# (The only proper) PDO tutorial

1. [Why PDO?](https://phpdelusions.net/pdo" \l "why)
2. [Connecting. DSN](https://phpdelusions.net/pdo" \l "dsn)
3. [Running queries. PDO::query()](https://phpdelusions.net/pdo" \l "query)
4. [Prepared statements. Protection from SQL injections](https://phpdelusions.net/pdo" \l "prepared)
   * [Binding methods](https://phpdelusions.net/pdo" \l "methods)
   * [Query parts you can bind](https://phpdelusions.net/pdo" \l "bindable)
5. [Prepared statements. Multiple execution](https://phpdelusions.net/pdo" \l "multiexec)
6. [Running SELECT INSERT, UPDATE, or DELETE statements](https://phpdelusions.net/pdo" \l "dml)
7. [Getting data out of statement. foreach()](https://phpdelusions.net/pdo" \l "foreach)
8. [Getting data out of statement. fetch()](https://phpdelusions.net/pdo" \l "fetch)
   * [Return types.](https://phpdelusions.net/pdo" \l "returntypes)
9. [Getting data out of statement. fetchColumn()](https://phpdelusions.net/pdo" \l "fetchcolumn)
10. [Getting data out of statement in dozens different formats. fetchAll()](https://phpdelusions.net/pdo" \l "fetchall)
    * [Getting a plain array.](https://phpdelusions.net/pdo" \l "array)
    * [Getting a column.](https://phpdelusions.net/pdo" \l "column)
    * [Getting key-value pairs.](https://phpdelusions.net/pdo" \l "pairs)
    * [Getting rows indexed by unique field](https://phpdelusions.net/pdo" \l "unique)
    * [Getting rows grouped by some field](https://phpdelusions.net/pdo" \l "group)
11. [Error handling. Exceptions](https://phpdelusions.net/pdo" \l "errors)
    * [Reporting PDO errors](https://phpdelusions.net/pdo" \l "reporting_errors)
    * [Catching PDO exceptions](https://phpdelusions.net/pdo" \l "catch)
12. [Getting row count with PDO](https://phpdelusions.net/pdo" \l "count)
13. [Affected rows and insert id](https://phpdelusions.net/pdo" \l "affected)
14. [Prepared statements and LIKE clause](https://phpdelusions.net/pdo" \l "like)
15. [Prepared statements and IN clause](https://phpdelusions.net/pdo" \l "in)
16. [Protecting table and field names](https://phpdelusions.net/pdo" \l "identifiers)
17. [A problem with LIMIT clause](https://phpdelusions.net/pdo" \l "limit)
18. [Transactions](https://phpdelusions.net/pdo" \l "transactions)
19. [Calling stored procedures in PDO](https://phpdelusions.net/pdo" \l "call)
20. [Running multiple queries with PDO](https://phpdelusions.net/pdo" \l "multiquery)
21. [Emulation mode. PDO::ATTR\_EMULATE\_PREPARES](https://phpdelusions.net/pdo" \l "emulation)
    * [When emulation mode is turned ON](https://phpdelusions.net/pdo" \l "emulation_on)
    * [When emulation mode is turned OFF](https://phpdelusions.net/pdo" \l "emulation_off)
22. [Mysqlnd and buffered queries. Huge datasets.](https://phpdelusions.net/pdo" \l "mysqlnd)
23. [Comments (288)](https://phpdelusions.net/pdo" \l "comments)

There are many tutorials on PDO already, but unfortunately, most of them fail to explain the real benefits of PDO, or even promote rather bad practices. The only two exceptions are [phptherightway.com](http://www.phptherightway.com/" \l "pdo_extension) and [hashphp.org](http://wiki.hashphp.org/PDO_Tutorial_for_MySQL_Developers), but they miss a lot of important information. As a result, half of PDO's features remain in obscurity and are almost never used by PHP developers, who, as a result, are constantly trying to reinvent the wheel which already exists in PDO.

