There are (more than) three popular ways to use MySQL from PHP.

1. (**DEPRECATED**) The [mysql functions](http://www.php.net/manual/en/ref.mysql.php) are procedural and use manual escaping.
2. [MySQLi](http://www.php.net/manual/en/intro.mysqli.php) is a replacement for the mysql functions, with object-oriented and procedural versions. It has support for prepared statements.
3. [PDO](http://php.net/manual/en/book.pdo.php) (PHP Data Objects) is a general database abstraction layer with support for MySQL among many other databases. It provides prepared statements, and significant flexibility in how data is returned.

I would recommend using PDO with prepared statements. It is a well-designed API and will let you more easily move to another database (including any that supports [ODBC](https://en.wikipedia.org/wiki/Open_Database_Connectivity)) if necessary.

Those are different APIs to access a MySQL backend

* The [mysql](http://php.net/manual/en/book.mysql.php) is the historical API
* The [mysqli](http://jp2.php.net/manual/en/book.mysqli.php) is a new version of the historical API. It should perform better and have a better set of function. Also, the API is object-oriented.
* [PDO\_MySQL](http://jp2.php.net/manual/en/ref.pdo-mysql.php), is the MySQL for PDO. PDO has been introduced in PHP, and the project aims to make a common API for all the databases access, so in theory you should be able to migrate between RDMS without changing any code (if you don't use specific RDBM function in your queries), also object-oriented.

So it depends on what kind of code you want to produce. If you prefer object-oriented layers or plain functions...

My advice would be

PDO

MySQLi

Mysql

## Connection

It's a cinch to connect to a database with both of these:

// PDO

$pdo = new PDO("mysql:host=localhost;dbname=database", 'username', 'password');

// mysqli, procedural way

$mysqli = mysqli\_connect('localhost','username','password','database');

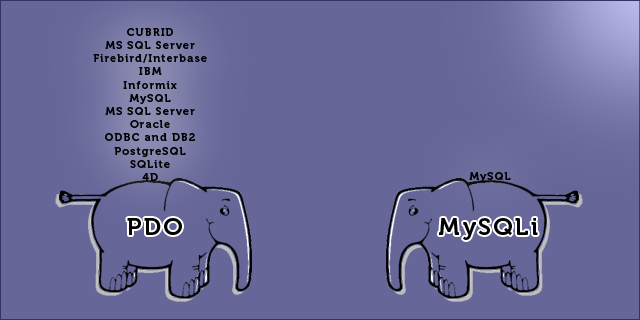
// mysqli, object oriented way

$mysqli = new mysqli('localhost','username','password','database');

## API Support

Both PDO and MySQLi offer an object-oriented API, but MySQLi also offers a procedural API - which makes it easier for newcomers to understand. If you are familiar with the native PHP MySQL driver, you will find migration to the procedural MySQLi interface much easier. On the other hand, once you master PDO, you can use it with any database you desire!

## Database Support

****

The core advantage of PDO over MySQLi is in its database driver support. At the time of this writing, **PDO supports 12 different drivers**, opposed to MySQLi, which supports **MySQL only**.

To print a list of all the drivers that PDO currently supports, use the following code:

|  |  |
| --- | --- |
| 1 | var\_dump(PDO::getAvailableDrivers()); |

What does this mean? Well, in situations when you have to switch your project to use another database, PDO makes the process transparent. So *all you'll have to do* is change the connection string and a few queries - if they use any methods which aren't supported by your new database. With MySQLi, you will need to *rewrite every chunk of code* - queries included.

**Named Parameters**

This is another important feature that PDO has; binding parameters is **considerably easier** than using the numeric binding:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | $params = array(':username' => 'test', ':email' => $mail, ':last\_login' => time() - 3600);    $pdo->prepare('      SELECT \* FROM users      WHERE username = :username      AND email = :email      AND last\_login > :last\_login');    $pdo->execute($params); |

