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.

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

Table of Contents

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:

`exec()`	Execute an SQL statement returning the number of rows affected
`query()`	Execute an SQL statement returning a result set as a `PDOStatement`
`prepare()`	Prepares a statement returning a result set as a `PDOStatement` You can use question marks (?) for values. You can then call the `execute(array())` 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:

`execute()`	Execute the prepared statement. You can use an array of values to replace the question mark parameters

For a result set:

`fetch()`	Returns the next row. Useful arguments: `PDO::FETCH_ASSOC`, `PDO::FETCH_NUM`, `PDO::FETCH_BOTH` (default)
`fetchAll()`	Returns the whole result set as an array
`fetchColumn()`	Returns a single column of the next row.

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

Working With PDO

PHP Data Objects allow you to work with a database without having to worry about the details of the database functions. In principal, you can use the same code to work with different database types, though some SQL statements may need adjustment, due to differences between databases.

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.

Establishing a Connection

Before working with PDO, you will need to create a connection. This is in the form of a PDO object, which represents the connection.

You will require the following:

- A connection string: this informs PDO which database you are connecting to.
  This will also include the location of the data.

- User name and password. Depending on the database, you may need to authenticate your connection.

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

$dsn="mysql:host=localhost;dbname=$database";     //    mysql
$dsn="sqlite:$database.sqlite";                   //    sqlite
```

A connection attempt may result in an error. The normal behaviour is to display as much information as possible, but this is probably more than you wish to share with others. For this reason, it is best to wrap the connection inside a `try ... catch` block:

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

Other Databases

If you want to connect to a different database, the following connection strings may be used:

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

For the connection script you may use:

```
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
}
```

Prepared Statements and SQL Injection

The Risk: SQL Injection

The biggest risk to your database comes from including user data in your SQL statements. This may be miss-interpreted as part of the SQL statement. Where a user is deliberately supplying this data to break into the database, this is called SQL Injection.

For example, suppose you are performing a simple login using an email and password supplied by the user. The SQL statement might be something like this:

```
SELECT count(*) FROM users WHERE email='...' AND passwd='...'
```

Now, suppose the user supplies the following as their email address:

```
fred' OR 1=1; --
```

This clearly is not a proper email address, but it might still be inserted as follows:

```
SELECT count(*) FROM users WHERE email='fred' OR 1=1; -- ' AND passwd='...'
```

The condition `OR 1=1` will always be true, and what follows after the comment code `--` will be ignored. This simple injection will allow a user to break into the database.

The problem arises because the inserted data will be interpreted with the rest of the SQL.

Prepared Statements

Most databases allow you to prepare a statement before executing it. SQL statements need to be interpreted, checked for errors, analyzed and optimised, all before executing them with actual data.

To protect yourself against SQL injection, you prepare your SQL statement first, and then execute it with the data afterwards. When the data is inserted, it can no longer be interpreted, and so will be passed purely as data. Note that the above email address would presumably not be in the database, and would result simply in a failed login.

To prepare and execute the data, you would follow these steps:

1. Define your SQL, using question marks as place holders
2. Using the PDO object, prepare the SQL. This will result in a `PDOStatement` object
3. Execute the `PDOStatement` object with an array of the data to replace the question marks
4. If your SQL is a SELECT statement, you will need to examine the results (later).

For example:

```
$sql='SELECT * FROM users WHERE email=? AND passwd=?'; //    SELECT
$sql='INSERT INTO users(email,passwd) VALUES(?,?)';     //    INSERT
$sql='UPDATE users SET email=?, passwd=? WHERE id=?';  //    UPDATE
$sql='DELETE FROM users WHERE id=?';                   //    DELETE

$pds=$pdo->prepare($sql);

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

Note that you do not put the question mark place holders inside quotes even if they represent strings. If you do, the quotes will be added to the data.

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

Remember, preparing your SQL statements is important if your data comes from a user. This is is essential to protect yourself from SQL injection

If there is no user data involved, or if the data is guaranteed be numeric (which could not possibly contain spurious SQL), then you might prefer the direct methods below.

Repeated Execution

Another use of prepared statements is with repeated execution. Whether the data is suspect or not, if you need to execute the same statement many times, it can be more efficient to prepare the statement once, and to execute the prepared statement many times, as in a loop.

For example, suppose you have a number of rows to be inserted, the data for which may already be inside an array. Then you could execute the SQL as follows:

```
$SQL='INSERT INTO products(description) VALUES(?)';

$pds=$pdo->prepare($sql);

foreach($products as $p) {
  $pds->execute(array($p));
}
```

Even if the data isn't suspect, the above code needs to prepare the statement only once, and so the overhead of interpreting, analyzing and optimising the statement is reduced. The multiple executes will run much faster.

