MySQL PDO PHP Security

PHP PDO Prepared Statements Tutorial to Prevent SQL Injection

Nov 26, 2017

Table of Contents

What's PDO?

PDO (PHP Data Objects) is an abstraction layer for your database queries and is an awesome alternative to MySQLi, as it supports 12 different database drivers (http://php.net/manual/en/pdo.drivers.php#pdo.drivers). This is an immense benefit for people and companies that need it. However, keep in mind that MySQL is by far the most popular database (https://insights.stackoverflow.com/survey/2018/#technology-databases). It's also exceedingly tightly coupled with PHP, which is why that number is significantly higher within the PHP world, as PHP and MYSQL are like peanut butter and jelly. NoSQL is a different story, and Firebase and MongoDB are excellent choices, especially the former, as it's a live database — both are obviously not supported in PDO anyway.

Note: For this tutorial, I will be showing non-emulated (native) PDO prepared statements strictly with MySQL, so there might be some differences on a different driver.

If you know for a fact that the only SQL databases you'll be using are either MySQL or MariaDB, then you can choose between PDO or MySQLi. Check out the following tutorial, If you'd like to learn MySQLi (https://websitebeaver.com/prepared-statements-in-php-mysqli-to-prevent-sql-injection). Either one of these is perfectly acceptable to use, though PDO is the better choice for most users, as it's simpler and more versatile, while MySQLi is sometimes more suitable for advanced users, due to a few of its MySQL-specific features.

A lot of people regurgitate that the main advantage of PDO is that it's portable from database-to-database. This is an extremely overstated benefit and is essentially nonsense. SQL is not meant to be transferred this way, as each DB driver has its own nuances; plus, how often are you really making decisions to switch databases on a specific project, unless you're at least a mid-level - large company? Even so, as a rule of thumb, it's generally preferred to stick with the current technology you're using, unless there's a justifiable reason to lose a variable amount of time (money) to do it.

The true advantage of PDO is the fact that you're using a virtually similar API for any of the myriad of databases it supports, so you don't need to learn a new one for each. Named parameters are also undoubtedly a huge win for PDO, since you can reuse the same values in different places in the queries. Unfortunately, you can't use the same named parameters more than once with emulation mode turned off, therefore making it useless for the sake of this tutorial.

A controversial advantage of PDO is the fact that you don't need to use `bindParam()` (http://php.net/manual/en/pdostatement.bindparam.php) nor `bindValue()` (http://php.net/manual/en/pdostatement.bindvalue.php), since you can simply pass in the values as arrays directly into execute. Some might argue that this is considered bad practice, as you can't specify the type (string, int, double, blob); everything will be treated as a string and gets converted to the correct type automagically. In practice, this shouldn't affect your ints or doubles, and is safe from SQL injection. This article will bind values directly into execute. Similar to `bindValue()` , you can use both values and variables. We won't be covering the two bind methods, but if you'd like to know a subtle difference between the two, read this part of the article.

If you'd like to learn how SQL injection works, you can read about it here (prepared-statements-in-php-mysqli-to-prevent-sql-injection#how-sql-injection-works).

How PDO Prepared Statements Work

In layman's terms, PDO prepared statements work like this:

1. Prepare an SQL query with empty values as placeholders with either a question mark or a variable name with a colon preceding it for each value
2. Bind values or variables to the placeholders
3. Execute query simultaneously

Creating a New PDO Connection

I recommend creating a file named `pdo_connect.php` and place it **outside** of your root directory (ex: html, public_html).

```
$dsn = "mysql:host=localhost;dbname=myDatabase;charset=utf8mb4";
$options = [
  PDO::ATTR_EMULATE_PREPARES   => false, // turn off emulation mode for "real" pr
  PDO::ATTR_ERRMODE            => PDO::ERRMODE_EXCEPTION, //turn on errors in the
  PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC, //make the default fetch be a
];
try {
  $pdo = new PDO($dsn, "username", "password", $options);
} catch (Exception $e) {
  error_log($e->getMessage());
  exit('Something weird happened'); //something a user can understand
}
```

So what's going on here? The first line is referred to as DSN (http://php.net/manual/en/pdo.construct.php#refsect1-pdo.construct-parameters) and has three separate values to fill out, your hostname, database and charset. This is the recommended way to do it, and you can obviously set your charset to whatever your application needs (though utf8mb4 is pretty standard). Now you can pass in your DSN info, username, password and options.

Alternatively, you can omit using a `try/catch` block by creating a global custom exception handler. If this is included on all your pages, then it will use this custom handler, unless you do restore_exception_handler() (http://php.net/manual/en/function.restore-exception-handler.php) to revert back to the built-in PHP exception handler or call set_exception_handler() (http://php.net/manual/en/function.set-exception-handler.php) with a new function and custom message.

```
set_exception_handler(function($e) {
  error_log($e->getMessage());
  exit('Something weird happened'); //something a user can understand
});
$dsn = "mysql:host=localhost;dbname=myDatabase;charset=utf8mb4";
$options = [
  PDO::ATTR_EMULATE_PREPARES   => false, // turn off emulation mode for "real" pr
  PDO::ATTR_ERRMODE            => PDO::ERRMODE_EXCEPTION, //turn on errors in the
  PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC, //make the default fetch be a
];
$pdo = new PDO($dsn, "username", "password", $options);
```

This is extremely debatable, but one thing I like about MySQLi is that error reporting is turned off by default. This is smart, so a beginner wouldn't accidentally print out his password. In PDO, even though you you have control to silence errors, you can't do this for the constructor (http://php.net/manual/en/pdo.construct.php#refsect1-pdo.construct-errors). So obviously you should first set up your php.ini for production.