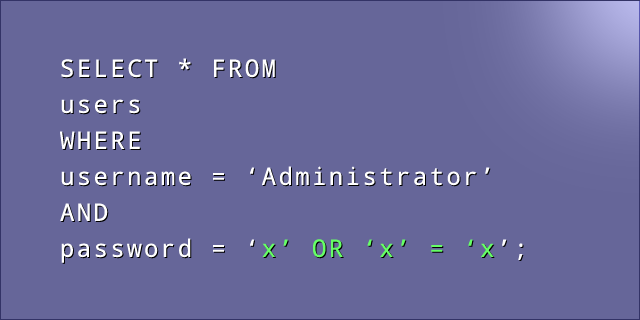
...opposed to the MySQLi way:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | $query = $mysqli->prepare('      SELECT \* FROM users      WHERE username = ?      AND email = ?      AND last\_login > ?');    $query->bind\_param('sss', 'test', $mail, time() - 3600);  $query->execute(); |

The question mark parameter binding might seem shorter, but it isn't nearly as flexible as named parameters, due to the fact that the developer must always *keep track of the parameter order*; it feels "hacky" in some circumstances.

Unfortunately, **MySQLi doesn't support named parameters**.

**Security**



Both libraries provide **SQL injection security, as long as the developer uses them the way they were intended** (read: escaping / parameter binding with prepared statements).

Lets say a hacker is trying to inject some malicious SQL through the 'username' HTTP query parameter (GET):

|  |  |
| --- | --- |
| 1 | $\_GET['username'] = "'; DELETE FROM users; /\*" |

If we fail to escape this, it will be included in the query "as is" - deleting all rows from the users table (both PDO and mysqli support multiple queries).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // PDO, "manual" escaping  $username = PDO::quote($\_GET['username']);    $pdo->query("SELECT \* FROM users WHERE username = $username");    // mysqli, "manual" escaping  $username = mysqli\_real\_escape\_string($\_GET['username']);    $mysqli->query("SELECT \* FROM users WHERE username = '$username'"); |

As you can see, **PDO::quote() not only escapes the string, but it also quotes it.** On the other side, mysqli\_real\_escape\_string() will only escape the string; you will need to apply the quotes manually.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | // PDO, prepared statement  $pdo->prepare('SELECT \* FROM users WHERE username = :username');  $pdo->execute(array(':username' => $\_GET['username']));    // mysqli, prepared statements  $query = $mysqli->prepare('SELECT \* FROM users WHERE username = ?');  $query->bind\_param('s', $\_GET['username']);  $query->execute(); |

I recommend that you always use prepared statements with bound queries instead of PDO::quote() and mysqli\_real\_escape\_string().

Advertisement

**Performance**

While both PDO and MySQLi are quite fast, MySQLi performs insignificantly faster in benchmarks - ~2.5% for non-prepared statements, and ~6.5% for prepared ones. Still, the native MySQL extension is even faster than both of these. So if you truly need to squeeze every last bit of performance, that is one thing you might consider.

**Summary**

Ultimately, PDO wins this battle with ease. With support for twelve different database drivers (eighteen different databases!) and named parameters, we can ignore the small performance loss, and get used to its API. From a security standpoint, both of them are safe as long as the developer uses them the way they are supposed to be used (read: prepared statements).

So if you're still working with MySQLi, maybe it's time for a change!

**Open a Connection to MySQL**

Before we can access data in the MySQL database, we need to be able to connect to the server:

**Example (MySQLi Object-Oriented)**

<?php  
$servername = "localhost";  
$username = "username";  
$password = "password";  
  
// Create connection  
$conn = new mysqli($servername, $username, $password);  
  
// Check connection  
if ($conn->connect\_error) {  
    die("Connection failed: " . $conn->connect\_error);  
}   
echo "Connected successfully";  
?>

**Example (MySQLi Procedural)**

<?php  
$servername = "localhost";  
$username = "username";  
$password = "password";  
  
// Create connection  
$conn = mysqli\_connect($servername, $username, $password);  
  
// Check connection  
if (!$conn) {  
    die("Connection failed: " . mysqli\_connect\_error());  
}  
echo "Connected successfully";  
?>

**Example (MySQLi Object-oriented)**

<?php  
$servername = "localhost";  
$username = "username";  
$password = "password";  
$dbname = "myDB";  
  
