (such as printing an error message), but on the other hand puts the burden on you to check the result of each call. Other approaches are possible. For example, if you don't want to test the result of every call, you can set the error handling mode to `PEAR_ERROR_DIE` to cause PEAR to print an error message and terminate the script automatically if an error occurs. To do this for your connection object, call `setErrorHandling()` as follows:

```
$conn->setErrorHandling (PEAR_ERROR_DIE);
```

After setting the error mode this way, any error that occurs when you issue a statement causes your script to exit. (In other words, you can assume that if `$conn->query()` returns, it succeeded and you need not test the result.) Note that `setErrorHandling()` is invoked here as a connection object method, so you can't call it until you have a valid connection—which means of course that you can't use it to trap errors that occur while attempting to connect. If you want to trap all PEAR errors, including those from `connect()` calls that fail, invoke `setErrorHandling()` as a `PEAR` class method instead:

```
PEAR::setErrorHandling (PEAR_ERROR_DIE);
```

Caveats

To port a PEAR DB script for use with a different database, ideally you just change the DSN string passed to the `connect()` call. For example, if you have a MySQL script that you want to use with PostgreSQL, change the DSN string to the format expected by the PostgreSQL driver. However, PEAR DB won't help make non-portable SQL work with other database engines. For example, if you use MySQL's `AUTO_INCREMENT` feature to generate sequence numbers, the SQL syntax for doing that is not portable to other databases. Sometimes you can use PEAR-level constructs to avoid SQL-level non-portabilities. For the case of sequence number generation, the PEAR DB module provides a facility for generating sequence numbers that makes no reference to SQL at all. The underlying implementation details are hidden in the drivers. (The implementation uses an `AUTO_INCREMENT` column for MySQL, but that is not seen by the script writer.)

Adding an abstraction layer on top of the engine-specific functions hides database-dependent details so that a uniform interface can be presented to script writers. This simplifies the interface for script writers, but at the same time adds complexity to the underlying implementation. The result is that using PEAR DB is slightly less efficient than calling the native database access functions directly.

Resources

The home site for MySQL is:

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

The home sites for PHP and PEAR are:

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

At *pear.php.net/support.php*, you can sign up for PEAR mailing lists and find links to other PEAR articles.

The book *MySQL* (third edition) discusses at length how to use PEAR DB for PHP programming. See Chapter 8, ''Writing Programs using PHP.'' Appendix I provides a reference for PEAR DB classes and methods. The Web site for this book has sample code for several PEAR DB applications that you can examine:

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

(Note: The first and second editions cover the native PHP MySQL functions, not PEAR DB.)

Revision History

- 1.00—Original version.
- 1.01, 2003-01-24—Minor revisions.
- 1.02, 2005-12-30—Add information about `mysqli`. Remove extraneous code from examples.

Acknowledgment

The original version of this article was written for NuSphere Corporation.