PDO Tutorial for MySQL Developers

From Hashphp.org

Contents

Why use PDO?

mysql_* functions are getting old. For a long time now mysql_* has been at odds with other common SQL database programming interfaces. It doesn't support modern SQL database concepts such as prepared statements, stored procs, transactions etc... and it's method for escaping parameters with `mysql_real_escape_string` and concatenating into SQL strings is error prone and old fashioned. The other issue with mysql_* is that it has had a lack of attention lately from developers, it is not being maintained... which could mean things like security vulnerabilities are not getting fixed, or it may stop working altogether with newer versions of MySQL. Also lately the PHP community have seen fit to start a soft deprecation of mysql_* which means you will start seeing a slow process of eventually removing mysql_* functions altogether from the language (Don't worry this will probably be awhile before it actually happens!).

PDO has a much nicer interface, you will end up being more productive, and write safer and cleaner code. PDO also has different drivers for different SQL database vendors which will allow you to easily use other vendors without having to relearn a different interface. (though you will have to learn slightly different SQL probably). Instead of concatenating escaped strings into SQL, in PDO you bind parameters which is an easier and cleaner way of securing queries. Binding parameters also allow for a performance increase when calling the same SQL query many times with slightly different parameters. PDO also has multiple methods of error handling. The biggest issue I have seen with mysql_* code is that it lacks consistent handling, or no handling at all! With PDO in exception mode, you can get consistent error handling which will end up saving you loads of time tracking down issues.

PDO is enabled by default in PHP installations now, however you need two extensions to be able to use PDO: PDO, and a driver for the database you want to use like pdo_mysql. Installing the MySQL driver is as simple as installing the php-mysql package in most distributions.

Connecting to MySQL

old way:

```
<?php
$link = mysql_connect('localhost', 'user', 'pass');
mysql_select_db('testdb', $link);
mysql_set_charset('UTF-8', $link);
```

new way: all you gotta do is create a new PDO object. PDO's constructor takes at most 4 parameters, DSN, username, password, and an array of driver options.

A DSN is basically a string of options that tell PDO which driver to use, and the connection details... You can look up all the options here PDO MYSQL DSN (http://www.php.net/manual/en/ref.pdo-mysql.connection.php) .

```
<?php
$db = new PDO('mysql:host=localhost;dbname=testdb;charset=utf8', 'username', 'pass
```

Note: If you get an error about character sets, make sure you add the charset parameter to the DSN. Adding the charset to the DSN is very important for security reasons, most examples you'll see around leave it out. MAKE SURE TO INCLUDE THE CHARSET!

You can also pass in several driver options as an array to the fourth parameters. I recommend passing the parameter which puts PDO into exception mode, which I will explain in the next section. The other parameter is to turn off prepare emulation which is enabled in MySQL driver by default, but really should be turned off to use PDO safely and is really only usable if you are using an old version of MySQL.

```
<?php
$db = new PDO('mysql:host=localhost;dbname=testdb;charset=utf8', 'username', 'pass
```

You can also set some attributes after PDO construction with the `setAttribute` method:

```
<?php
$db = new PDO('mysql:host=localhost;dbname=testdb;charset=utf8', 'username', 'pass
$db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
$db->setAttribute(PDO::ATTR_EMULATE_PREPARES, false);
```

Error Handling

Consider your typical mysql_* error handling:

```
<?php
//connected to mysql
$result = mysql_query("SELECT * FROM table", $link) or die(mysql_error($link));
```

OR die is a pretty bad way to handle errors, yet this is typical mysql code. You can't handle die(); as it will just end the script abruptly and then echo the error to the screen which you usually do **NOT** want to show to your end users allowing nasty hackers discover your schema.

PDO has three error handling modes.

1. *PDO::ERRMODE_SILENT* acts like mysql_* where you must check each result and then look at `$db->errorInfo();` to get the error details.
2. *PDO::ERRMODE_WARNING* throws PHP Warnings
3. *PDO::ERRMODE_EXCEPTION* throws PDOException. In my opinion this is the mode you should use. It acts very much like `or die(mysql_error());` when it isn't caught, but unlike `or die()` the PDOException can be caught and handled gracefully if you choose to do so.

```
<?php
try {
    //connect as appropriate as above
    $db->query('hi'); //invalid query!
} catch(PDOException $ex) {
    echo "An Error occured!"; //user friendly message
    some_logging_function($ex->getMessage());
}
```

NOTE: you do not have to handle with try catch right away. You can catch it anytime that is appropriate. It may make more sense to catch it at a higher level like outside of the function that calls the PDO stuff:

```
<?php
function getData($db) {
   $stmt = $db->query("SELECT * FROM table");
   return $stmt->fetchAll(PDO::FETCH_ASSOC);
}

//then much later
try {
   getData($db);
} catch(PDOException $ex) {
   //handle me.
}
```

or you may not want to handle the exception with try/catch at all, and have it work much like or die(); does. You can hide the dangerous error messages in production by turning display_errors off and just reading your error log.

