Using MySQL with PDO

Peter Lavin

revised July 25, 2011 (originally published on the MySQL site)

Table of Contents

Prerequisites for Using PDO

PDO is a PHP extension providing a data-access abstraction layer that can be used with a variety of databases. This gives you the flexibility of changing the database backend without having to alter your access methods. Even if you use MySQL exclusively, PDO can provide advantages; you can use the same data-access methods regardless of the MySQL version. This does away with the need for using the standard MySQL extension with older versions of MySQL and using the MySQLi extension for later versions. An additional advantage is the ability to use object-oriented code regardless of the version of MySQL.

PDO requires the object-oriented capabilities of PHP 5, so PHP 5.0 or higher is a prerequisite. The PDO extension ships with binary versions of PHP 5.1 and 5.2 and is very simple to implement on most operating systems.

Compiling PHP from source is the one sure way to customize PHP to your exact specifications and ensure that you have not only PDO but also the drivers you need. However, the package managers of most current Linux distributions make it easy to add support—if it's not already there.

Note

Some earlier versions of Mac OS X do not include the PDO driver.

The PDO extension is entirely object-oriented—there is no procedural version of this extension so some knowledge of object-oriented programming is assumed.

Project Outline

This article examines a project that uses PDO to select from and insert into a MySQL database of PHP classes. The PDO classes are as follows:

- `PDO` – the PDO connection object

- `PDOStatement` – the PDO statement object returned by the connection `query` method or the `prepare` method
- `PDOException` – the PDO-specific exception object
- `PDORow` – a representation of a result set row as an object

All PDO classes are used with the exception of the `PDORow` class.

XML representations of PHP classes, both internal and user-defined, are transformed into SQL INSERT statements and added to a MySQL database. This is done using the SimpleXMLElement class, one of the built-in PHP classes. Even if your grasp of XML is elementary, you'll find the SimpleXMLElement easy to understand and use.

The attached compressed file contains all the code and preserves the required directory structure. Find that file here. Any user-defined classes are found in the `classes` directory and XML files are found in the `xml` directory. The PHP scripts are found at the root directory.

The XML Document Format

Using PHP's reflection classes you can automate the process of converting a PHP class to XML. We won't concern ourselves here with the details of how to do this; we'll review an example file in this section and include other examples in the `xml` directory. The reflection classes allow you to introspect your own classes or built-in classes. They are useful tools for reverse engineering code and discovering the properties of objects at runtime. More importantly perhaps, they provide a way of generating documentation from source code.

If you haven't used the reflection classes before you can quickly get a sense of their capabilities by executing `Reflection::export( new ReflectionClass( ClassName));`. This static method of the Reflection class dumps all the methods, data members, and constants of the ReflectionClass object passed to it.

The following example is an XML representation of what reflection reveals about the `mysqli_sql_exception` class.

```
<?xml version="1.0"?>
<classobj>
  <name>mysqli_sql_exception</name>
  <documenting_date>2007-07-31</documenting_date>
  <php_version>5.2.0</php_version>
  <type final="" abstract="">class</type>
  <origin mod_date="" num_lines="">internal</origin>
  <parent_class>RuntimeException</parent_class>
  <class_doc_comments/>
  <interfaces_list/>
  <methods_list>
    <method static="0" final="1" abstract="0" declaring_class="Exception" priority="2">
      <method_name>__clone</method_name>
      <method_origin>internal</method_origin>
      <visibility>private</visibility>
      <method_doc_comment/>
    </method>
    <method static="0" final="0" abstract="0" declaring_class="Exception" priority="2">
      <method_name>__construct</method_name>
      <method_origin>internal</method_origin>
      <visibility>public</visibility>
      <param classtype="" defaultvalue="" byreference="" isoptional="1">message</param>
      <param classtype="" defaultvalue="" byreference="" isoptional="1">code</param>
      <method_doc_comment/>
      ...
    </method>
  </methods_list>
  <datamembers_list>
    <datamember visibility="protected" static="0" defaultvalue="""">
      <datamember_name>message</datamember_name>
      <datamember_doc_comment/>
    </datamember>
    ...
  </datamembers_list>
  <constants_list/>
```

```
</classobj>
```

Not all methods or data members of the `mysqli_sql_exception` class are included but there's enough detail here to form an idea of what any XML representation of a PHP class might look like.

Most of the tags and attributes are self explanatory. For example, by looking at the `message` data member you can readily determine that it is a protected, non-static data member with no default value. Details will become more apparent if you examine the DDL scripts used to create the database tables. Find these statements in the attached compressed file.