Unprepared (Direct) SQL Statements

If there is no risk of malicious user data, then you may not need to prepare your statements first. This will result in slightly simpler code. In this case, you can use one of two PDO functions to run your SQL statement.

SELECT Statements

SELECT Statements expect a result set. In some cases, the result set will have only one row, while in some other cases, the result set may have many.

To get data using an unprepared statement:

1. Define your SQL, including the data. This may include data in variables, if you use a double-quoted string.
2. Using the PDO object, use the **`query()`** function on the SQL statement. This will also result in a `PDOStatement` object, but this will contain the result set if any.
3. In the case of a SELECT statement, you will need to examine the results (later).

For example:

```
$sql='SELECT code,description,price FROM products';
$sql="SELECT code,description,price FROM products WHERE id=$id";

$pds=$PDO->query($sql);
```

The variable `$pds` will contain the result set. It is technically a `PDOStatement` object, though, in this case, does not contain a prepared statement.

The variable `$id` in the second SQL statement above may be subject to SQL injection unless your data has already been tested for this. For example, the PHP `intval()` function will always guarantee an integer, which cannot contain malicious SQL.

INSERT, UPDATE, and DELETE Statements.

INSERT, UPDATE and DELETE statements do not expect a result set. In each case PDO will return a value which is the number of records affected by the SQL statement, but you may choose to ignore this result.

To put data using an unprepared statement:

1. Define your SQL, including the data
2. Using the PDO object, use the exec**`()`** function on the SQL statement. This will return the number of rows affected.

For example:

```
$price=20; $id=3;

$sql="UPDATE products SET price=$price,modified=now() WHERE id=$id";

$PDO->exec($sql);   //    or $rowcount=$PDO->exec($sql);
```

The variable, `$rowcount`, will contain the number of rows affected. Typically for an INSERT statement, or when a WHERE clause has been used to identify a single row, this will be 1. However, it may contain 0 or any other number, depending on the SQL statement.

Again, as above, your variables need to be checked for malicious SQL before including them directly into an SQL statement

PDO Techniques

SELECT Data

To select data from a database table, use the SELECT command:

```
SELECT ... FROM ...;
SELECT ... FROM ... WHERE ...;
```

Prepared Statements

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

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

Unprepared Statements

```
$sql="SELECT ... FROM ... ...";

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

In both cases, you will have a result set in `$pds`.

Fetching Data

To retrieve the data, you can fetch one row at a time, or you can iterate through collection.

To fetch a single row:

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

To iterate through the collection:

```
while($row=$pds->fetch()) {

  ...

}
```

or, more simply:

```
foreach($pds as $row) {

  ...

}
```

The Result Set

Each row in a result set, unless set otherwise, will be an array containing the data *twice*, both with numbered keys, and with associative keys.

For example:

```
SELECT code,description,price FROM products
```

will return rows of the following arrays:

key	**value**
code	…
description	…
price	…
0	…
1	…
2	…

This redundancy will allow you to read the values in a convenient way. For example, to use the row data inside a string, you may wish to use the associative keys:

```
$tr="<tr><th>$row[code]</th><th>$row[description]</th><th>$row[price]</th></tr>";
```

On the other hand, you can use the numeric keys as follows:

```
$code=$row[0];
$description=$row[1];
$price=$row[2];
```

In the above example, you can also use PHP's `list` construct:

```
list($code,$description,$price)=$row;
```

The `list` construct only works with numeric keys.

Variations

The fetched array

As mentioned before, the fetched data will appear as an array with both numeric and associative keys. 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);
```

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.

Fetching a Single Column

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.

Simple PDO Recipes

Count Records

Count All Records

```
$sql='SELECT count(*) FROM ...';
$count=$pdo->query($sql)->fetchColumn();
```

or

```
$count=$pdo->query('SELECT count(*) FROM ...')->fetchColumn();
```

Count Selected Records (Prepared)

```
$sql='SELECT count(*) WHERE ...';
$pds=$pdo->prepare($sql);
$pds->execute(array(...));

$count=$pds->fetchColumn();
```

Login Script

```
$sql='SELECT ... FROM users WHERE email=? AND passwd=?';
$pds=$pdo->prepare($sql);
$pds->execute(array(...));

if($row=$pds->fetch()) {
  // successful; $row now contains the rest of the details
}
else {
  // unsuccessful ($row is FALSE)
}
```

Summary of PDO

Connection

To connect to a database

```
$pdo=new PDO(DSN[,USER,PASSWORD]);
```

Because the default error reporting might give away to much detail, it is normal to include the connection inside a try ... catch block:

```
try {
  $pdo=new PDO(DSN[,USER,PASSWORD]);
}
catch (PDOException $e) {
  //    Handle Error
}
```

Executing Simple Statements

Simple statements include any data directly in the SQL string.

