PHP, SQL injection, and Prepared Statements with MySQL

Association of Professional Programmers

Presented by Ryan Conklin

Introduction

- Many web developers, programmers, and coders that have been taught how to create dynamic web pages have been taught in **PHP**.
- **PHP** is similar to the other popular scripting languages, however, it has risen to **being used on roughly 82.1%**[1] **of all webservers** for server side scripting.
- Developers in today's arena should have an understanding of the most up to date best practices to enable them to make conscious and informed decisions on **how to write better code, and above all else more *secure* code**.

1. https://w3techs.com/technologies/overview/programming_language/all

MySQL Databases

- Naturally, in many web applications, there is a need to store persistent data, in which case the developer would rely on the use of **databases**.
  - This is most commonly **accomplished through a MySQL database**.
- The relationship between websites, servers, and the data they share has led to the **close integration of MySQL with PHP in most Linux/Apache based web servers**.

(This includes those on localhost. --> MAMP, WAMP, LAMP, XAMPP)

MySQL Databases contd.

- A big problem with the introductory level of learning PHP and connecting to a database is the ease and simplicity of creating a PHP/SQL soup of code that neither the programmer nor anyone else can ever hope to understand.
- This leads to a huge error that coders make while working with SQL and their databases: **not checking, verifying, and sanitizing user input and ensuring they're using secure SQL code.**
- SQL can be very finicky as it **must be typed syntactically accurate** and without the developer's assistance, **does not take into account special characters** that could be present in the SQL commands.
  - Enter: **SQL Injection**

SQL Injection

- SQL Injection: **Web programmers' and DBAs' worst nightmare.**
- This is where user input that has gone unchecked can be crafted in such a way that additional SQL commands can be entered via user input.
  - can cause really big problems.
- Computerphile: SQL Injection: https://youtu.be/_jKylhJtPmI (8:58)
- Computerphile: How to test for SQL Injection: https://youtu.be/ciNHn38EyRc (17:10)

SQL Injection

For example:

1) Input: ryans_username

```
SELECT * FROM users WHERE username = 'ryans_username' AND email = 'xyz@email.com';
```

This would run fine.

2) Input: ryan's_username

```
SELECT * FROM users WHERE username = 'ryan's_username' AND email = 'xyz@email.com';
```

This would fail.

Can you see why?

However,

3) Input: ryan'; DROP * FROM users;--

```
SELECT * FROM users WHERE username = 'ryan'; DROP * FROM users;-- s_username' AND email = 'xyx@email.com';
```

THIS WOULD ALSO RUN FINE!!

Which would be bad because it would delete all users from the database, even if it didn't find any users with the username 'ryan'.

The old PHP/SQL way

If you have worked with PHP and a MySQL database before, you may be used to:

- Creating a database connection like this:

```
$conn = mysqli_connect('localhost', 'username', 'password', 'database');
```

- Creating a query and executing it like this:

```
$query = "SELECT * FROM users WHERE username = '" . $_POST['txtusername'] . "'";
```

  - Maybe even using something like this:

```
$username = mysqli_real_escape_string($conn,$_POST['txtusername']);
$result = mysqli_query($conn,$query);
```

- Then handling the results using some variation of:

```
$row = mysqli_fetch_array($result);
```

OR

```
$row = mysqli_fetch_assoc($result);
```

OR

```
while($row = mysqli_fetch_array($result)){
    //do code in here for each row in result set.
}
```

The old PHP/SQL way contd.

Again, while this does work, it is completely open to SQL Injection!

Specifically, here:

```
$query = "SELECT * FROM users WHERE username = '" . $_POST['txtusername'] . "'";
```

But this is a very poor way to accomplish the task for a couple of reasons: it can get clunky, reduces code readability, and above all, is very error prone.

Take for example:

```
$query = "INSERT INTO users (email,username,password,address,city,state,zip) VALUES ('"
. $_POST['email'] . "','" . $_POST['username'] . "','" . $hashedpwd . "','" .
$_POST['address'] . "','" . $_POST['city'] . "','" . $_POST['state'] . "'," .
$_POST['zip'] . ")";
```

And while this helps:

```
$username = mysqli_real_escape_string($conn,$_POST['txtusername']);
```

or

```
$query = sprintf("SELECT * FROM table WHERE id='%s' AND name='%s'",
mysqli_real_escape_string($conn,$id), mysqli_real_escape_string($conn,$name));
```

It has flaws.

