PHP

Why you Should be using PHP's PDO for Database Access

by Erik Wurzer 25 Jan 2012 233 Comments English

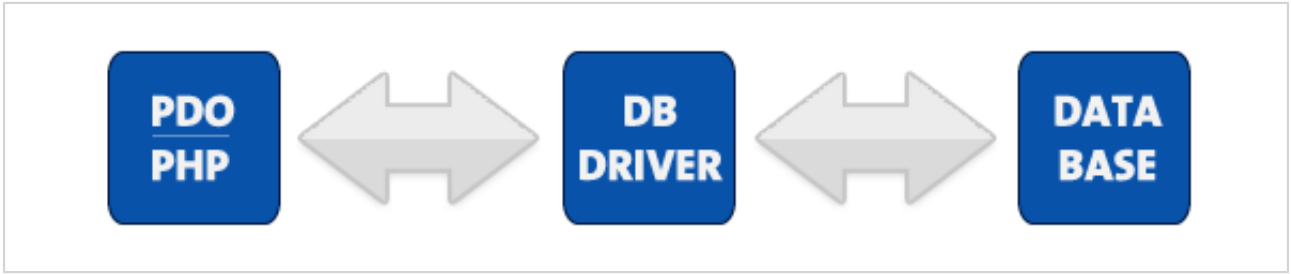
Many PHP programmers learned how to access databases by using either the MySQL or MySQLi extensions. As of PHP 5.1, there's a better way. PHP Data Objects (PDO) provide methods for prepared statements and working with objects that will make you far more productive!

Republished Tutorial
Every few weeks, we revisit some of our reader's favorite posts from throughout the history of the site. This tutorial was first published in May of 2010.

PDO Introduction

> *"PDO - PHP Data Objects - is a database access layer providing a uniform method of access to multiple databases."*

It doesn't account for database-specific syntax, but can allow for the process of switching databases and platforms to be fairly painless, simply by switching the connection string in many instances.



This tutorial isn't meant to be a complete how-to on SQL. It's written primarily for

people currently using the mysql or mysqli extension to help them make the jump to the more portable and powerful PDO.

Database Support

The extension can support any database that a PDO driver has been written for. At the time of this writing, the following database drivers are available:

- PDO_DBLIB (FreeTDS / Microsoft SQL Server / Sybase)
- PDO_FIREBIRD (Firebird/Interbase 6)
- PDO_IBM (IBM DB2)
- PDO_INFORMIX (IBM Informix Dynamic Server)
- PDO_MYSQL (MySQL 3.x/4.x/5.x)
- PDO_OCI (Oracle Call Interface)
- PDO_ODBC (ODBC v3 (IBM DB2, unixODBC and win32 ODBC))
- PDO_PGSQL (PostgreSQL)
- PDO_SQLITE (SQLite 3 and SQLite 2)
- PDO_4D (4D)

All of these drivers are not necessarily available on your system; here's a quick way to find out which drivers you have:

```
1 print_r(PDO::getAvailableDrivers());
```

Connecting

Different databases may have slightly different connection methods. Below, the method to connect to some of the most popular databases are shown. You'll notice that the first three are identical, other then the database type - and then SQLite has its own syntax.

```
                    Database Type
$DBH = new PDO("mysql:host=$host;dbname=$dbname", $user, $pass);
Database Handle      Database Specific Connection String
```

```
01 try {
```

```
try {
  # MS SQL Server and Sybase with PDO_DBLIB
  $DBH = new PDO("mssql:host=$host;dbname=$dbname, $user, $pass")
  $DBH = new PDO("sybase:host=$host;dbname=$dbname, $user, $pass"

  # MySQL with PDO_MYSQL
  $DBH = new PDO("mysql:host=$host;dbname=$dbname", $user, $pass)

  # SQLite Database
  $DBH = new PDO("sqlite:my/database/path/database.db");
}
catch(PDOException $e) {
    echo $e->getMessage();
}
```

Please take note of the try/catch block - you should always wrap your PDO operations in a try/catch, and use the exception mechanism - more on this shortly. Typically you're only going to make a single connection - there are several listed to show you the syntax. $DBH stands for 'database handle' and will be used throughout this tutorial.