> To prevent leaking your password, here's what your php.ini file should look like in production: do both `display_errors = Off` and `log_errors = On`. Then restart Apache or Ngnix

Now all errors on your site will solely accumulate in your error log, instead of printing them out.

Still, I don't see a reason to print out your password in your error log, so **I'd recommend doing `try/catch` or `set_exception_handler`, while doing `error_log($e->getMessage())` , not `$e`, which would still contain your sensitive information**. This obviously exclusively applies to when you create a new connection.

Named Parameters

I really love this feature, and it's a huge advantage for PDO. You specify a variable named `:id` and give it its value on execute. Though as stated earlier, its only advantage of being used multiple times is rendered useless if emulation mode is turned off.

```
$stmt = $pdo->prepare("UPDATE myTable SET name = :name WHERE id = :id");
$stmt->execute([':name' => 'David', ':id' => $_SESSION['id']]);
$stmt = null;
```

You technically don't need the leading colon on `id` for the execute part, as stated here (https://stackoverflow.com/a/38925175). However, this isn't explicitly stated anywhere in the docs, so while it should work as some users have astutely concluded by looking in the C code, it is not *technically* recommended. My hunch is that PHP will document this eventually anyway, since it seems like there are enough people who omit the leading colon.

I dedicated a section to using named parameters, since the rest of the post will be using `?` instead. Keep in mind that you can't mix both together when binding values.

Insert, Update and Delete

All of these are extremely similar to each other, so they will be combined.

Insert

```
$stmt = $pdo->prepare("INSERT INTO myTable (name, age) VALUES (?, ?)");
$stmt->execute([$_POST['name'], 29]);
$stmt = null;
```

Update

You can even chain `prepare()` and `execute()`. Though you won't be able to use any functions, like `rowCount()`, so it's pretty much useless in practice.

```
$stmt = $pdo->prepare("UPDATE myTable SET name = ? WHERE id = ?")->execute([$_POS
$stmt = null;
```

Delete

```
$stmt = $pdo->prepare("DELETE FROM myTable WHERE id = ?");
$stmt->execute([$_SESSION['id']]);
$stmt = null;
```

Get Number of Affected Rows

Getting the number of affected rows is exceedingly simple, as all you need to do is `$stmt->rowCount()`. Normally if you update your table with the same values, it'll return 0. If you'd like to change this behavior, then the only way to do this is by globally adding this option when you create a new connection `PDO::MYSQL_ATTR_FOUND_ROWS => true`.

```
$stmt = $pdo->prepare("UPDATE myTable SET name = ? WHERE id = ?");
$stmt->execute([$_POST['name'], $_SESSION['id']]);
echo $stmt->rowCount();
$stmt = null;
```

-1 - Query returned an error. Redundant if there is already error handling for execute()

0 - No records updated on UPDATE, no rows matched the WHERE clause or no query been executed; just rows matched if `PDO::MYSQL_ATTR_FOUND_ROWS => true`

Greater than 0 - Returns number of rows affected; rows matched if `PDO::MYSQL_ATTR_FOUND_ROWS => true`. Can be used to get number of rows in SELECT if the database driver supports it, which MySQL does

Get Latest Primary Key Inserted

```
$stmt = $pdo->prepare("INSERT INTO myTable (name, age) VALUES (?, ?)");
$stmt->execute([$_POST['name'], 29]);
echo $pdo->lastInsertId();
$stmt = null;
```

Check if Duplicate Entry

Sometimes you might need to enforce a unique value for one or more columns. You obviously could simply to a SELECT statement to check if there's already a row with the values attempted to be inserted. It's not necessarily wrong to do this, but it doesn't make sense to do an extra database query, when you could easily just check the error message.

You can either check for the SQLSTATE or the vendor-specific error. For a duplicate entry on a unique constaint The SQLSTATE is **23000**, while the MySQL error code is **1062**. Here's a nice reference (https://dev.mysql.com/doc/refman/5.5/en/error-messages-server.html#error_er_dup_entry) for a list of errors. To get the SQLSTATE, you can either use `$e->getCode()` or `$e->errorInfo[0]`; to get the MySQL error code, you must do `$e->errorInfo[1]`. *Note, the behavior of `$e->getCode()` is the opposite of MySQLi, which will print the MySQL-specific error code*. Also, don't use `PDO::errorCode` (http://php.net/manual/en/pdo.errorcode.php) or `PDO::errorInfo` (http://php.net/manual/en/pdo.errorinfo.php). Stick with the PDOException class, as for some reason, the PDO class error methods just print out **00000**.

```
try {
  $stmt = $pdo->prepare("INSERT INTO myTable (name, age) VALUES (?, ?)");
  $stmt->execute([$_POST['name'], 29]);
  $stmt = null;
} catch(Exception $e) {
  if($e->errorInfo[1] === 1062) echo 'Duplicate entry';
}
```