- Isn't even supported in some versions of php and depends on the mysql driver it was built with.
- SQL injection can still be achieved in some scenarios.
- Is also very clunky and cumbersome.

Enter PDO: (PHP Data Objects)

PDO:

- Defines a lightweight, consistent **interface for accessing databases in PHP**.
- Each database driver that implements the PDO interface can expose database-specific features as regular extension functions.
- PDO provides a *data-access* abstraction layer.
  - Regardless of which database you're using, you **use the same functions to issue queries and fetch data**.
- Requires the new OO (**Object Oriented**) and PDO features in the core of **PHP 5+**, and so **will not run with earlier versions of PHP**. (A reason you should be updated!)

Getting Started with PDO for MySQL

The first thing to remember here is that the **PDO interface is Object Oriented PHP** and **not Procedural**, however, **PHP allows for jumping in and out of both "on-the-fly"**.

To begin, setup a new PDO object that connects to the mysql driver.

```
$conn = new PDO('mysql:host=localhost;dbname=database;charset=utf8',
'username', 'password');
```

Then set a few additional parameters on the PDO object.

```
$conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
$conn->setAttribute(PDO::ATTR_EMULATE_PREPARES, false);
```

TIP: using -> is the OO way of accessing public object properties and methods in PHP. This is similar to how you would access functions in other OO languages like Java or C# using String.charAt(0), except the . is reserved for concatenating in PHP.

Setting up PDO further explained

Take note to these two lines of code.

```
$conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
```

This is used for **setting the way PHP should fail** if there is a problem with using the database.

- **PDO::ERRMODE_SILENT** acts like mysql_error($conn) where you must look to get the error details.
- **PDO::ERRMODE_WARNING** throws PHP Warnings.
- **PDO::ERRMODE_EXCEPTION** throws PDOException. Acts like die(mysql_error()); when it isn't caught, but unlike die() the PDOException can be caught using try/catch and handled gracefully if desired.

```
$conn->setAttribute(PDO::ATTR_EMULATE_PREPARES, false);
```

This is used to turn off prepared statement "emulation mode", which is turned on be default (why!?) in the recent MySQL drivers, and is only for use with older versions anyway. **PLEASE ALWAYS TURN THIS OFF, WE WANT THE REAL THING HERE.**

Querying using prepared statements method 1

Next were going to actually prepare a SQL statement.

```
$query = "SELECT username,email,password FROM users WHERE id=? AND email=?";
NOTE:(use ? to represent where your inputs and variables will be!)

$stmt = $db->prepare($query);
```

That's it. Literally.

Next you will execute the query. This is done by passing an array of the values you want to put in place of the **?** in the same order.

```
$stmt->execute(array($userid,$txtemail));
```

Alternatively, this can be done by binding parameters to values using their index (starting at 1) and specifying their type.

```
$stmt->bindValue(1, $userid, PDO::PARAM_INT);
$stmt->bindValue(2, $txtemail, PDO::PARAM_STR);
$stmt->execute();
```

Querying using prepared statements method 2

This can also be achieved by using a named placeholder.

```
$query = "SELECT username,email,password FROM users WHERE id=:id AND email=:email";
$stmt = $db->prepare($query);
```

That's it. Literally.

Next you will execute the query. This is done by passing an associative array of the values you want to put in place of the named placeholders.

```
$stmt->execute(array(':id'=>$userid, ':email'=>$txtemail));
```

Alternatively, this can be done by binding parameters to values using their named placeholder and specifying their type.

```
$stmt->bindValue(':id', $userid, PDO::PARAM_INT);
$stmt->bindValue(':email', $txtemail, PDO::PARAM_STR);
$stmt->execute();
```

Getting the results of the query

Finally, we want the rows back from the query.

Rows can be returned by using either: `fetch` or `fetchAll` and passing the type of keyed array we want back for accessing the values.

Types include:

- PDO::FETCH_ASSOC – this will return an associative array using the column names as the keys. (i.e. `$row['username']`)
- PDO::FETCH_NUM – this will return a numerical array using indices for the columns as keys (i.e. `$row[0]`)

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

OR

```
$row = $stmt->fetch(PDO::FETCH_ASSOC);
$row = $stmt->fetch(PDO::FETCH_NUM);
```

NOTE: If you do not specify either PDO::FETCH_ASSOC or PDO::FETCH_NUM, it will be returned back to you with both keys, doubling the size of your result set!! This could be useful, but typically, especially with larger datasets, you will not want this.