Unlike those, this tutorial is written by someone who has used PDO for many years, dug through it, and answered thousands questions on Stack Overflow (the [sole gold PDO badge bearer](http://stackoverflow.com/help/badges/4220/pdo)). Following [the mission of this site](https://phpdelusions.net/), this article will disprove various delusions and bad practices, while showing the right way instead.

Although this tutorial is based on mysql driver, the information, in general, is applicable for any driver supported.

### Why PDO?

First things first. Why at all?

PDO is a [Database Access Abstraction Layer](https://en.wikipedia.org/wiki/Database_abstraction_layer). The abstraction, however, is two-fold: one is widely known but less significant, while another is obscure but of most importance.

Everyone knows that PDO offers unified interface to access [many different databases](http://php.net/manual/en/pdo.drivers.php). Although this feature is magnificent by itself, it doesn't make a big deal for the particular application, where only one database backend is used anyway. And, despite some rumors, it is impossible to switch database backends by changing a single line in PDO config - due to different SQL flavors (to do so, one needs to use an averaged query language like [DQL](http://doctrine-orm.readthedocs.org/projects/doctrine-orm/en/latest/reference/dql-doctrine-query-language.html)). Thus, for the average LAMP developer, this point is rather insignificant, and to him, PDO is just a more complicated version of familiar mysql(i)\_query() function. However, it is not; it is much, much more.

PDO abstracts not only a database API, but also basic operations that otherwise have to be repeated hundreds of times in every application, making your code extremely . [Unlike *mysql* and *mysqli*](https://phpdelusions.net/pdo/mysqli_comparison), both of which are low level bare APIs not intended to be used directly (but only as a building material for some higher level abstraction layer), PDO is such an abstraction already. Still incomplete though, but at least usable.

The real PDO benefits are:

* security (usable prepared statements)
* usability (many helper functions to automate routine operations)
* reusability (unified API to access multitude of databases, from SQLite to Oracle)

Note that although PDO is the best out of native db drivers, for a modern web-application consider to use an ORM with a Query Builder, or any other higher level abstraction library, with only occasional fallback to vanilla PDO. Good ORMs are Doctrine, Eloquent, RedBean, and Yii::AR. Aura.SQL is a good example of a PDO wrapper with many additional features.

Either way, it's good to know the basic tools first. So, let's begin:

### Connecting. DSN

PDO has a fancy connection method called [DSN](https://en.wikipedia.org/wiki/Data_source_name). It's nothing complicated though - instead of one plain and simple list of options, PDO asks you to input different configuration directives in three different places:

* database driver, host, db (schema) name and charset, as well as less frequently used port and unix\_socket go into DSN;
* username and password go to constructor;
* all other options go into options array.

where DSN is a semicolon-delimited string, consists of param=value pairs, that begins from the driver name and a colon:

mysql:host=localhost;dbname=test;port=3306;charset=utf8mb4

driver^ ^ colon ^param=value pair ^semicolon

Note that it's important to follow the proper format - no spaces or quotes or other decorations have to be used in DSN, but only parameters, values and delimiters, as shown in the [manual](http://php.net/manual/en/pdo.construct.php).

Here goes an example for mysql:

$host = '127.0.0.1';

$db = 'test';

$user = 'root';

$pass = '';

$charset = 'utf8mb4';

$dsn = "mysql:host=$host;dbname=$db;charset=$charset";

$options = [

PDO::ATTR\_ERRMODE => PDO::ERRMODE\_EXCEPTION,

PDO::ATTR\_DEFAULT\_FETCH\_MODE => PDO::FETCH\_ASSOC,

PDO::ATTR\_EMULATE\_PREPARES => false,

];

try {

$pdo = new PDO($dsn, $user, $pass, $options);

} catch (\PDOException $e) {

throw new \PDOException($e->getMessage(), (int)$e->getCode());

}

With all aforementioned variables properly set, we will have proper PDO instance in $pdo variable.