You can close any connection by setting the handle to null.

```
# close the connection
$DBH = null;
```

You can get more information on database specific options and/or connection strings for other databases from PHP.net.

Exceptions and PDO

PDO can use exceptions to handle errors, which means anything you do with PDO should be wrapped in a try/catch block. You can force PDO into one of three error modes by setting the error mode attribute on your newly created database handle.

Here's the syntax:

```
$DBH->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_SILENT );
$DBH->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING );
$DBH->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION );
```

No matter what error mode you set, an error connecting will always produce an

exception, and creating a connection should always be contained in a try/catch block.

PDO::ERRMODE_SILENT

This is the default error mode. If you leave it in this mode, you'll have to check for errors in the way you're probably used to if you used the mysql or mysqli extensions. The other two methods are more ideal for DRY programming.

PDO::ERRMODE_WARNING

This mode will issue a standard PHP warning, and allow the program to continue execution. It's useful for debugging.

PDO::ERRMODE_EXCEPTION

This is the mode you should want in most situations. It fires an exception, allowing you to handle errors gracefully and hide data that might help someone exploit your system. Here's an example of taking advantage of exceptions:

```
01  # connect to the database
02  try {
03    $DBH = new PDO("mysql:host=$host;dbname=$dbname", $user, $pass)
04    $DBH->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION )
05
06    # UH-OH! Typed DELECT instead of SELECT!
07    $DBH->prepare('DELECT name FROM people');
08  }
09  catch(PDOException $e) {
10      echo "I'm sorry, Dave. I'm afraid I can't do that.";
11      file_put_contents('PDOErrors.txt', $e->getMessage(), FILE_APP
12  }
```

There's an intentional error in the select statement; this will cause an exception. The exception sends the details of the error to a log file, and displays a friendly (or not so friendly) message to the user.

Insert and Update

Inserting new data, or updating existing data is one of the more common database operations. Using PDO, this is normally a two-step process. Everything covered in this section applies equally to both UPDATE and INSERT operations.

Here's an example of the most basic type of insert:

```
# STH means "Statement Handle"
$STH = $DBH->prepare("INSERT INTO folks ( first_name ) values ( 'C
$STH->execute();
```

You could also accomplish the same operation by using the exec() method, with one less call. In most situations, you're going to use the longer method so you can take advantage of prepared statements. Even if you're only going to use it once, using prepared statements will help protect you from SQL injection attacks.

Prepared Statements

> *Using prepared statements will help protect you from SQL injection.*

A prepared statement is a precompiled SQL statement that can be executed multiple times by sending just the data to the server. It has the added advantage of automatically making the data used in the placeholders safe from SQL injection attacks.

You use a prepared statement by including placeholders in your SQL. Here's three examples: one without placeholders, one with unnamed placeholders, and one with named placeholders.

```
# no placeholders - ripe for SQL Injection!
$STH = $DBH->("INSERT INTO folks (name, addr, city) values ($name,

# unnamed placeholders
$STH = $DBH->("INSERT INTO folks (name, addr, city) values (?, ?,

# named placeholders
$STH = $DBH->("INSERT INTO folks (name, addr, city) value (:name,
```

You want to avoid the first method; it's here for comparison. The choice of using

named or unnamed placeholders will affect how you set data for those statements.

Unnamed Placeholders

```
01 # assign variables to each place holder, indexed 1-3
02 $STH->bindParam(1, $name);
03 $STH->bindParam(2, $addr);
04 $STH->bindParam(3, $city);
05
06 # insert one row
07 $name = "Daniel"
08 $addr = "1 Wicked Way";
09 $city = "Arlington Heights";
10 $STH->execute();
11
12 # insert another row with different values
13 $name = "Steve"
14 $addr = "5 Circle Drive";
15 $city = "Schaumburg";
16 $STH->execute();
```

There are two steps here. First, we assign variables to the various placeholders (lines 2-4). Then, we assign values to those placeholders and execute the statement. To send another set of data, just change the values of those variables and execute the statement again.

Does this seem a bit unwieldy for statements with a lot of parameters? It is. However, if your data is stored in an array, there's an easy shortcut:

```
1 # the data we want to insert
2 $data = array('Cathy', '9 Dark and Twisty Road', 'Cardiff');
3
4 $STH = $DBH->("INSERT INTO folks (name, addr, city) values (?, ?,
5 $STH->execute($data);
```

That's easy!