Getting the results of the query

At this point you should be familiar with what to do with the results given back to you from executing the query.

Accessing the values from each row using:

```
$username = $row['username'];
```

Or

```
$username = $row[0];
```

Depending on which array type was specified to be generated.

Other SQL query types

It should also be noted that in addition to **SELECT**, other SQL queries such as **INSERT**, **UPDATE**, and **DELETE** also work in the same ways.

INSERT

```
$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));
```

UPDATE

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

DELETE

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

Special SQL functions

Also note that in order to use certain SQL functions such as **NOW()**, **DATE()**, **SHA256()**, **MD5()** etc.. They need to go directly into the query. For example:

```
$query = "INSERT INTO messages (time,message) VALUES(NOW(),?)";
$stmt = $db->prepare($query);
$stmt->execute(array($msg));
```

So trying to do:

```
$time = "NOW()";
$msg = "testing";
$query = "INSERT INTO messages (time,message) VALUES(?,?)";
$stmt = $db->prepare($query);
$stmt->execute($time,$msg);
```

WILL NOT WORK

Special SQL regex

Again note that in order to use certain SQL regular expressions such as **`LIKE`** etc.. They need to **NOT** go directly into the query. For example:

```
$term = "%javascri%";
$query = "SELECT * FROM languages WHERE language_name LIKE ?";
$stmt = $db->prepare($query);
$stmt->execute($term);
```

So trying to do:

```
$term = "javascri";
$query = "SELECT * FROM languages WHERE language_name LIKE %?%";
$stmt = $db->prepare($query);
$stmt->execute(array($term));
```

WILL NOT WORK

My custom PHP PDO_MySQL class

To simplify things, I have created a custom implementation of MySQL PDO class. It aims to reduce the amount of new syntax and commands one needs to familiarize themselves with by providing a streamlined way to connect and query a MySQL database.

The methods are similar in syntax to the mysqli_* methods that many people are already accustomed to.

It uses the PDO class in the backend, so prepared statements are performed and variable binding is taken care of automatically.

How to use the custom PDO_MySQL

Quick example.

Instead of:

```
$conn = new PDO('mysql:host=localhost;dbname=database;charset=utf8',
'username', 'password');
$conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
$conn->setAttribute(PDO::ATTR_EMULATE_PREPARES, false);
```

This class allows:

```
require('pdo_mysql_class.php');
$conn = new PDO_MySQL();
$conn->connect('localhost', 'username', 'password', 'database');
```

How to use the custom PDO_MySQL

From there it gets even simpler:

Instead of:

```
$query = "SELECT username,email,password FROM users WHERE id=? AND email=?";
$stmt = $db->prepare($query);
$stmt->execute(array($userid,$txtemail));
$stmt->bindValue(1, $userid, PDO::PARAM_INT);
$stmt->bindValue(2, $txtemail, PDO::PARAM_STR);
$stmt->execute();
$result = $stmt->fetchAll(PDO::FETCH_ASSOC);
```

This class allows:

```
$query = "SELECT username,email,password FROM users WHERE id=? AND email=?";
$values = array($userid,$txtemail);
$result = $conn->query_prepared($query,$values,"ASSOC");
```

As you can see, this automatically takes care of preparing the SQL statement, binding the variables, and allows the developer to choose type of keys to be returned in the result set.

How to use the custom PDO_MySQL

Then getting the data out is just as simple:

Instead of:

```
$row = $stmt->fetch(PDO::FETCH_ASSOC);
```

This class uses:

```
$row = $conn->fetch_row($result);
```

And this can even be used like:

```
while($row = $conn->fetch_row($result)){
    //Do work in here for each row
}
```

How to use the custom PDO_MySQL

Other useful functions included with this class:
change_database
change_host
change_user
create_connection
field_names
has_rows
is_connected
num_rows
pdo_self
query
reconnect
set_testing
set_err_mode
set_err_mode_default

How to use the custom PDO_MySQL

The PDO_MySQL class along with full documentation on how to use all the functions can be found on GitHub at:

https://github.com/ryanc16/PHP-PDO_MySQL