

# A Crash Course in PDO

PDO (PHP Data Objects) provides a vendor-neutral method of accessing a database through PHP. This means that, once you have established a connection to the specific database, the methods used to access and manipulate data are all generic, and do not require re-writing if you change the type of database. Features which may not be present in a particular database will generally be emulated, or at least ignored.

This article will outline the most useful techniques of using PDO to access your database. The first part outlines the objects which you will be using. The second part is a recipe book of techniques using these objects.

PDO Requires PHP Version 5. If you're stuck with Version 4, you might try:

<http://www.phpclasses.org/browse/package/2572.html>

The main references for PDO are:

<http://www.php.net/manual/en/class.pdo.php>

<http://www.php.net/manual/en/class.pdostatement.php>

## Contents

|  |          |
|--|----------|
| <b>PDO Objects</b> .....                                     | <b>2</b> |
| PDO Object .....   | 2        |
| PDOStatement Object.....                                     | 2        |
| <b>PDO Recipe book</b> .....                                 | <b>3</b> |
| Establishing a Connection .....                              | 3        |
| Simple Queries Without User Input.....                       | 4        |
| SELECT .....   | 4        |
| INSERT, UPDATE & DELETE.....                                 | 5        |
| Queries With User Input Queries Which Will Be Repeated ..... | 6        |
| Preparing .....  | 6        |
| Executing .....  | 6        |
| <b>Summary of Process</b> .....                              | <b>7</b> |

# PDO Objects

PDO makes use of two main objects. The PDO object itself represents a connection to the database, and provides simple methods to execute an SQL statement. It also provides a method to prepare an SQL statement for later use. The PDOStatement object represents a prepared statement, as well as a result set from an executed SQL statement.

## PDO Object

This represents a connection to the Database. All database operations are initiated through the PDO object. The PDO object is created when you connect to the database. After that, you use its methods to access the database. The most useful methods are:

|                        |   |
|------------------------|---|
| <code>exec()</code>    | Execute an SQL statement returning the number of rows affected  |
| <code>query()</code>   | Execute an SQL statement returning a result set as a PDOStatement   |
| <code>prepare()</code> | Prepares a statement returning a result set as a PDOStatement<br>You can use question marks (?) for values.<br>You can then call the <code>execute(array())</code> method |

## PDOStatement Object

The PDOStatement represents a prepared statement, as well as a returned result set. The name is possibly confusing, since it represents a prepared statement before it is executed, as well as the result after it is executed.

A PDOStatement is created as a result of a `PDO->query` operation (where it represents a result set), a `PDO->prepare` operation (where it represents a prepared statement) or a `PDO->execute` operation (where it represents a result set from your prepared statement).

The most useful methods are:

### For a prepared statement:

|                        |   |
|------------------------|---|
| <code>execute()</code> | Execute the prepared statement.<br>You can use an array of values to replace the question mark parameters |
|------------------------|---|

### For a result set:

|                            |   |
|----------------------------|---|
| <code>fetch()</code>       | Returns the next row.<br>Useful arguments: <code>PDO::FETCH_ASSOC</code> ,<br><code>PDO::FETCH_NUM</code> ,<br><code>PDO::FETCH_BOTH</code> (default) |
| <code>fetchAll()</code>    | Returns the whole result set as an array  |
| <code>fetchColumn()</code> | Returns a single column of the next row.  |

PDOStatement implements the `Traversable` interface. This means that a result set can be iterated with `foreach()`.

# PDO Recipe book

---

For the following recipes, these examples will be used:

```
$database='things';
$user='me';
$password='secret';
```

The connection statement includes a connection string. This string is specific to individual databases. Below are some examples:

```
$dsn="mysql:host=localhost;dbname=$database";
$dsn="pgsql:host=localhost;dbname=$database";
$dsn="odbc:Driver={Microsoft Access Driver (*.mdb)};Dbq=C:\$database.mdb;Uid=Admin";
$dsn="sqlite:$database.sqlite";
$dsn="sqlite::memory";
```

## Establishing a Connection

To connect to a database you create a new PDO object. Since a failure will generate an exception, and since the default behaviour is to display more information than you probably want to share, it is best to wrap this inside a `try .. catch` block.