In case you were wondering, you can create a unique constraint by doing:

```
ALTER TABLE myTable ADD CONSTRAINT unique_person UNIQUE (name, age)
```

Select

To fetch results in PDO, you have the option of `$stmt->fetch()` or `$stmt->fetchAll()`. The former is more versatile, as it can be used to fetch one row, or all if used in a loop. The latter is basically syntactic sugar, as it lets fetch your entire result set in an array with that one command. It is preferred to use `$stmt->fetch()` in a loop if you are modifying that array, as it saves you from having to "re-loop" it. You also can use `$stmt->setFetchMode()` (http://php.net/manual/en/pdostatement.setfetchmode.php) to change the default fetch mode, rather than passing it into `fetch()` or `fetchAll()`. But this is just a wasted extra line, and should only be done in cases where it's required.

The fetch modes in PDO are easily my favorite aspect. I will be mixing them into my examples, but here are some of the constants I find to be the be the most useful.

> Note: some of these fetch modes use a bitwise operator (http://php.net/manual/en/language.operators.bitwise.php), like **|**. In the case of PDO, you can essentially think of it as combining fetch modes. This is referred to an **inclusive or** and is the only bitwise operator you need to worry about.

Fetch Modes

Both fetch() and fetchAll()

- **`PDO::FETCH_NUM`** - Fetch a numeric array
- **`PDO::FETCH_ASSOC`** - Fetch an associative array
- **`PDO::FETCH_COLUMN`** - Fetch just one column. Scalar if `fetch()` is used; 1d array if `fetchAll()` is used. If `fetch()`, then you can use `fetchColumn()`

(http://php.net/manual/en/pdostatement.fetchcolumn.php) for syntactic sugar instead

- **`PDO::FETCH_CLASS | PDO::FETCH_CLASSTYPE`** - Fetches an object with a class name according to the database value. If class doesn't exist, will default to **stdClass**

Just fetch()

- **`PDO::FETCH_OBJ`** - Fetch as a generic stdClass Object. It's advisable to use the `fetchObject()` (http://php.net/manual/en/pdostatement.fetchobject.php) syntactic sugar instead, as it allows you fetch an anonymous and specified class. You can still use `PDO::FETCH_CLASS`, but you must set it with `setFetchMode(PDO::FETCH_CLASS, 'MyClass')`, followed by `fetch()`. Can be used with `fetchAll()`, but not preferred, as it's less versatile
- **`PDO::FETCH_LAZY`** - Fetches a **PDORow** reference that has the ability to be an object, associative or numeric array. Is memory-efficient, as it will strictly fetch the results when an object, associative or numeric index is called
- **`PDO::FETCH_INTO`** - Fetch into an existing class. Must use `setFetchMode(PDO::FETCH_INTO, $myClass)`, followed by `fetch()`

Just fetchAll()

- **`PDO::FETCH_CLASS`** - Fetch as either a generic stdClass Object or into an existing class if specified; the same as using `PDO::FETCH_OBJ` if anonymous object, so it makes more sense to use this, as it's more flexible. Can be used with `fetch()` as well, but isn't preferred.
- **`PDO::FETCH_KEY_PAIR`** - Fetch a key/value pair with the first column as a unique key and second one as the single value
- **`PDO::FETCH_UNIQUE`** - Same as `PDO::FETCH_KEY_PAIR`, only the value part is an array
- **`PDO::FETCH_GROUP`** - Fetch by a common column name and group all rows to that key as an array of associative arrays
- **`PDO::FETCH_GROUP | PDO::FETCH_COLUMN`** - Is the same as `PDO::FETCH_GROUP`, only the value part is an array of a 1d numeric arrays
- **`PDO::FETCH_GROUP | PDO::FETCH_CLASS`** - Same as `PDO::FETCH_GROUP`, but with an array of object arrays instead

Fetch Associative Array

Since we set the default fetch type to be an associative array, we don't have specify anything when fetching results.

```
$stmt = $pdo->prepare("SELECT * FROM myTable WHERE id <= ?");
$stmt->execute([5]);
$arr = $stmt->fetchAll(PDO::FETCH_ASSOC);
if(!$arr) exit('No rows');
var_export($arr);
$stmt = null;
```

There's also the slightly longer while loop version, which is sometimes handy for manipulations.

```
$arr = [];
$stmt = $pdo->prepare("SELECT * FROM myTable WHERE id <= ?");
$stmt->execute([5]);
while ($row = $stmt->fetch(PDO::FETCH_ASSOC)) {
  $arr[] = $row;
}
if(!$arr) exit('No rows');
var_export($arr);
$stmt = null;
```

Output:

```
[
  ['name' => 'Jerry', 'age' => 14, 'weight' => 129],
  ['name' => 'Alexa', 'age' => 22, 'weight' => 108]
]
```