Running Simple Select Statements

Consider the mysql_* code:

```
<?php
$result = mysql_query('SELECT * from table') or die(mysql_error());

$num_rows = mysql_num_rows($result);

while($row = mysql_fetch_assoc($result)) {
```

```
    echo $row['field1'].' '.$row['field2']; //etc...
}
```

In PDO You can run such queries like this:

```
<?php
foreach($db->query('SELECT * FROM table') as $row) {
    echo $row['field1'].' '.$row['field2']; //etc...
}
```

`query()` method returns a `PDOStatement` object. You can also fetch results this way:

```
<?php
$stmt = $db->query('SELECT * FROM table');

while($row = $stmt->fetch(PDO::FETCH_ASSOC)) {
    echo $row['field1'].' '.$row['field2']; //etc...
}
```

or

```
<?php
$stmt = $db->query('SELECT * FROM table');
$results = $stmt->fetchAll(PDO::FETCH_ASSOC);
//use $results
```

Fetch Modes

Note the use of `PDO::FETCH_ASSOC` in the `fetch()` and `fetchAll()` code above. This tells PDO to return the rows as an associative array with the field names as keys. Other fetch modes like `PDO::FETCH_NUM` returns the row as a numerical array. The default is to fetch with PDO::FETCH_BOTH which duplicates the data with both numerical and associative keys. It's recommended you specify one or the other so you don't have arrays that are double the size! PDO can also fetch objects with `PDO::FETCH_OBJ`, and can take existing classes with `PDO::FETCH_CLASS`. It can also bind into specific variables with `PDO::FETCH_BOUND` and using `bindColumn` method. There are even more choices! Read about them all here: PDOStatement Fetch documentation (http://www.php.net/manual/en/pdostatement.fetch.php) .

Getting Row Count

Instead of using mysql_num_rows to get the number of returned rows you can get a `PDOStatement` and do rowCount();

```
<?php
$stmt = $db->query('SELECT * FROM table');
$row_count = $stmt->rowCount();
echo $row_count.' rows selected';
```

NOTE: Though the documentation says this method is only for returning affected rows from `UPDATE`, `INSERT`, `DELETE` queries, with the PDO_MYSQL driver (and this driver only) you can get the row count for `SELECT` queries. Keep this in mind when writing code for multiple databases.

This is because MySQL's protocol is one of the very few that give this information to the client for `SELECT` statements. Most other database vendors don't bother divulging this information to the client as it would incur more overhead in their implementations.

Getting the Last Insert Id

Previously in mysql_* you did something like this.

```
<?php
$result = mysql_query("INSERT INTO table(firstname, lastname) VALUES('John', 'Doe'
$insert_id = mysql_insert_id();
```

With PDO you just do run the lastInsertId method.

```
<?php
$result = $db->exec("INSERT INTO table(firstname, lastname) VAULES('John', 'Doe')"
$insertId = $db->lastInsertId();
```

Running Simple INSERT, UPDATE, or DELETE statements

Consider the mysql_* code.

```
<?php
$results = mysql_query("UPDATE table SET field='value'") or die(mysql_error());
$affected_rows = mysql_affected_rows($result);
echo $affected_rows.' were affected';
```

for PDO this would look like:

```
<?php
$affected_rows = $db->exec("UPDATE table SET field='value'");
echo $affected_rows.' were affected'
```

Do the same for simple `DELETE`, and `INSERT statements as well`

Running Statements With Parameters

So far we've only shown simple statements that don't take in any variables. These are simple statements and PDO has the shortcut methods `query` for `SELECT` statements and `exec` for `INSERT`, `UPDATE`, `DELETE` statements. For statements that take in variable parameters, you should use bound parameter methods to execute your queries safely. Consider the following mysql_* code.

```
<?php
```

```
$results = mysql_query(sprintf("SELECT * FROM table WHERE id='%s' AND name='%s'",
                        mysql_real_escape_string($id), mysql_real_escape_string($na
$rows = array();
while($row = mysql_fetch_assoc($results)){
    $rows[] = $row;
}
```

Man! that's a pain, especially if you have lots of parameters. This is how you can do it in PDO:

```
<?php
$stmt = $db->prepare("SELECT * FROM table WHERE id=? AND name=?");
$stmt->execute(array($id, $name));
$rows = $stmt->fetchAll(PDO::FETCH_ASSOC);
```

So what's going on here? The `prepare` method sends the query to the server, and it's compiled with the '?' placeholders to be used as expected arguments. The `execute` method sends the arguments to the server and runs the compiled statement. Since the query and the dynamic parameters are sent separately, there is no way that any SQL that is in those parameters can be executed... so NO SQL INJECTION can occur! This is a much better and safer solution than concatenating strings together.

NOTE: when you bind parameters, do NOT put quotes around the placeholders. It will cause strange SQL syntax errors, and quotes aren't needed as the type of the parameters are sent during `execute` so they are not needed to be known at the time of `prepare`.