// Create connection  
$conn = new mysqli($servername, $username, $password, $dbname);  
// Check connection  
if ($conn->connect\_error) {  
    die("Connection failed: " . $conn->connect\_error);  
}   
  
$sql = "INSERT INTO MyGuests (firstname, lastname, email)  
VALUES ('John', 'Doe', 'john@example.com');";  
$sql .= "INSERT INTO MyGuests (firstname, lastname, email)  
VALUES ('Mary', 'Moe', 'mary@example.com');";  
$sql .= "INSERT INTO MyGuests (firstname, lastname, email)  
VALUES ('Julie', 'Dooley', 'julie@example.com')";  
  
if ($conn->multi\_query($sql) === TRUE) {  
    echo "New records created successfully";  
} else {  
    echo "Error: " . $sql . "<br>" . $conn->error;  
}  
  
$conn->close();  
?>

<?php  
$servername = "localhost";  
$username = "username";  
$password = "password";  
$dbname = "myDB";  
  
// Create connection  
$conn = mysqli\_connect($servername, $username, $password, $dbname);  
// Check connection  
if (!$conn) {  
    die("Connection failed: " . mysqli\_connect\_error());  
}  
  
$sql = "INSERT INTO MyGuests (firstname, lastname, email)  
VALUES ('John', 'Doe', 'john@example.com');";  
$sql .= "INSERT INTO MyGuests (firstname, lastname, email)  
VALUES ('Mary', 'Moe', 'mary@example.com');";  
$sql .= "INSERT INTO MyGuests (firstname, lastname, email)  
VALUES ('Julie', 'Dooley', 'julie@example.com')";  
  
if (mysqli\_multi\_query($conn, $sql)) {  
    echo "New records created successfully";  
} else {  
    echo "Error: " . $sql . "<br>" . mysqli\_error($conn);  
}  
  
mysqli\_close($conn);  
?>

## Prepared Statements and Bound Parameters

A prepared statement is a feature used to execute the same (or similar) SQL statements repeatedly with high efficiency.

Prepared statements basically work like this:

1. Prepare: An SQL statement template is created and sent to the database. Certain values are left unspecified, called parameters (labeled "?"). Example: INSERT INTO MyGuests VALUES(?, ?, ?)
2. The database parses, compiles, and performs query optimization on the SQL statement template, and stores the result without executing it
3. Execute: At a later time, the application binds the values to the parameters, and the database executes the statement. The application may execute the statement as many times as it wants with different values

Compared to executing SQL statements directly, prepared statements have two main advantages:

* Prepared statements reduces parsing time as the preparation on the query is done only once (although the statement is executed multiple times)
* Bound parameters minimize bandwidth to the server as you need send only the parameters each time, and not the whole query
* Prepared statements are very useful against SQL injections, because parameter values, which are transmitted later using a different protocol, need not be correctly escaped. If the original statement template is not derived from external input, SQL injection cannot occur.

## Prepared Statements in MySQLi

The following example uses prepared statements and bound parameters in MySQLi:

### Example (MySQLi with Prepared Statements)

<?php  
$servername = "localhost";  
$username = "username";  
$password = "password";  
$dbname = "myDB";  
  
// Create connection  
$conn = new mysqli($servername, $username, $password, $dbname);  
  
// Check connection  
if ($conn->connect\_error) {  
    die("Connection failed: " . $conn->connect\_error);  
}  
  
// prepare and bind  
$stmt = $conn->prepare("INSERT INTO MyGuests (firstname, lastname, email) VALUES (?, ?, ?)");  
$stmt->bind\_param("sss", $firstname, $lastname, $email);  
  
// set parameters and execute  
$firstname = "John";  
$lastname = "Doe";  
$email = "john@example.com";  
$stmt->execute();  
  
$firstname = "Mary";  
$lastname = "Moe";  
$email = "mary@example.com";  
$stmt->execute();  
  
$firstname = "Julie";  
$lastname = "Dooley";  
$email = "julie@example.com";  
$stmt->execute();  
  
echo "New records created successfully";  
  
$stmt->close();  
$conn->close();  
?>

## Select Data With MySQLi

The following example selects the id, firstname and lastname columns from the MyGuests table and displays it on the page:

**Example (MySQLi Procedural)**

<?php  
$servername = "localhost";  
$username = "username";  
$password = "password";  
$dbname = "myDB";  
  
// Create connection  
$conn = mysqli\_connect($servername, $username, $password, $dbname);  
// Check connection  
if (!$conn) {  
    die("Connection failed: " . mysqli\_connect\_error());  
}  
  
$sql = "SELECT id, firstname, lastname FROM MyGuests";  
$result = mysqli\_query($conn, $sql);  
  
if (mysqli\_num\_rows($result) > 0) {  
    // output data of each row  
    while($row = mysqli\_fetch\_assoc($result)) {  
        echo "id: " . $row["id"]. " - Name: " . $row["firstname"]. " " . $row["lastname"]. "<br>";  
    }  
} else {  
    echo "0 results";  
}  
  
mysqli\_close($conn);  
?>

**Delete Data From a MySQL Table Using MySQLi and PDO**

The DELETE statement is used to delete records from a table:

DELETE FROM table\_name  
WHERE some\_column = some\_value

**Example (MySQLi Procedural)**

<?php  
$servername = "localhost";  
$username = "username";  
$password = "password";  
$dbname = "myDB";  
  
// Create connection  
$conn = mysqli\_connect($servername, $username, $password, $dbname);  
// Check connection  
if (!$conn) {  
    die("Connection failed: " . mysqli\_connect\_error());  
}  
  
// sql to delete a record  
$sql = "DELETE FROM MyGuests WHERE id=3";  
  
if (mysqli\_query($conn, $sql)) {  
    echo "Record deleted successfully";  
} else {  
    echo "Error deleting record: " . mysqli\_error($conn);  
}  
  
mysqli\_close($conn);  
?>

<?php  
$servername = "localhost";  
$username = "username";  
$password = "password";  
$dbname = "myDB";  
  
// Create connection  
$conn = mysqli\_connect($servername, $username, $password, $dbname);  
// Check connection  
if (!$conn) {  
    die("Connection failed: " . mysqli\_connect\_error());  
}  
  
$sql = "INSERT INTO MyGuests (firstname, lastname, email)  
VALUES ('John', 'Doe', 'john@example.com')";  
  
if (mysqli\_query($conn, $sql)) {  
    echo "New record created successfully";  
} else {  
    echo "Error: " . $sql . "<br>" . mysqli\_error($conn);  
}  
  
mysqli\_close($conn);  
?>

# PHP Update Data in MySQL

## Update Data In a MySQL Table Using MySQLi and PDO

The UPDATE statement is used to update existing records in a table:

UPDATE table\_name  
SET column1=value, column2=value2,...  
WHERE some\_column=some\_value

**Example (MySQLi Procedural)**

<?php  
$servername = "localhost";  
$username = "username";  
$password = "password";  
$dbname = "myDB";  
  
// Create connection  
$conn = mysqli\_connect($servername, $username, $password, $dbname);  
// Check connection  
if (!$conn) {  
    die("Connection failed: " . mysqli\_connect\_error());  
}  
  
$sql = "UPDATE MyGuests SET lastname='Doe' WHERE id=2";  
  
if (mysqli\_query($conn, $sql)) {  
    echo "Record updated successfully";  
} else {  
    echo "Error updating record: " . mysqli\_error($conn);  
}  
  
mysqli\_close($conn);  
?>

## 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=utf8mb4', 'username', 'password');

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=utf8mb4', 'username', 'password', array(PDO::ATTR\_EMULATE\_PREPARES => false,

PDO::ATTR\_ERRMODE => PDO::ERRMODE\_EXCEPTION));

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

<?php

$db = new PDO('mysql:host=localhost;dbname=testdb;charset=utf8mb4', 'username', 'password');

$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')") or die("Insert Failed ".mysql\_error());

$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($id), mysql\_real\_escape\_string($name))) or die(mysql\_error());

$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(:field1,:field2,:field3,:field4,:field5)");

$stmt->execute(array(':field1' => $field1, ':field2' => $field2, ':field3' => $field3, ':field4' => $field4, ':field5' => $field5));

$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(?), PASSWORD(?))");

$stmt->execute(array($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();

}