Fetch Numeric Array

```
$stmt = $pdo->prepare("SELECT first_name, squat, bench_press FROM myTable WHERE w
$stmt->execute([200]);
$arr = $stmt->fetchAll(PDO::FETCH_NUM);
if(!$arr) exit('No rows');
var_export($arr);
$stmt = null;
```

Output:

```
[
  ['Thad', 305, 250],
  ['Larry', 225, 180]
]
```

Fetch Array of Objects

Similar to fetching an associative array, but with objects, so you could access it like, `$arr[0]->age` for instance.

```
$stmt = $pdo->prepare("SELECT name, age, weight FROM myTable WHERE id > ?");
$stmt->execute([12]);
$arr = $stmt->fetchAll(PDO::FETCH_CLASS);
if(!$arr) exit('No rows');
var_export($arr);
$stmt = null;
```

Output:

```
[
  stdClass Object ['name' => 'Jerry', 'age' => 14, 'weight' => 129],
  stdClass Object ['name' => 'Alexa', 'age' => 22, 'weight' => 108]
]
```

You can even append property values to an already existing class, like so.

```
class myClass {}
$stmt = $pdo->prepare("SELECT name, age, weight FROM myTable WHERE id > ?");
$stmt->execute([12]);
$arr = $stmt->fetchAll(PDO::FETCH_CLASS, 'myClass');
if(!$arr) exit('No rows');
var_export($arr);
$stmt = null;
```

Output:

```
[
  myClass Object ['name' => 'Jerry', 'age' => 14, 'weight' => 129],
  myClass Object ['name' => 'Alexa', 'age' => 22, 'weight' => 108]
]
```

Keep in mind that this has unpredictable behavior of injecting the property value *before* setting it in the constructor (if you have one). This means that if you already used one of the variable names in the constructor, then the fetch value will get overwritten by default value. This behavior is noted here (http://php.net/manual/en/pdostatement.fetchall.php#97973). To ensure the values are assigned *after* the constructor is called, you must do `fetchAll(PDO::FETCH_CLASS | PDO::FETCH_PROPS_LATE, 'myClass')`.

Another unexpected, yet potentially useful behavior this has is that you can modify private variables. I'm really not sure how I feel about this, as this seems to violate principles of encapsulation.

Fetch Single Row

```
$stmt = $pdo->prepare("SELECT id, name, age FROM myTable WHERE name = ?");
$stmt->execute([$_POST['name']]);
$arr = $stmt->fetch();
if(!$arr) exit('No rows');
var_export($arr);
$stmt = null;
```

Now you access each variable, like `$arr['name']` for instance.

Output:

```
['id' => 645, 'name' => 'Joey', 'age' => 28]
```

Fetch Single Row Like MySQLi bind_result()

```
$stmt = $pdo->prepare("SELECT id, name, age FROM myTable WHERE id = ?");
$stmt->execute([3]);
$arr = $stmt->fetch(PDO::FETCH_NUM); //FETCH_NUM must be used with list
if(!$arr) exit('no rows');
list($id, $name, $age) = $arr;
echo $name; //Output: 'Jeremy'
$stmt = null;
```

This is to mimic the (only beneficial) behavior of `bind_result()` in MySQLi, which is to be able to bind values to a variable name. Now you can access each variable like so: `$name`.

Fetch Scalar (Single Value)

A common use case for this is if you just want to get a row count and store it in a variable. There's a gotcha with using `fetch(PDO::FETCH_COLUMN)` with a boolean value, as there is no way to distinguish (http://php.net/manual/en/pdostatement.fetchcolumn.php) between no rows and a falsy value. The following example uses the MySQL `COUNT()` function, which would obviously be fine to just check for truthiness. However, for every other case, if the column itself is a boolean value, like 0, then you should must use either `$stmt->rowCount() === 0` or `$colVal === false` to check if there are no rows.

To be clear, this behavior *doesn't* occur when you need to fetch an array with `fetchAll(PDO::FETCH_COLUMN)`. Nonetheless, if you were to use `fetch(PDO::FETCH_COLUMN)` in a loop to store values in your array, then this unexpected behavior still occurs. I honestly don't see why anyone would do this over using `fetchAll(PDO::FETCH_COLUMN)`, but it should be noted.

```
$stmt = $pdo->prepare("SELECT COUNT(*) FROM myTable WHERE weight < ?");
$stmt->execute([185]);
$count = $stmt->fetch(PDO::FETCH_COLUMN);
//Syntactic sugar for previous line: $count = $stmt->fetchColumn();
if(!$count) exit('No rows');
echo $count; //Output: 1784
$stmt = null;
```

Now `$count` is the literal value of the row count.