Designing the Tables

An XML class file is effectively a flat-file database. To convert this file to a relational database format requires a number of tables. These are:

- Classes (including interfaces)
- Parent interfaces
- Methods
- Method parameters
- Data members
- Constants

To view the SQL needed to create these tables find the `phpclasses.sql` script in the `scripts` directory. You may want to look at this file as each of the tables is discussed in the following sections.

The Classes Table

The most obvious table required is the table of classes and interfaces. Find below the SQL necessary to create this table.

The `name` field uniquely identifies each record so it could be used as a primary key but an AUTO_INCREMENT field is more convenient. Only two modifiers can be applied to a class or interface and these are `final` and `abstract` . (Static classes don't exist in PHP so there's no need for this modifier and all classes are public so there is no need for a visibility specifier.)

The `type` and `origin` fields are candidates for the ENUM data type since, in both cases, there are only two options; object templates are either classes or interfaces and these are either user-defined or built-in. The `last_modified` ,`num_lines` and `class_doc_comment` fields only apply to user-defined classes since reflection cannot capture this information when used with an internal class.

The Parent Interfaces Table

PHP doesn't support multiple inheritance for classes so a simple `parent_class` field is all that's required in the classes table. On the other hand, multiple interfaces can be implemented; hence the need for an interfaces table.

This is a simple table made up of the id number of the implementing class and the interface name.

The Methods Table

Classes can also have any number of methods, requiring a table similar to the classes table.

Like the table of classes, this table supports a `method_origin` field and the modifiers, final and abstract. However, methods can also be static and have a visibility modifier. The `declaring_class` field simply indicates whether a method is inherited as is, overridden in the child, or entirely new.

Unlike some other object-oriented languages, method overloading (in the sense of two methods with the same name but a different number or different types of parameters) is not supported in PHP. For this reason, a primary key may be created from the class id and method name.

The Parameters Table

Methods may have any number of parameters, hence the need for a parameters table.

Parameters may have default values, may be optional, and may be passed by value or by reference. As of PHP 5.1 parameters may also be type-hinted so a `class_type` field is also necessary. Parameter variable names must be unique to the method and method names must be unique to a class so these two fields along with the class id uniquely identify a parameter and form the primary key.

The Data Members and Constants Tables

The data members table incorporates a number of fields common to the other tables and is a simpler version of the methods table.

A table of constants is much simpler than the data members table and is made up of three fields only, the `class_id` , `constant_name` , and its associated value.

Unlike data members, any comments that accompany constants cannot be retrieved so there is no `doc_comment` field for the table of constants.

