Home Discussions

'MySQLi' for Beginners

Introduction

Painless Time Tracking
HARVEST

Painlessly Track Time
Time tracking for creative professionals.
via Ad Packs

Nearly every site that you visit nowadays has some sort of database storage in place, many sites opt to use MySQL databases when using PHP. However, many people haven't yet taken the step to interacting with databases properly in PHP. Here we guide you through what you should be doing - using PHP's MySQLi class - with a hat-tip to the one way that you **definitely** shouldn't be doing it.

The Wrong Way

If you're using a function called `mysql_connect()` or `mysql_query()` you really need to take note and change what you're doing. I understand that it's not easy to change current large projects, but look to change future ones.

Any of the functions that are prefixed with `mysql_` are now being discouraged by PHP themselves as visible on **this doc page**, instead you should look to use one of the following:

- **MySQLi** - The *i* standing for 'improved'.
- **PDO**

Each has its advantages, PDO for example will work with various different database systems, where as MySQLi will only work with MySQL databases. Both are object oriented, but MySQLi allows procedural usage as well. There are other minor differences between the two, but it's up to you to choose which you want to use, here we'll be looking at MySQLi.

PHP MySQLi

Here we'll mostly be looking at the **object oriented** implementation, however, there is no reason you can't use this in a procedural format, but again no reason you shouldn't use the OO implementation.

Connecting

Connecting is as simple as just instantiating a new instance of MySQLi, we'll be using a username of `user` with a password of `pass` connecting to the `demo` database on the `localhost` host:

```
$db = new mysqli('localhost', 'user', 'pass', 'demo');

if($db->connect_errno > 0){
    die('Unable to connect to database [' . $db->connect_error . ']');
}
```

Obviously, the database name is optional and can be omitted. If you omit the database name you must be sure to prefix your tables with the database in all of your queries.

Querying

Let's go ahead and pull out all of the users from the `users` table where they have `live = 1`:

```
$sql = <<<SQL
    SELECT *
    FROM `users`
    WHERE `live` = 1
SQL;

if(!$result = $db->query($sql)){
    die('There was an error running the query [' . $db->error . ']');
}
```

We now have a variable that contains a `mysqli_result` object, we can now go ahead and do various things with this such as looping through the results, displaying how many there are and freeing the result.

Output query results

To loop through the results and output the `username` for each row on a new line we'd do the following:

```
while($row = $result->fetch_assoc()){
    echo $row['username'] . '<br />';
}
```

As you can see from this, the syntax isn't too dissimilar to the old `mysql_` syntax that you're probably used to, this is just better and improved!

Number of returned rows

Each `mysqli_result` object that is returned has a variable defined which is called `$num_rows`, so all we need to do is access that variable by doing:

```
<?php
echo 'Total results: ' . $result->num_rows;
?>
```

Number of affected rows

When running an `UPDATE` query you sometimes want to know how many rows have been updated, or deleted if running a `DELETE` query, this is a variable which is inside the `mysqli` object.

```
<?php
echo 'Total rows updated: ' . $db->affected_rows;
?>
```

Free result

It's advisable to free a result when you've finished playing with the result set, so in the above example we should put the following code after our `while()` loop:

```
$result->free();
```

This will free up some system resources, and is a good practice to get in the habit of doing.

Escaping characters

When inserting data into a database, you'll have been told (I hope) to escape it first, so that single quotes get preceeded be a backslash. This will mean that any quotes won't break out of any that you use in your SQL. This is still the case - and you should look to use the below method:

```
$db->real_escape_string('This is an unescaped "string"');
```

However, because this is a commonly used function, there is an alias function that you can use which is shorter and less to type:

```
$db->escape_string('This is an unescape "string"');
```

This string should now be safer to insert into your database through a query.

Close that connection

Don't forget, when you've finished playing with your database to make sure that you close the connection:

```
$db->close();
```

Prepared Statements

Prepared statements are complex to get your head around, but are really useful and can help alleviate a lot of the potential issues that you might have with escaping. Prepared statements basically work by you playing a `?` where you want to substitute in a `string`, `integer`, `blob` or `double`. Prepared statements don't substitute the value into the SQL so the issues with SQL injections are mostly removed.

Define a statement

Let's try to grab all of the users from the `users` table where they have a username of `bob`. We'd firstly define the SQL statement that we'd use:

```
$statment = $db->prepare("SELECT `name` FROM `users` WHERE `username` = ?");
```

That question mark there is what we're going to be assigning the word 'bob' to.

Bind parameters

We simply use the method `bind_param` to bind a parameter. You must specify the type as the first parameter then the variable as the second - so for instance we'd use `s` as the first parameter (for string), and our `$name` variable as the second:

```
$name = 'Bob';
$statement->bind_param('s', $name);
```

If we had 3 parameters to bind which are of varying types we could use `bind_param('sdi', $name, $height, $age);` for example. Note the types are not separated at all as the first parameter.

Execute the statement

No fuss, no mess, just execute the statement so that we can play with the result:

```
$statement->execute();
```

Iterating over results

Firstly we'll bind the result to variables, we do this using the `bind_result()` method which allow us specify some variables to assign the result to. So if we assign the returned `name` to the variable `$returned_name` we'd use:

```
$statement->bind_result($returned_name);
```

As before, if you have multiple variables to assign, just comma separate them - simple as that.

Now we have to actually fetch the results, this is just as simple as the earlier mysqli requests that we were doing - we'd use the method `fetch()`, which returns will assign the returned values into the binded variables - if we'd binded some.

```
while($statement->fetch()){
    echo $returned_name . '<br />';
}
```

Close statement

Don't forget to forgo a few seconds of your time to free the result - keep your code neat, clean and lean:

```
$statement->free_result();
```

MySQLi Transactions

One of the major improvements that MySQLi brings is the ability to use transactions. A transaction is a group of queries that execute but don't save their effects in the database. The advantage of this is if you have 4 inserts that all rely on each other, and one fails, you can roll back the others so that none of the data is inserted, or if updating fields relies on fields being inserted correctly.

You need to ensure that the database engine that you're using supports transactions.

Disable auto commit

Firstly you need to make it so that any query you submit doesn't automatically commit in the database. It's a simple one line boolean value:

```
$db->autocommit(FALSE);
```

Commit the queries

After a few queries that you've ran using `$db->query()` we can call a simple function to commit the transaction:

```
$db->commit();
```

Pretty simple stuff so far, and it's meant to be easy and approachable so that you have no reason to not use it.

Rollback

Just as easy as it is to commit something, it's just as simple to roll something back:

```
$db->rollback();
```

Take a look at the **PHP documentation** for an example of how to use rollbacks. I personally haven't found a scenario where I would use them, but they're worth knowing about so that you are aware they're there to be used.

Final Thoughts

Using `mysql_` functions is a foolish move to make, don't use these outdated and useless methods because they're *easier*, or *quicker*. Man up and tackle one of the new forms of database interaction - MySQLi or PDO - you'll make **@mfrost503 happier**, and have better code too.

Tags: **PHP**, **MySQL**

Top

Tweet

Posted 16th July 2012 by **Michael**