Important notes for the late mysql extension users:

1. Unlike old mysql\_\* functions, which can be used anywhere in the code, PDO instance is stored in a regular variable, which means it can be inaccessible inside functions - so, one has to make it accessible, by means of passing it via function parameters or using more advanced techniques, such as IoC container.
2. The connection has to be made only once! No connects in every function. No connects in every class constructor. Otherwise, multiple connections will be created, which will eventually kill your database server. Thus, a sole PDO instance has to be created and then used through whole script execution.
3. It is very important to set charset through DSN - that's the only proper way because it tells PDO which charset is going to be used. Therefore forget about running SET NAMES query manually, either via query() or PDO::MYSQL\_ATTR\_INIT\_COMMAND. Only if your PHP version is unacceptably outdated (namely below 5.3.6), you have to use SET NAMES query and always turn [emulation mode](https://phpdelusions.net/pdo" \l "emulation) off.

More details regarding Mysql can be found in the corresponding chapter, [Connecting to MySQL](https://phpdelusions.net/pdo_examples/connect_to_mysql)

### Running queries. PDO::query()

There are two ways to run a query in PDO. If no variables are going to be used in the query, you can use the [PDO::query()](http://php.net/manual/en/pdo.query.php) method. It will run your query and return special object of [PDOStatement class](http://php.net/manual/en/class.pdostatement.php) which can be roughly compared to a resource, returned by mysql\_query(), especially in the way you can get actual rows out of it:

$stmt = $pdo->query('SELECT name FROM users');

while ($row = $stmt->fetch())

{

echo $row['name'] . "\n";

}

Also, the query() method allows us to use a neat method chaining for SELECT queries, which will be shown below.

### Prepared statements. Protection from SQL injections

This is the main and the only important reason why you were deprived from your beloved mysql\_query() function and thrown into the harsh world of Data Objects: PDO has prepared statements support out of the box. Prepared statement is the only proper way to run a query, if any variable is going to be used in it. The reason why it is so important is explained in detail in [The Hitchhiker's Guide to SQL Injection prevention](https://phpdelusions.net/sql_injection).

So, for every query you run, if at least one variable is going to be used, you have to substitute it with a placeholder, then prepare your query, and then execute it, passing variables separately.

Long story short, it is not as hard as it seems. In most cases, you need only two functions - [prepare()](http://php.net/manual/en/pdo.prepare.php) and [execute()](http://php.net/manual/en/pdostatement.execute.php).

First of all, you have to alter your query, adding placeholders in place of variables. Say, a code like this

$sql = "SELECT \* FROM users WHERE email = '$email' AND status='$status'";

will become

$sql = 'SELECT \* FROM users WHERE email = ? AND status=?';

or

$sql = 'SELECT \* FROM users WHERE email = :email AND status=:status';

Note that PDO supports positional (?) and named (:email) placeholders, the latter always begins from a colon and can be written using letters, digits and underscores only. Also note that no quotes have to be ever used around placeholders.

Having a query with placeholders, you have to prepare it, using the PDO::prepare() method. This function will return the same PDOStatement object we were talking about above, but without any data attached to it.

Finally, to get the query executed, you must run execute() method of this object, passing variables in it, in the form of array. And after that, you will be able to get the resulting data out of statement (if applicable):

$stmt = $pdo->prepare('SELECT \* FROM users WHERE email = ? AND status=?');

$stmt->execute([$email, $status]);

$user = $stmt->fetch();

// or

$stmt = $pdo->prepare('SELECT \* FROM users WHERE email = :email AND status=:status');

$stmt->execute(['email' => $email, 'status' => $status]);

$user = $stmt->fetch();

As you can see, for the positional placeholders, you have to supply a regular array with values, while for the named placeholders, it has to be an associative array, where keys have to match the placeholder names in the query. You cannot mix positional and named placeholders in the same query.