There's a few other ways you can bind parameters as well. Instead of passing them as an array, which binds each parameter as a String type, you can use `bindValue` and specify the type for each parameter:

```
<?php
$stmt = $db->prepare("SELECT * FROM table WHERE id=? AND name=?");
$stmt->bindValue(1, $id, PDO::PARAM_INT);
$stmt->bindValue(2, $name, PDO::PARAM_STR);
$stmt->execute();
$rows = $stmt->fetchAll(PDO::FETCH_ASSOC);
```

Named Placeholders

Now if you have lots of parameters to bind, doesn't all those '?' characters make you dizzy and are hard to count? Well, in PDO you can use *named placeholders* instead of the '?':

```
<?php
$stmt = $db->prepare("SELECT * FROM table WHERE id=:id AND name=:name");
$stmt->bindValue(':id', $id, PDO::PARAM_INT);
$stmt->bindValue(':name', $name, PDO::PARAM_STR);
$stmt->execute();
$rows = $stmt->fetchAll(PDO::FETCH_ASSOC);
```

You can bind using `execute` with an array as well:

```
<?php
$stmt = $db->prepare("SELECT * FROM table WHERE id=:id AND name=:name");
$stmt->execute(array(':name' => $name, ':id' => $id));
```

```
$rows = $stmt->fetchAll(PDO::FETCH_ASSOC);
```

INSERT, DELETE, UPDATE Prepared Queries

Prepared Statements for `INSERT`, `UPDATE`, and `DELETE` are not different than `SELECT`. But lets do some examples anyway:

```
<?php
$stmt = $db->prepare("INSERT INTO table(field1,field2,field3,field4,field5) VALUES
$stmt->execute(array(':field1' => $field1, ':field2' => $field2, ':field3' => $fie
$affected_rows = $stmt->rowCount();
```

```
<?php
$stmt = $db->prepare("DELETE FROM table WHERE id=:id");
$stmt->bindValue(':id', $id, PDO::PARAM_STR);
$stmt->execute();
$affected_rows = $stmt->rowCount();
```

```
<?php
$stmt = $db->prepare("UPDATE table SET name=? WHERE id=?");
$stmt->execute(array($name, $id));
$affected_rows = $stmt->rowCount();
```

Preparing Statements using SQL functions

You may ask how do you use SQL functions with prepared statements. I've seen people try to bind functions into placeholders like so:

```
<?php
//THIS WILL NOT WORK!
$time = 'NOW()';
$name = 'BOB';
$stmt = $db->prepare("INSERT INTO table(`time`, `name`) VALUES(?, ?)");
$stmt->execute(array($time, $name));
```

This does not work, you need to put the function in the query as normal:

```
<?php
$name = 'BOB';
$stmt = $db->prepare("INSERT INTO table(`time`, `name`) VALUES(NOW(), ?)");
$stmt->execute(array($name));
```

You can bind arguments into SQL functions however:

```
<?php
$name = 'BOB';
$password = 'badpass';
$stmt = $db->prepare("INSERT INTO table(`hexvalue`, `password`) VALUES(HEX(?), PAS
```

```
$stmt->execute(array($name, $password));
```

Also note that this does NOT work for LIKE statements:

```
<?php
//THIS DOES NOT WORK
$stmt = $db->prepare("SELECT field FROM table WHERE field LIKE %?%");
$stmt->bindParam(1, $search, PDO::PARAM_STR);
$stmt->execute();
```

So do this instead:

```
<?php
$stmt = $db->prepare("SELECT field FROM table WHERE field LIKE ?");
$stmt->bindValue(1, "%$search%", PDO::PARAM_STR);
$stmt->execute();
```

Note we used **bindValue** and not **bindParam**. Trying to bind a parameter by reference will generate a Fatal Error and this cannot be caught by PDOException either.

Executing prepared statements in a loop

Prepared statements excel in being called multiple times in a row with different values. Because the sql statement gets compiled first, it can be called multiple times in a row with different arguments, and you'll get a big speed increase vs calling mysql_query over and over again!

Typically this is done by binding parameters with `bindParam`. `bindParam` is much like `bindValue` except instead of binding the value of a variable, it binds the variable itself, so that if the variable changes, it will be read at the time of `execute`.

```
<?php
$values = array('bob', 'alice', 'lisa', 'john');
$name = '';
$stmt = $db->prepare("INSERT INTO table(`name`) VALUES(:name)");
$stmt->bindParam(':name', $name, PDO::PARAM_STR);
foreach($values as $name) {
    $stmt->execute();
}
```

Transactions

Here's an example of using transactions in PDO: (note that calling `beginTransaction()` turns off auto commit automatically):

```
<?php
try {
    $db->beginTransaction();

    $db->exec("SOME QUERY");
```

```
    $stmt = $db->prepare("SOME OTHER QUERY?");
    $stmt->execute(array($value));

    $stmt = $db->prepare("YET ANOTHER QUERY??");
    $stmt->execute(array($value2, $value3));

    $db->commit();
} catch(PDOException $ex) {
    //Something went wrong rollback!
    $db->rollBack();
    echo $ex->getMessage();
}
```

See Also

Validation and SQL Injection

External links

PDO Documentation (http://www.php.net/manual/en/book.pdo.php)

Retrieved from "http://wiki.hashphp.org/PDO_Tutorial_for_MySQL_Developers"

- This page was last modified on 7 August 2013, at 17:37.
- This page has been accessed 439,491 times.