The data in the array applies to the placeholders in order. $data[0] goes into the first placeholder, $data[1] the second, etc. However, if your array indexes are not in order, this won't work properly, and you'll need to re-index the array.

Named Placeholders

You could probably guess the syntax, but here's an example:

```
# the first argument is the named placeholder name - notice named
# placeholders always start with a colon.
$STH->bindParam(':name', $name);
```

You can use a shortcut here as well, but it works with associative arrays. Here's an example:

```
# the data we want to insert
$data = array( 'name' => 'Cathy', 'addr' => '9 Dark and Twisty', '

# the shortcut!
$STH = $DBH->("INSERT INTO folks (name, addr, city) value (:name,
$STH->execute($data);
```

The keys of your array do not need to start with a colon, but otherwise need to match the named placeholders. If you have an array of arrays you can iterate over them, and simply call the execute with each array of data.

Another nice feature of named placeholders is the ability to insert objects directly into your database, assuming the properties match the named fields. Here's an example object, and how you'd perform your insert:

```
# a simple object
class person {
    public $name;
    public $addr;
    public $city;

    function __construct($n,$a,$c) {
        $this->name = $n;
        $this->addr = $a;
        $this->city = $c;
    }
    # etc ...
}

$cathy = new person('Cathy','9 Dark and Twisty','Cardiff');

# here's the fun part:
$STH = $DBH->("INSERT INTO folks (name, addr, city) value (:name,
$STH->execute((array)$cathy);
```

By casting the object to an array in the execute, the properties are treated as array keys.

Selecting Data

Data is obtained via the ->fetch(), a method of your statement handle. Before calling fetch, it's best to tell PDO how you'd like the data to be fetched. You have the following options:

- **PDO::FETCH_ASSOC:** returns an array indexed by column name
- **PDO::FETCH_BOTH (default):** returns an array indexed by both column name and number
- **PDO::FETCH_BOUND:** Assigns the values of your columns to the variables set with the ->bindColumn() method
- **PDO::FETCH_CLASS:** Assigns the values of your columns to properties of the named class. It will create the properties if matching properties do not exist
- **PDO::FETCH_INTO:** Updates an existing instance of the named class
- **PDO::FETCH_LAZY**: Combines PDO::FETCH_BOTH/PDO::FETCH_OBJ, creating the object variable names as they are used
- **PDO::FETCH_NUM:** returns an array indexed by column number
- **PDO::FETCH_OBJ:** returns an anonymous object with property names that correspond to the column names

In reality, there are three which will cover most situations: FETCH_ASSOC, FETCH_CLASS, and FETCH_OBJ. In order to set the fetch method, the following syntax is used:

```
1 $STH->setFetchMode(PDO::FETCH_ASSOC);
```

You can also set the fetch type directly within the ->fetch() method call.

FETCH_ASSOC

This fetch type creates an associative array, indexed by column name. This should be quite familiar to anyone who has used the mysql/mysqli extensions. Here's an

be quite familiar to anyone who has used the mysql/mysqli extensions. Here's an example of selecting data with this method:

```
# using the shortcut ->query() method here since there are no var
# values in the select statement.
$STH = $DBH->query('SELECT name, addr, city from folks');

# setting the fetch mode
$STH->setFetchMode(PDO::FETCH_ASSOC);

while($row = $STH->fetch()) {
    echo $row['name'] . "\n";
    echo $row['addr'] . "\n";
    echo $row['city'] . "\n";
}
```

The while loop will continue to go through the result set one row at a time until complete.

FETCH_OBJ

This fetch type creates an object of std class for each row of fetched data. Here's an example:

```
# creating the statement
$STH = $DBH->query('SELECT name, addr, city from folks');

# setting the fetch mode
$STH->setFetchMode(PDO::FETCH_OBJ);

# showing the results
while($row = $STH->fetch()) {
    echo $row->name . "\n";
    echo $row->addr . "\n";
    echo $row->city . "\n";
}
```

FETCH_CLASS

> *The properties of your object are set BEFORE the constructor is called. This is important.*

This fetch method allows you to fetch data directly into a class of your choosing. When you use FETCH_CLASS, the properties of your object are set BEFORE the constructor is called. Read that again, it's important. If properties matching the

column names don't exist, those properties will be created (as public) for you.