Fetch Multiple Columns as Separate Array Variable

This is can be handy, as you can easily separate it into a bunch of separate 1D arrays, rather than just one multi-dimensional array. Keep in mind that I used `rowCount()` to check if there are any rows. Most drivers don't have ability to use `rowCount()` on `SELECT` statements, but MySQL does. If you are using a different driver, you can use `isset()` on each array variable after the while loop or declare each variable to an empty array.

```
$stmt = $pdo->prepare("SELECT id, age, height FROM myTable WHERE weight > ?");
$stmt->execute([120]);
if($stmt->rowCount() === 0) exit('No rows');
while ($row = $stmt->fetch()) {
  $ids[] = $row['id'];
  $ages[] = $row['age'];
  $names[] = $row['name']
}
var_export($ages);
$stmt = null;
```

Output:

```
[8, 12, 28, 64, 43, 29]
```

Fetch Single Column as Array Variable

The same concept as the example right before, but this is handy if all you need to do is get the an array of only one column.

```
$stmt = $pdo->prepare("SELECT height FROM myTable WHERE id < ?");
$stmt->execute([500]);
$heights = $stmt->fetchAll(PDO::FETCH_COLUMN);
if(!$heights) exit('No rows');
var_export($heights);
$stmt = null;
```

Output:

```
[78, 64, 68, 54, 58]
```

Fetch Key/Value Pair

This creates an associative array with the format of the first column as the key and the second column as the value. Therefore, your first column needs to be a unique value.

```
$stmt = $pdo->prepare("SELECT id, name FROM myTable WHERE age < ?");
$stmt->execute([25]);
$arr = $stmt->fetchAll(PDO::FETCH_KEY_PAIR);
if(!$arr) exit('No rows');
var_export($arr);
$stmt = null;
```

Output:

```
[7 => 'Jerry', 10 => 'Bill', 29 => 'Bobby']
```

Fetch Key/Value Pair Array

For lack of a better term obviously. What I mean by this is that the key will be your first column, which needs to be a unique value, while the value will be the rest of the columns as an associative array.

```
$stmt = $pdo->prepare("SELECT id, max_bench, max_squat FROM myTable WHERE weight
$stmt->execute([225]);
$arr = $stmt->fetchAll(PDO::FETCH_UNIQUE);
if(!$arr) exit('No rows');
var_export($arr);
$stmt = null;
```

I'm not sure why this comment (http://php.net/manual/en/pdostatement.fetchall.php#115175) on the PHP docs states that you must bitwise it and add `FETCH_GROUP`, like so: `$stmt->fetchAll(PDO::FETCH_UNIQUE | PDO::FETCH_GROUP)`. It has the same effect either way from my testings.

Output:

```
[
  17 => ['max_bench' => 230, 'max_squat' => 175],
  84 => ['max_bench' => 195, 'max_squat' => 235],
  136 => ['max_bench' => 135, 'max_squat' => 285]
]
```

Fetch in Groups

Let's say you want to group by eye color for instance. This handy fetch mode allows you to do it extremely trivially.

```
$stmt = $pdo->prepare("SELECT eye_color, name, weight FROM myTable WHERE age < ?"
$stmt->execute([35]);
$arr = $stmt->fetchAll(PDO::FETCH_GROUP);
if(!$arr) exit('No rows');
var_export($arr);
$stmt = null;
```

Output:

```
[
  'green' => [
    ['name' => 'Patrick', 'weight' => 178],
    ['name' => 'Olivia', 'weight' => 132]
  ],
  'blue' => [
    ['name' => 'Kyle', 'weight' => 128],
    ['name' => 'Ricky', 'weight' => 143]
  ],
  'brown' => [
    ['name' => 'Jordan', 'weight' => 173],
    ['name' => 'Eric', 'weight' => 198]
  ]
]
```

Fetch in Groups, One Column

The difference between this and the previous example is essentially the same situation as `FETCH_KEY_PAIR` vs `FETCH_UNIQUE`. The preceding example groups the first column, with an array, while this one groups the first column with all values from the second column.

```
$stmt = $pdo->prepare("SELECT eye_color, name FROM myTable WHERE age < ?");
$stmt->execute([35]);
$arr = $stmt->fetchAll(PDO::FETCH_GROUP | PDO::FETCH_COLUMN);
if(!$arr) exit('No rows');
var_export($arr);
$stmt = null;
```

Output:

```
[
  'green' => ['Patrick', 'Olivia'],
  'blue' => ['Kyle', 'Ricky'],
  'brown' => ['Jordan', 'Eric']
]
```

Fetch in Groups, Object Arrays

Same as fetching in a regular group, but with object subarrays instead.

```
$stmt = $pdo->prepare("SELECT eye_color, name, weight FROM myTable WHERE age < ?"
$stmt->execute([35]);
$arr = $stmt->fetchAll(PDO::FETCH_GROUP | PDO::FETCH_CLASS);
if(!$arr) exit('No rows');
var_export($arr);
$stmt = null;
```

Output:

```
[
  'green' => [
    stdClass Object ['name' => 'Patrick', 'weight' => 178],
    stdClass Object ['name' => 'Olivia', 'weight' => 132]
  ],
  'blue' => [
    stdClass Object ['name' => 'Kyle', 'weight' => 128],
    stdClass Object ['name' => 'Ricky', 'weight' => 143]
  ],
  'brown' => [
    stdClass Object ['name' => 'Jordan', 'weight' => 173],
    stdClass Object ['name' => 'Eric', 'weight' => 198]
  ]
]
```

You can specify a classname too.

```
class myClass {}
$stmt->fetchAll(PDO::FETCH_GROUP | PDO::FETCH_CLASS, 'myClass');
```