INSERT, UPDATE & DELETE

```
PDO->exec(...);
```

```
$sql="...";
$count=$pdo->exec($sql);
```

The returned value will be the number of rows affected.

SELECT Statements

```
PDO->query(...);
```

```
$sql="SELECT ... FROM ... ";
$result=$pdo-?query($sql);
```

The returned value will be a PDOStatement pointing to the result set. See Reading Data (below) on how to use this.

Executing Prepared Statements

```
PDOStatement=PDO->prepare($sql);
PDOStatement->execute(array(...))
```

Although some data may be included directly in the SQL string, the major benefit from preparing statements is the ability to insert the data after the SQL string has been prepared. In this case you replace the data with question mark place holders; place holders are never to be quoted, even if they are strings.

```
$sql='INSERT into ... VALUES(?,?)';
$pds=$pdo->prepare($sql);
$pds->execute(array(...,...));
```

Reading Data

Whether or not the SQL statement was prepared, the data set will always be in a `PDOStatement`.

Reading a Single Row

Each row is an array containing data with both numeric and associative keys. You may use either (or both) types of key as convenient.

To fetch a single row:

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

This will fetch the *next* row, which may, of course be the first or only row.

If there is no next row (or no result to begin with), `fetch()` will return `FALSE`.

Reading Multiple Rows

To fetch multiple rows:

```
while($row=PDOStatement->fetch() {
  …
}
```

or

```
foreach(PDOStatement as $row) {
  …
}
```

Each will produce exactly the same result. The `foreach` statement is similar to iterating through an array, and automatically fetches the next row and assigns it to `$row`.

Reading a Single Column

For convenience there is a function which will read a single value from a row. This will return a simple value, and avoids having to deal with the data in an array.

```
PDOStatement->fetchColumn([col]);
```

The optional parameter is the number of the column, and defaults to 0, the first column.

This is particularly handy when the result set itself has only one row.

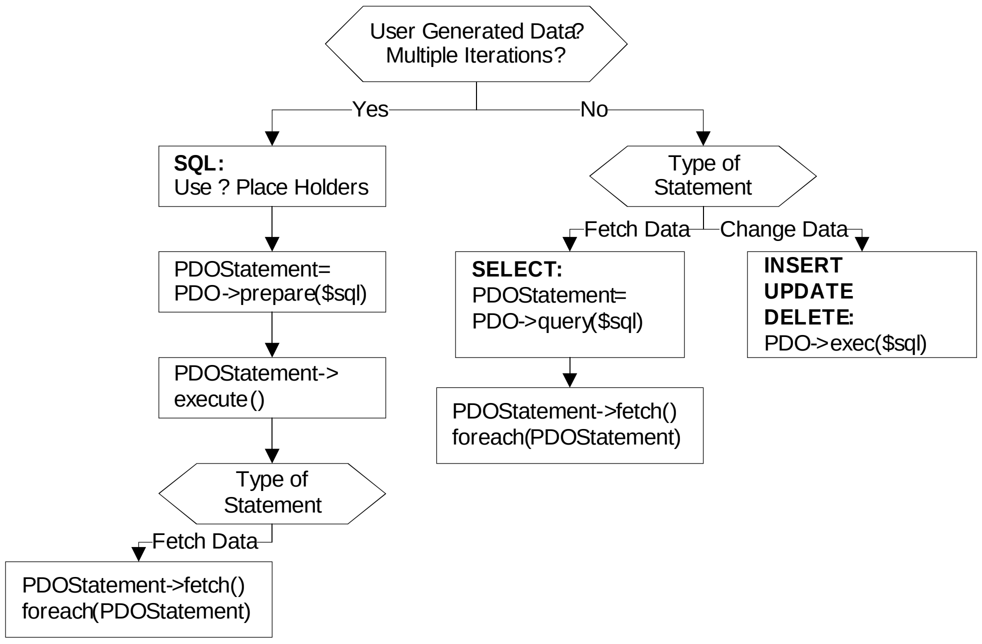
For example, to count the number of records in a table:

```
$sql='SELECT count(*) FROM users';
$result=$pdo->query($sql);
$count=$result->fetchColumn(0);
```

or, more simply,

```
$count=$pdo->query('SELECT count(*) FROM users')->fetchColumn();
```

Summary of Process


User Generated Data?
Multiple Iterations?
Yes
No
SQL:
Use ? Place Holders
PDOStatement=
PDO->prepare($sql)
PDOStatement->
execute()
Type of
Statement
Fetch Data
PDOStatement->fetch()
foreach(PDOStatement)
Type of
Statement
Fetch Data
Change Data
SELECT:
PDOStatement=
PDO->query($sql)
INSERT
UPDATE
DELETE:
PDO->exec($sql)
PDOStatement->fetch()
foreach(PDOStatement)