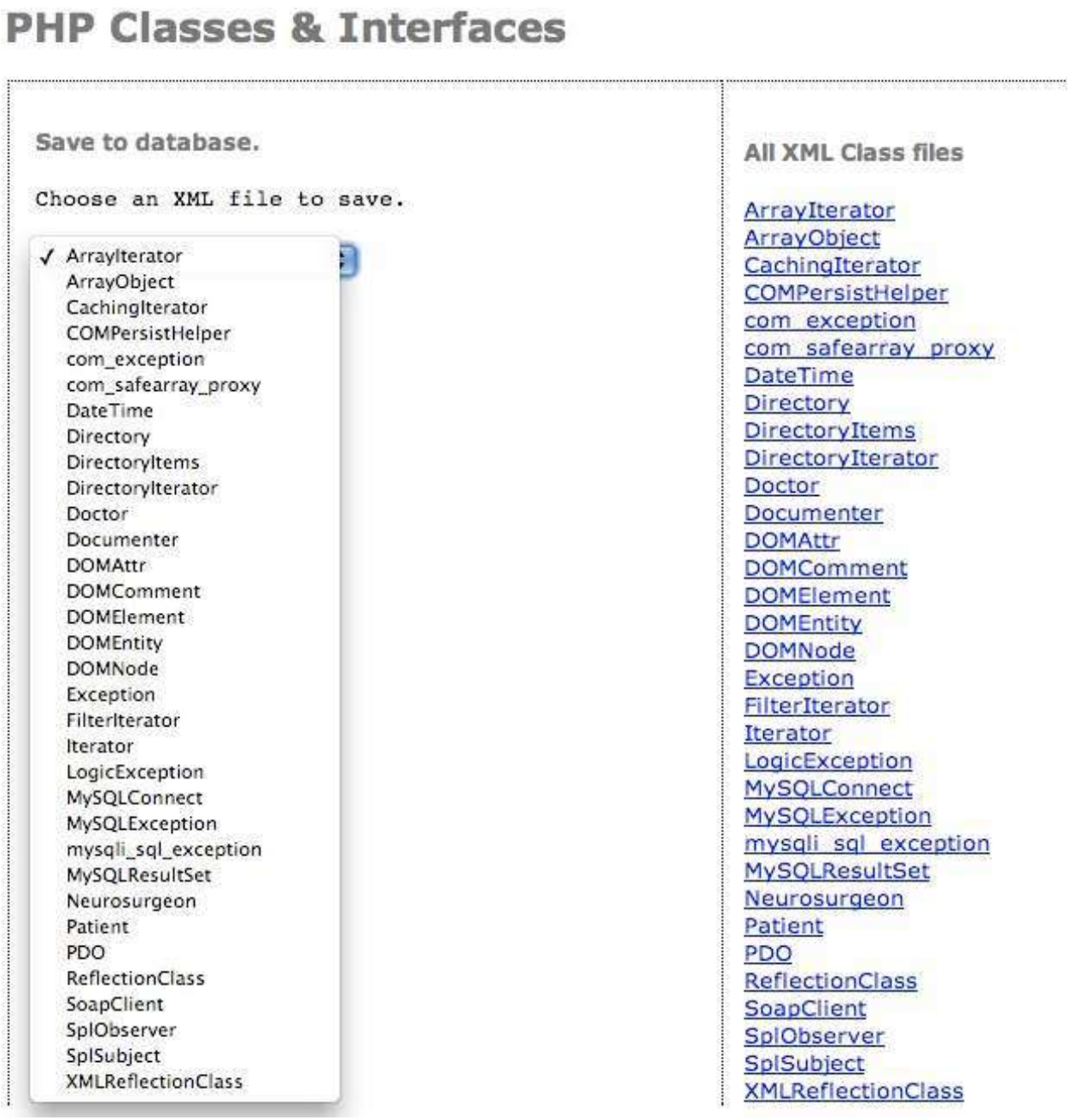
Creating the Tables

The script file `phpclasses.sql` contains the DDL statements for creating the database, `phpclasses`, and the tables. Execute this script from the command line by navigating to the `scripts` directory and executing `mysql -u` *`user_name`* `-p`*`password`* `< phpclasses.sql`.

Creating a PDO Connection

Records are added to a database using a web form. Within this form a drop-down list box is populated with the names of classes that are not already included in the database, thereby eliminating duplicate submissions. The result is pictured below:

Figure 1. Select control



The elements of the list box match the XML files on the right because no XML files have been added to the database yet.

Existing XML class files are found in a directory named `xml`. The XML file name (excluding the `xml` extension) matches the corresponding PHP class name. These names are copied from the `xml` directory into an array using a user-defined class, `DirectoryItems` . This array is compared to the classes already contained in the database to create a drop-down list of classes (the `<select>` control in the following HTML) that aren't yet included. The code to do this follows.

```
<form name= "select_class" action="add_record.php" method="get" style="padding:5px;">
  <select style="max-width: 180px; min-width: 180px;" name = "xmlfilename" >
    <?php
      //autoload user-defined classes
```

```
      function __autoload($class){
          include 'classes/'.$class.'.php';
      }
      include 'connection.php';
      //get names of files in the xml dir
      $di = new DirectoryItems('xml');
      $di->filter('xml');
      $arr = $di->getFilearray();
      //now get array of names from database
      try{
        $db = new PDO("mysql:host=$host;dbname=$database",
            $username, $password);❶
        $db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);❷
        $sql = "SELECT name FROM tblclasses";
        $stmt = $db->query($sql);❸
        //return a simple array of all the names
        $arr2 = $stmt->fetchAll(PDO::FETCH_COLUMN, 0);❹
        $db = NULL;
      }catch(PDOException $e){
        echo $e;
      }
      //select only what is not in the db
      $not_in_db = array_diff( $arr, $arr2);
      natcasesort($not_in_db);
      foreach($not_in_db as $v){
        echo "<option>$v</option>\n";
      }
      ?>
      </select><br /><br />
    <input type="submit" value="submit" />
</form>
```

❶ The PDO connection to the database is created by passing a Data Source Name (DSN) to the PDO constructor. A DSN is made up of the driver name followed by a colon and then driver-specific connection requirements. When using MySQL, you specify `mysql:` followed by the host name, database name, and database credentials. These connection parameters are contained in the `connection.php` file.

Note

If you want to execute the code you must change the username and the password contained in the `connection.php` file. The assumption is that you will be connecting to the database on localhost.