Fetch Into Existing Class

This is almost the same as `PDO::FETCH_CLASS`, `PDO::FETCH_OBJ` or `fetchObject()`. The only differences are that this fetches into an already constructed class and for some reason it won't let you modify private variables. I personally don't understand why they made a separate fetch mode for this, rather than allow you to pass it into `fetch()` with

`PDO::FETCH_OBJ`. Another annoying aspect is that PDO forces you to use `$stmt->setFetchMode(PDO::FETCH_INTO, $myClass)`, followed by `fetch()` (`fetchAll()` will give you the exact same result).

```
class myClass {
  public $max_bench;
  public $max_squat;
}

$myClass = new myClass();
var_export($myClass);

$stmt = $pdo->prepare("SELECT max_bench, max_squat FROM myTable WHERE weight < ?"
$stmt->execute([200]);
$stmt->setFetchMode(PDO::FETCH_INTO, $myClass);
$arr = $stmt->fetch();
if(!$arr) exit('No rows');
var_export($arr); //$arr === $myClass
$stmt = null;
```

Output:

```
myClass Object ['max_bench' => null, 'max_squat' => null]
myClass Object ['max_bench' => 225, 'max_squat' => 95]
```

Fetch Class from Database Value

For the average person, this probably isn't too useful. But for users who heavily use object mapping in PDO, this actually pretty cool. Though these type of users would like be using an ORM or query builder, it nevertheless showcases how powerful PDO is on its own.

For this work, you need to declare the names of your classes, otherwise it'll just use **stdClass**. So you need to know the values of your database, which could be inconvenient. You are also not allowed to declare parameter arguments, like you would with `PDO::FETCH_CLASS` on its own.

```
Class California {}
Class Florida {}
Class NewYork {}

$stmt = $pdo->prepare("SELECT state, age, first_name FROM info WHERE id > ?");
$stmt->execute([20]);
$arr = $stmt->fetchAll(PDO::FETCH_CLASS | PDO::FETCH_CLASSTYPE);
if(!$arr) exit('No rows');
var_export($arr);
$stmt = null;
```

Output:

```
[
  Florida object ['age' => 22, 'first_name' => 'Billy'],
  California object ['age' => 19, 'first_name' => 'Mary'],
  NewYork object['age' => 26, 'first_name' => 'Ryan']
]
```

Fetch Object, Associative and Numeric Array Lazily

I'm sure it sounds confusing, but I couldn't think of a better way to describe it. It's really pretty neat, since you're fetching a **PDORow** object that's a pointer to the result set essentially. It doesn't actually fetch anything at all, until you use an array or object index (lazy). The most brilliant part of the implementation is that once you "fetch" it, you have the option of using it as an object, associative or numeric array in the most memory-efficient manner possible. It should be noted that if the index is out-of-bounds, it'll return null instead of throw an error.

```
$stmt = $pdo->prepare("SELECT state, age, first_name FROM info WHERE id > ?");
$stmt->execute([320]);
$arr = $stmt->fetch(PDO::FETCH_LAZY);
if(!$arr) exit('No rows');
//The following all have the same value
echo $arr[2];
echo $arr->first_name;
echo $arr['first_name'];
$stmt = null;
```

Like

You might intuitively try to do something like the following.

```
$stmt = $pdo->prepare("SELECT id, name, age FROM myTable WHERE name LIKE %?%");
```

However, this will not work. This is how you would do it the right way.

```
$search = "%{$_POST['search']}%";
$stmt = $pdo->prepare("SELECT id, name, age FROM myTable WHERE name LIKE ?");
$stmt->execute([$search]);
$arr = $stmt->fetchAll();
if(!$arr) exit('No rows');
var_export($arr);
$stmt = null;
```

Where In Array

As you can see, PDO clearly excels in this too, as the code is much shorter, due to not needing to specify the type with `bindValue()` or `bindParam()`.

```
$inArr = [1, 3, 5];
$clause = implode(',', array_fill(0, count($inArr), '?')); //create 3 question ma
$stmt = $pdo->prepare("SELECT * FROM myTable WHERE id IN ($clause)");
$stmt->execute($inArr);
$resArr = $stmt->fetchAll();
if(!$resArr) exit('No rows');
var_export($resArr);
$stmt = null;
```

With Other Placeholders

```
$inArr = [1, 3, 5];
$clause = implode(',', array_fill(0, count($inArr), '?')); //create 3 question ma
$fullArr = array_merge($inArr, [5]); //merge WHERE IN array with other value(s)
$stmt = $pdo->prepare("SELECT * FROM myTable WHERE id IN ($clause) AND id < ?");
$stmt->execute($fullArr);
$resArr = $stmt->fetchAll();
if(!$resArr) exit('No rows');
var_export($resArr);
$stmt = null;
```