Some databases require you to log in with a userid and password; some do not:

```
try {
    $pdo = new PDO($dsn); // sqlite, MSAccess
    $pdo = new PDO($dsn,$user,$password);
    // mysql, postgresql, etc
} catch(PDOException $e) {
    die ('Oops'); // Exit, displaying an error message
}
```

### Simple Queries Without User Input

For simple queries which do not involve user input, you can use the `PDO->exec()` and `PDO->query()` methods. These methods will send the SQL statement to the database and return either a row count or a result set. They should not be used if there is user input involved, since you will need to protect yourself against SQL injection attacks which involve specially crafted user inputs. If this is a possibility, you should instead use the prepare method below.

#### SELECT

To retrieve data using `SELECT`, you use the `PDO->query()` method. This will return a result set as a `PDOStatement`.

```
$sql='SELECT ...';  
$pds=$pdo->query($sql);
```

#### Reading the result set.

To fetch a single row, or a single row at a time (starting with the first), use the `PDOStatement->fetch()` method:

```
$row=$pds->fetch();
```

Of itself, this will give you an array of columns. Two arrays, actually, combined. The array will contain both associative keys (the same as the fields specified or retrieved) and numbered keys (in the specified order of the columns).

If you want to specify the type of array you get, you can use one of:

```
$row=$pds->fetch(PDO::FETCH_ASSOC);  
$row=$pds->fetch(PDO::FETCH_NUM);  
// This is the default:  
$row=$pds->fetch(PDO::FETCH_BOTH);
```

If you want more than the first row, you can use:

```
while($row = $pds->fetch()) {  
    // Fetch & assign to row  
  
}
```

Since `PDOStatement` implements `Traversable`, you can iterate it as an array. In this case, you can use `foreach`

```
foreach($pds as $row) {  
    // Does the fetching for you  
  
}
```

#### Fetching the Whole Result Set

You can fetch the entire result set into a array with all of the rows:

```
$rows=$pds->fetchAll();
```

You can, but you probably shouldn't, unless you can be sure that your result set isn't too big for memory.

Sometimes, you need only one column of the result set. For this you can use `fetchColumn()`. The optional parameter is the column number (starting at 0, which is the default). Each subsequent call to `fetchColumn()` will fetch the same column from the *next* row.

### INSERT, UPDATE & DELETE

To alter data using INSERT, UPDATE or DELETE, you can use the PDO->exec() method:

```
$sql='INSERT ...';  
$sql='UPDATE ...';  
$sql='DELETE ...';  
  
$count = $pdo->exec($sql);
```

In each case, the return result will be the number of rows affected.

# Queries With User Input

## Queries Which Will Be Repeated

Most databases allow you to prepare a statement before executing it. Preparing a statement has two benefits:

1. SQL statements need to be interpreted, checked for errors, analysed and optimised, all before executing them with actual data. If you have a statement which will be repeated, with only the data changing, then a prepared statement will have a performance benefit.
2. Since the SQL statement is interpreted first, and the data inserted later, there is no risk of data being misinterpreted as part of the SQL statement. This means you are protected from SQL injection attacks, which rely on sending carefully crafted SQL expressions masquerading as data.

If you have no untrusted data (such as from users), and are executing a statement only once, the overhead of preparing is probably not worth while. In this case, you can use the `exec()` or `query()` methods above.

To use prepared statements will require two stages: preparing the statement, and executing it with data.

### Preparing

You prepare a statement by replacing all data with question marks:

```
$sql='INSERT INTO ... (...) VALUES (?, ?, ?)';  
$sql='SELECT ... FROM ... WHERE ...=?';  
$sql='DELETE FROM ... WHERE ...=?';  
$sql='UPDATE ... SET ...=? WHERE ...=?';  
  
$pds=$pdo->prepare($sql);
```

Note that the question mark parameter does not include quotes, whether or not it represents a string.

The result is a `PDOStatement`, but not (yet) a result set.

### Executing

To execute a prepare statement, use the `execute` method. This takes an array of values to replace the question marks.

```
$pds->execute(array(...));
```

You do not need to assign the result into a new variable. The same `PDOStatement` object will now contain the result set, and you can fetch the results as you would above.

To use a prepared statement repetitively, prepare it before you begin the loop, and execute it repetitively:

```
$data=array(...);  
$pds=$pdo->prepare('INSERT INTO ... (...) VALUES (?)');  
  
foreach ($data as $value) {  
    $pds->execute(array($value));  
}
```

Note that `execute` always takes an array argument, even if there is only one value.

# Summary of Process