This means if your data needs any transformation after it comes out of the database, it can be done automatically by your object as each object is created.

As an example, imagine a situation where the address needs to be partially obscured for each record. We could do this by operating on that property in the constructor. Here's an example:

```
01 class secret_person {
02     public $name;
03     public $addr;
04     public $city;
05     public $other_data;
06
07     function __construct($other = '') {
08         $this->address = preg_replace('/[a-z]/', 'x', $this->addr
09         $this->other_data = $other;
10     }
11 }
```

As data is fetched into this class, the address has all its lowercase *a-z* letters replaced by the letter *x*. Now, using the class and having that data transform occur is completely transparent:

```
1 $STH = $DBH->query('SELECT name, addr, city from folks');
2 $STH->setFetchMode(PDO::FETCH_CLASS, 'secret_person');
3
4 while($obj = $STH->fetch()) {
5     echo $obj->addr;
6 }
```

If the address was '5 Rosebud,' you'd see '5 Rxxxxxx' as your output. Of course, there may be situations where you want the constructor called before the data is assigned. PDO has you covered for this, too.

```
1 $STH->setFetchMode(PDO::FETCH_CLASS | PDO::FETCH_PROPS_LATE, 'secr
```

Now, when you repeat the previous example with this fetch mode (PDO::FETCH_PROPS_LATE) the address will NOT be obscured, since the constructor was called and the properties were assigned.

Finally, if you really need to, you can pass arguments to the constructor when

Finally, if you really need to, you can pass arguments to the constructor when fetching data into objects with PDO:

```
$STH->setFetchMode(PDO::FETCH_CLASS, 'secret_person', array('stuff
```

If you need to pass different data to the constructor for each object, you can set the fetch mode inside the fetch method:

```
$i = 0;
while($rowObj =  $STH->fetch(PDO::FETCH_CLASS, 'secret_person', ar
    // do stuff
    $i++
}
```


Some Other Helpful Methods

While this isn't meant to cover everything in PDO (it's a huge extension!) there are a few more methods you'll want to know in order to do basic things with PDO.

```
$DBH->lastInsertId();
```

The ->lastInsertId() method is always called on the database handle, not statement handle, and will return the auto incremented id of the last inserted row by that connection.

```
$DBH->exec('DELETE FROM folks WHERE 1');
$DBH->exec("SET time_zone = '-8:00'");
```

The ->exec() method is used for operations that can not return data other then the affected rows. The above are two examples of using the exec method.

```
$safe = $DBH->quote($unsafe);
```

The ->quote() method quotes strings so they are safe to use in queries. This is your fallback if you're not using prepared statements.

```
$rows_affected = $STH->rowCount();
```

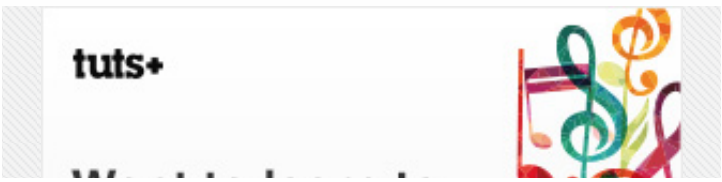
The ->rowCount() method returns an integer indicating the number of rows affected by an operation. In at least one known version of PDO, according to [this bug report](http://bugs.php.net/40822) the method does not work with select statements. If you're having this problem, and can't upgrade PHP, you could get the number of rows with the following:

```
$sql = "SELECT COUNT(*) FROM folks";
if ($STH = $DBH->query($sql)) {
    # check the row count
    if ($STH->fetchColumn() > 0) {

    # issue a real select here, because there's data!
    }
    else {
        echo "No rows matched the query.";
    }
}
```

Conclusion

I hope this helps some of you migrate away from the mysql and mysqli extensions. What do you think? Are there any of you out there who might make the switch?



Difficulty:
Intermediate

Length:
Short

Categories:

PHP | Databases | Web Development | SQL | Back-End | Database | OOP | Object-Oriented Programming

PDO
PHP

Translations Available:
Português

Tuts+ tutorials are translated by our community members. If you'd like to translate this post into another language, let us know!

About Erik Wurzer

N/A

Suggested Tuts+ Course