It is also possible to connect to a database by invoking a URI. You could, for example, construct a PDO instance by pointing to a file that contains the DSN string. Do this in the following way: `$db = new PDO("uri:file:///path/to/dsnfile");` .

❷ The `setAttribute` method of the PDO class lets you determine whether you want to raise errors or throw exceptions. Throwing exceptions seems to be the better choice since you can then enclose all your code in a try block and deal with errors in one place. The line `$db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);` ensures the creation of exceptions rather than errors . In order to set the error mode you need to create a connection object first. (The `setAttribute` method performs a number of functions besides setting the error type. It can be used to force column names to upper or lower case and also to set database-specific capabilities such as `MYSQL_ATTR_USE_BUFFERED_QUERY` .)

❸ To execute a database query you need to use the `PDO::query` method to send the statement to the MySQL server and have the statement executed. For our application, we use this method to obtain a list of the rows from the class table, extracting only the name field so that we can generate a list of class names that have already been added to the database.

The `query` method returns a PDOStatement object. There are a number of ways that we can use this object in order to determine all the classes in the database. One possibility is to use the `fetch` method of the PDOStatement class to return each row and then examine that row to determine whether or not the specific class is already in the database. Do this and you can use a `while` statement to iterate over each row returned. However, since PHP has the useful function, `array-diff` , for determining elements that are in one array but not another, capturing the result set as an array seems the better solution.

❹ Our query only retrieves one field so we can create a one dimensional array of all the records returned by using the `fetchAll` method, specifying the mode as `PDO::FETCH_COLUMN` and the target column as the first and only column, `0` . However, using `fetchAll` with large result sets is not recommended as it may tax memory. As there only about 120 PHP classes this is not a concern here.

There is no method for explicitly closing a PDO connection but this can be done by setting the connection object to `NULL` . This isn't strictly necessary since PHP removes references to all objects when a script terminates.

Using the appropriate method to fetch query results makes it easy to populate a drop-down list box with XML files that are not yet included in the database. The next section looks at inserting data from these XML files into a database.

Inserting into a Database

Selecting a class from the list box pictured in Figure 1, "Select control" and submitting it invokes the `add_record.php` script. This script contains the code that inserts records into the database tables. You might want to have a quick look at this file to get an overview of what it does. We won't discuss every line of code, we'll concentrate on the PDO code and only on methods that haven't yet been discussed. An abbreviated version of that script is shown below:

```
try{
  $dsn = "mysql:host=$host;dbname=$database, $username, $password";
  $db = new PDO( $dsn);
  $db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
  ...
  $db->beginTransaction();❶
  $xml = new XMLFormatToSQL("xml/".$xmlfilename. '.xml', $db);
  $sql = $xml->createClassSQL();❷
  $db->exec($sql);
  //retrieve id
  $class_id = $db->lastInsertId();❸
  //get array of interfaces
  $sql = $xml->createInterfacesSQL($class_id);
  //check before executing
  $db->exec($sql);
   ...
   /* For constants return a two-dimensional array
   and use a prepared statement */
   $values_array = $xml->createConstantsArray();❹
   if(isset($values_array)){
   $sql = "INSERT INTO tblconstants VALUES($class_id, ?, ? )";
   $stmt = $db->prepare($sql);❺
   $stmt->bindParam(1, $name, PDO::PARAM_STR);❻
   $stmt->bindParam(2, $value);
   foreach($values_array as $arr){
     list($name, $value) = $arr;
     $stmt->execute();❼
   }
   /*easier way to do it -- no binding
   foreach($values_array as $arr){
     $stmt->execute($arr);
   }*/
   }
  //now can commit
  $db->commit();❽
  $message = "Successfully added.";
}catch(Exception $e)❾{
  $message = $e->getMessage();
  //a PDOException may be thrown even if db not set
  if(isset($db)){
    $db->rollBack();❿
  }
  echo $message;
}
```

This script makes use of an ad hoc class, XMLFormatToSQL. This class uses a SimpleXMLElement object and its methods to generate the SQL necessary to insert records into the `phpclasses` database. The SimpleXML extension is enabled by default so it should be available (unless you've compiled from source and deliberately disabled it).

Simply put, the XMLFormatToSQL class first creates the SQL necessary to create a record in the `tblclasses` table and then, if necessary, creates the related SQL statements for the other tables in the database. To examine the details of this class see the `XMLFormatToSQL.php` file.

❶ The first unfamiliar method used in the `add_record.php` script is the `beginTransaction` method. Using transactions means that changes can be rolled back should errors occur. (However, in some circumstances database commits occur regardless of whether the `beginTransaction` method is used or not—for example, with MySQL, when DDL statements are issued.)