Please note that positional placeholders let you write shorter code, but are sensitive to the order of arguments (which have to be exactly the same as the order of the corresponding placeholders in the query). While named placeholders make your code more verbose, they allow random binding order.

Also note that despite a widespread delusion, no ":" in the keys is required.

After the execution you may start getting your data, using all supported methods, as described down in this article.

More examples can be found in the [respective article](https://phpdelusions.net/pdo_examples).

#### Binding methods

Passing data into execute() (like shown above) should be considered default and most convenient method. When this method is used, all values will be bound as strings (save for NULL values, that will be sent to the query as is, i.e. as SQL NULL), but most of time it's all right and won't cause any problem.

However, sometimes it's better to set the data type explicitly. Possible cases are:

* [LIMIT](https://phpdelusions.net/pdo" \l "limit) clause (or any other SQL clause that just cannot accept a string operand) if [emulation mode](https://phpdelusions.net/pdo" \l "emulation) is turned ON.
* complex queries with non-trivial query plan that can be affected by a wrong operand type
* peculiar column types, like BIGINT or BOOLEAN that require an operand of exact type to be bound (note that in order to bind a BIGINT value with PDO::PARAM\_INT you need a [mysqlnd](https://phpdelusions.net/pdo" \l "mysqlnd)-based installation).

In such a case explicit binding have to be used, for which you have a choice of two functions, [bindValue()](http://php.net/manual/en/pdostatement.bindvalue.php) and [bindParam()](http://php.net/manual/en/pdostatement.bindparam.php). The former one has to be preferred, because, unlike bindParam() it has no side effects to deal with.

#### Query parts you can bind

It is very important to understand which query parts you can bind using prepared statements and which you cannot. In fact, the list is overwhelmingly short: only string and numeric literals can be bound. So you can tell that as long as your data can be represented in the query as a numeric or a quoted string literal - it can be bound. For all other cases you cannot use PDO prepared statements at all: neither an identifier, or a comma-separated list, or a part of a quoted string literal or whatever else arbitrary query part cannot be bound using a prepared statement.

Workarounds for the most frequent use cases can be found in the [corresponding part of the article](https://phpdelusions.net/pdo" \l "like)

### Prepared statements. Multiple execution

Sometimes you can use prepared statements for the multiple execution of a prepared query. It is slightly faster than performing the same query again and again, as it does query parsing only once. This feature would have been more useful if it was possible to execute a statement prepared in another PHP instance. But alas - it is not. So, you are limited to repeating the same query only within the same instance, which is seldom needed in regular PHP scripts and which is limiting the use of this feature to repeated inserts or updates:

$data = [

1 => 1000,

5 => 300,

9 => 200,

];

$stmt = $pdo->prepare('UPDATE users SET bonus = bonus + ? WHERE id = ?');

foreach ($data as $id => $bonus)

{

$stmt->execute([$bonus, $id]);

}

Note that this feature is a bit overrated. Not only it is needed too seldom to talk about, but the performance gain is not that big - query parsing is *real* fast these times.

Note that you can get this advantage only when [emulation mode](https://phpdelusions.net/pdo" \l "emulation) is turned off.

### Running SELECT INSERT, UPDATE, or DELETE statements

Come on folks. There is absolutely nothing special in these queries. To PDO they all the same. It doesn't matter which query you are running.

Just like it was shown above, what you need is to prepare a query with placeholders, and then execute it, sending variables separately. Either for DELETE and SELECT query the process is essentially the same. The only difference is (as queries do not return any data), that you can use the method chaining and thus call execute() right along with prepare():

$sql = "UPDATE users SET name = ? WHERE id = ?";

$pdo->prepare($sql)->execute([$name, $id]);

However, if you want to get the number of affected rows, the code will have to be the same boresome three lines:

$stmt = $pdo->prepare("DELETE FROM goods WHERE category = ?");

$stmt->execute([$cat]);

$deleted = $stmt->rowCount();

More examples can be found in the [respective article](https://phpdelusions.net/pdo_examples).