Multiple Prepared Statements in Transactions

If you want to ensure that multiple SQL calls are concurrent, then you must use transactions. This ensures that either all of your operations or none of them will succeed. For instance, this could be useful for transferring a row to a different table. You'll want copy the row over to the new table and delete the other one. If one of the operations fails, then it needs to revert back to its previous state.

```
try {
  $pdo->beginTransaction();
  $stmt1 = $pdo->prepare("INSERT INTO myTable (name, state) VALUES (?, ?)");
  $stmt2 = $pdo->prepare("UPDATE myTable SET age = ? WHERE id = ?");
  if(!$stmt1->execute(['Rick', 'NY'])) throw new Exception('Stmt 1 Failed');
  else if(!$stmt2->execute([27, 139])) throw new Exception('Stmt 2 Failed');
  $stmt1 = null;
  $stmt2 = null;
  $pdo->commit();
} catch(Exception $e) {
  $pdo->rollback();
  throw $e;
}
```

Reuse Same Template, Different Values

```
try {
  $pdo->beginTransaction();
  $stmt = $pdo->prepare("INSERT INTO myTable (name, age) VALUES (?, ?)");
  if(!$stmt->execute(['Joe', 19])) throw new Exception('Stmt 1 Failed');
  else if(!$stmt->execute(['Ryan', 44])) throw new Exception('Stmt 2 Failed');
  $stmt = null;
  $pdo->commit();
} catch(Exception $e) {
  $pdo->rollback();
  throw $e;
}
```

Error Handling

> To prevent leaking your password, here's what your php.ini file should look like in production: do both `display_errors = Off` and `log_errors = On`. Then restart Apache or Ngnix

If you turned on errors and forced them to be exceptions, like in the create new connection section then the easiest way to handle your errors is by putting them in a try/catch block. Now you have access to the PDOException class (http://php.net/manual/en/class.pdoexception.php).

A beginner might assume that proper error handling entails wrapping each query block in a separate `try/catch` block, similar to regular error handling with an `if` statement. While there's nothing technically wrong with doing that, it just looks a lot more elegant to use a single, global `try/catch` using the base `Exception` class or to use `set_exception_handler()`. The only *exception* to this is with transactions, which should have its on separate one, but then throw the exception for it to go to the global `try/catch`.

All of your pages — even ones without PDO — should be set up like this, as you typically just need to give a message for the entire php page. However, sometimes you might need to catch specific cases, so you can use as many specific exception types as you need, along with `Exception $e`.

```
try {
  $stmt = $pdo->prepare("INSERT INTO myTable (name, age) VALUES (?, ?)");
  if(!$stmt->execute(['Justin', 18])) throw new Exception('Stmt Failed');
  $stmt = null;

  $stmt = $pdo->prepare("SELECT * FROM myTable WHERE power_level > ?");
  if(!$stmt->execute([9000])) throw new Exception('Stmt Failed');
  $arr = $stmt->fetchAll();
  $stmt = null;

  try {
    $pdo->beginTransaction();
    $stmt1 = $pdo->prepare("INSERT INTO myTable (name, state) VALUES (?, ?)");
    $stmt2 = $pdo->prepare("UPDATE myTable SET age = ? WHERE id = ?");
    if(!$stmt1->execute(['Rick', 'NY'])) throw new Exception('Stmt 1 Failed');
    else if(!$stmt2->execute([27, 139])) throw new Exception('Stmt 2 Failed');
    $stmt1 = null;
    $stmt2 = null;
    $pdo->commit();
  } catch(Exception $e) {
    $pdo->rollback();
    throw $e;
  }
} catch(Exception $e) {
  error_log($e);
  exit('Error message that user can understand for this page');
}
```

Another way to handle the exceptions is by creating a user-defined exception handler, which I mentioned earlier. You would add the following on each page *after* including `pdo_connect.php`. This way you can leave out `try/catch` on almost all of your queries *except* for transactions, which you would throw an exception after `catch`ing if something went wrong

```
set_exception_handler(function($e) {
  error_log($e);
  exit('Error inserting');
});
$stmt = $pdo->prepare("INSERT INTO myTable (name, age) VALUES (?, ?)");
if(!$stmt->execute(['Justin', 18])) throw new Exception('Stmt Failed');
$stmt = null;
```

Why check for truthiness?

You may have noticed that I'm throwing an exception for execute if it's fasly, which seems redundant, as we already turned on error handling in the form of exceptions. Nevertheless, I noticed an odd behavior, which is that `execute()` can solely return false in some scenarios if emulation mode is turned off, which is the only mode this tutorial is discussing. It could be MySQL specific, but I'm leaving it in since I personally have experienced this when there are too many parameters bound to execute. It will simply return false and act as if nothing went wrong. This would give especially undesirable behavior in transactions, since a query would silently fail, while the others would work, therefore defeating its purpose of being linearizable. This is why you must check for truthiness in case this happens. I actually couldn't find too much info about it, but this StackOverflow (https://stackoverflow.com/a/32790480) describes the issue pretty well. Weirdly enough, if you don't bind *enough* variables, it'll correctly throw an exception.

Some Extras

Do I need $stmt = null?

This is essentially the same as using `$stmt->close()` in MySQLi and the same applies. No, it's certainly not required, but is considered good coding practice by some (obviously subjective). I prefer to be explicit and I also do both `$stmt = null` and `$pdo = null`. If you are closing the PDO connection, then you must close the prepared statements as well, as stated here (http://php.net/manual/en/pdo.connections.php#114822). While this isn't exactly the same as using `$mysqli->close()`, it's pretty similar. A PDO function to close the connection is something that has been requested for years, and is dubious if it'll ever be implemented.