❷ Creating a record in the primary table, `tblclasses` , is done by first creating the necessary SQL statement. This is done using the `XMLFormatToSQL::createClassSQL` method. Note how this method uses the `PDO::quote` method to quote variables. Literals may also be used (see the `XMLFormatToSQL::createMethodsSQL` for example) but `quote` has the advantage of being database-neutral. However, the recommended way for quoting and escaping special characters is to use prepared statements. You'll see how this is done shortly.

The SQL statement returned by `createClassSQL` is passed to the `exec` method of the PDO class. In the previous section, when issuing a SELECT statement, we used the `query` method of the PDO class. `exec` is used with SQL statements that do not return result sets; it returns the number of rows affected by the SQL statement; in this case only one record is created.

❸ The other tables in the database are dependent on the id created in the `tblclasses` table. To retrieve this id use the `lastInsertId` method exactly as you would the `mysql_insert_id` function.

Records in the other tables are created in the came way as the record in the `tblclasses` table, the only difference being that an array of multiple insert statements is returned.

❹ However, the approach taken to creating the entries in the constants table differs. The `createConstantsArray` method of the `XMLFormatToSQL` class returns a two-dimensional array of the values that need inserting rather than a series of SQL statements. Using a prepared statement is the ideal solution when working with arrays of values especially when the same statement needs to be executed a number of times. Using prepared statements has a couple of advantages; it not only improves efficiency — only the parameters need to be sent to the server repeatedly — but it also improves security by automatically quoting variables. There is no need to use the `PDO::quote` method as shown earlier.

❺ To create a prepared statement, construct an SQL statement with replaceable parameters indicated by question marks. Pass this statement as a parameter to the `PDO::prepare` method. Use the returned `PDOStatement` object to bind parameters. Binding can be more or less fine-grained. You must pass a parameter identifier and the associated variable but you can also optionally pass the data type, length of the data type, and other driver-specific options.

❻ The first prepared statement parameter, the name of the constant, is always a string so it make sense to also specify the data type, `PDO::PARAM_STR` . The length of this parameter will vary so specifying the length doesn't make sense. The second parameter to the prepared statement is the value of the constant. This can be either an integer or a string so the data type cannot be specified when this parameter is bound.

❼ There are a number of different ways of using the `PDOStatement` class. You do not need to bind parameters. The code commented out in the preceding listing shows that an array of values can be passed directly to the `PDOStatement::execute` method, bypassing the need to bind parameters. It is also possible to use names rather than question marks as place holders for the replaceable parameters in an SQL statement.

❽ The `commit` method of the PDO object, ends the transaction and commits all the changes. Use of transactions means that the creation of a class record and all its related records in other tables is an atomic operation.

❾ There is only one `catch` block. Because a PDO exception is derived from the `Exception` class, this block catches any kind of exception. Happily, should you pass an incorrect DSN to the constructor, a PDOException is thrown even though no PDO object is created. (Hence the need to test the `$db` variable before calling the `rollBack` method.)

Using catch blocks is much less tedious than error trapping and this is a capability of PDO that you can use regardless of the MySQL server version you are using.

❿ Should an exception be thrown any changes already made can be rolled back using the `rollBack` method. If there is a failure at any point, no records will be added to any tables. Do this and you avoid the possibility of storing incomplete information in the database. You will know that you have all the elements of a class or none of them, saving what could perhaps be a messy clean-up job if only some statements fail and others are committed.

Conclusion

PDO is a data-access abstraction layer providing uniform, object-oriented methods for accessing different databases or different versions of the same database. For this reason, PDO makes migrating to different databases easy; in some cases requiring only that the DSN be updated. For example, an SQLite DSN requires only the driver name and a path to the database — `"sqlite:mydb.sqlite"` . Changing this to `"mysql:host=$host;dbname=$database,$username, $password"` may be all you need

to do to change the database backend. However, any non-standard SQL statements also need to be changed. For instance, any statements that used the SQLite string concatenation operator, `||` , would need to be changed to use the MySQL function, `CONCAT()`.

Using PDO has many advantages but there are some things that it can't help you with. You can't use SQL statements that are unsupported by the underlying database. You cannot, for instance successfully issue a `CREATE TRIGGER` statement against a MySQL 4.1 database.

About the Author

Peter Lavin has been published in a number of print and online magazines. He is also the author of Object Oriented PHP, published by No Starch Press and a contributor to PHP Hacks by O'Reilly Media.

Being a full-time technical writer, Peter occasionally feels the need to depart from the restrained style typical of his profession by writing articles with code snippets that use background colours other than grey.