Closing the prepared statements would be useful if you're reusing the same variable name. Both are not truly necessary, as they will close at the end of the script's execution anyway.

Emulation Mode vs. Native Prepared Statements

When using prepared statements, you have two options: emulation mode on or off. This article strictly covered native prepared statements, as I believe that you should use real prepared statements if your driver version supports it. Emulation mode seems more like a fallback solution for drivers/versions not supporting native prepared statements; this has been supported in MySQL since version 4.1 (http://php.net/manual/en/ref.pdo-mysql.php#pdo-mysql.intro).

When emulation mode is turned on, it's essentially like using `PDO::quote` (http://php.net/manual/en/pdo.quote.php) or type casting to manually format your queries — it'll automagically always do this securely. While this should still be just as secure in theory by using MySQL 5.5+ and setting the charset to **utf8mb4** when you create a connection, I'd still suggest using native prepared statements. Check out this excellent write up on an obscure edge case attack (https://stackoverflow.com/a/12202218).

Even though we're talking about theoretical threats, non-emulated prepared statements completely eliminate the possibility of an SQL injection attack. The query with the dummy placeholders is sent to the server first, followed by the values to bind — the query and data are completely isolated.

Here are some key differences between the two.

Emulation Mode:

- Can reuse named placeholders
- Allows multiple query support
- Need to use `bindValue()` for certain edge cases, like with LIMIT
- Faster for single statement, but can't run prepared once, execute multiple
- Costs **n** client-server round trips for each statement
- Reports errors when statement is executed

Native Prepared Statements:

- Can run prepared once, execute multiple for efficiency
- Costs **1+n** client-server round trips for each statement
- Can only use each named placeholder once
- Can't run multiple queries (though you can use transactions)

- In theory, more secure due to the query and values being isolated
- Reports errors when statement is prepared

Here's an example of how you would use LIMIT with emulation mode on. The reason it's happening, is because MySQL ends up interpreting it as **LIMIT '23'**. So you can either use native prepared statements, or use `bindValue()` to explicitly define it as an **int**.

```
$stmt = $pdo->prepare("SELECT full_name, gender FROM myTable WHERE id > ? LIMIT ?
$stmt->bindValue(1, 39, PDO::PARAM_INT);
$stmt->bindValue(2, 23, PDO::PARAM_INT);
$arr = $stmt->fetchAll();
if(!$arr) exit('No rows');
var_export($arr);
$stmt = null;
```

bindValue() vs. bindParam()

This tutorial didn't really go over either too much, since you don't really need these, except for in edge cases when you need enforce the data type. Nevertheless, it's worthwhile to understand the differences, as you never know when you might run into a situation where it could be useful.

Both methods are used to manually bind to the prepared statement. The difference is that `bindValue()` is more versatile, as you can bind variables *and* values, while `bindParam()` can only accept variables. Therefore, `bindParam()` is identical to `bind_param()` in MySQLi. So why does this method even exist, if it only has disadvantages? Consider the following case.

```
$id = 25;
$stmt = $pdo->prepare("SELECT * FROM myTable WHERE id < ?");
$stmt->bindParam($id);
$id = 98;
$id = 39; //Final value
$stmt->execute();
$arr = $stmt->fetchAll();
if(!$arr) exit('No rows');
var_export($arr);
$stmt = null;
```

With `bindParam()`, you can continually change the variable and re-execute. This is not the case with `bindValue()`, as you will need call the method again. I doubt I'll ever need this, but it's nice to have the option. The reason it acts like this is obvious if you take a look at the docs, as it's a pass by reference function argument.

```
$id = 25; //Final value
$stmt = $pdo->prepare("SELECT * FROM myTable WHERE id < ?");
$stmt->bindValue($id);
$id = 98;
$id = 39;
$stmt->execute();
$arr = $stmt->fetchAll();
if(!$arr) exit('No rows');
var_export($arr);
$stmt = null;
```

So Using Prepared Statements Means I'm Safe From Attackers?

While you are safe from SQL injection, you still need validate and sanitize your user-inputted data. You can use a function like filter_var() (http://php.net/manual/en/function.filter-var.php) to validate *before* inserting it into the database and htmlspecialchars() (http://php.net/manual/en/function.htmlspecialchars.php) to sanitize *after* retrieving it.

Series

1	◂ PHP MySQLi Prepared Statements Tutorial to Prevent SQL Injection	Nov 8, 2017	(prepared-statements-in-php-mysqli-to-prevent-sql-injection)
2	◂ PHP PDO Prepared Statements Tutorial to Prevent SQL Injection	Nov 26, 2017	(php-pdo-prepared-statements-to-prevent-sql-injection)
3	◂ PDO vs. MySQLi: The Battle of PHP Database APIs	Jun 8, 2018	(php-pdo-vs-mysqli)

Author - Daniel Marcus

Firmly believes that web technologies should take over everything. Enjoys writing tutorials about JavaScript and PHP.

5 comments

Sort by Nev