### Getting data out of statement. foreach()

The most basic and direct way to get multiple rows from a statement would be foreach() loop. Thanks to [Traversable](http://php.net/manual/en/class.traversable.php) interface, PDOStatement can be iterated over by using foreach() operator:

$stmt = $pdo->query('SELECT name FROM users');

foreach ($stmt as $row)

{

echo $row['name'] . "\n";

}

Note that this method is memory-friendly, as it doesn't load all the resulting rows in the memory but delivers them one by one (though keep in mind this [issue](https://phpdelusions.net/pdo" \l "mysqlnd)).

### Getting data out of statement. fetch()

We have seen this function already, but let's take a closer look. It fetches a single row from database, and moves the internal pointer in the result set, so consequent calls to this function will return all the resulting rows one by one. Which makes this method a rough analogue to mysql\_fetch\_array() but it works in a slightly different way: instead of many separate functions (mysql\_fetch\_assoc(), mysql\_fetch\_row(), etc), there is only one, but its behavior can be changed by a parameter. There are many fetch modes in PDO, and we will discuss them later, but here are few for starter:

* PDO::FETCH\_NUM returns enumerated array
* PDO::FETCH\_ASSOC returns associative array
* PDO::FETCH\_BOTH - both of the above
* PDO::FETCH\_OBJ returns object
* PDO::FETCH\_LAZY allows all three (numeric associative and object) methods without memory overhead.

From the above you can tell that this function have to be used in two cases:

1. When only one row is expected - to get that only row. For example,

$row = $stmt->fetch(PDO::FETCH\_ASSOC);

Will give you single row from the statement, in the form of associative array.

1. When we need to process the returned data somehow before use. In this case it have to be run through usual while loop, like one shown [above](https://phpdelusions.net/pdo" \l "query).

Another useful mode is PDO::FETCH\_CLASS, which can create an object of particular class

$news = $pdo->query('SELECT \* FROM news')->fetchAll(PDO::FETCH\_CLASS, 'News');

will produce an array filled with objects of News class, setting class properties from returned values. Note that in this mode

* properties are set before constructor call
* for all undefined properties \_\_set magic method will be called
* if there is no \_\_set method in the class, then new property will be created
* private properties will be filled as well, which is a bit unexpected but quite handy

Note that default mode is PDO::FETCH\_BOTH, but you can change it using PDO::ATTR\_DEFAULT\_FETCH\_MODE configuration option as shown in the connection example. Thus, once set, it can be omitted most of the time.

#### Return types.

Only when PDO is built upon [mysqlnd](https://phpdelusions.net/pdo" \l "mysqlnd) and [emulation mode](https://phpdelusions.net/pdo" \l "emulation) is off, then PDO will return int and float values with respective types. Say, if we create a table

create table typetest (string varchar(255), `int` int, `float` float, `null` int);

insert into typetest values('foo',1,1.1,NULL);

And then query it from mysqlnd-based PDO with emulation turned off, the output will be

array(4) {

["string"] => string(3) "foo"

["int"] => int(1)

["float"] => float(1.1)

["null"] => NULL

}

Otherwise the familiar mysql\_fetch\_array() behavior will be followed - all values returned as strings with only NULL returned as NULL.

If for some reason you don't like this behavior and prefer the old style with strings and NULLs only, then you can use the following configuration option to override it:

$pdo->setAttribute(PDO::ATTR\_STRINGIFY\_FETCHES, true);

Note that for the DECIMAL type the string is always returned, due to nature of this type intended to retain the precise value, unlike deliberately non-precise FLOAT and DOUBLE types.

### Getting data out of statement. fetchColumn()

A neat helper function that returns value of the singe field of returned row. Very handy when we are selecting only one field:

// Getting the name based on id

$stmt = $pdo->prepare("SELECT name FROM table WHERE id=?");

$stmt->execute([$id]);

$name = $stmt->fetchColumn();

// getting number of rows in the table utilizing method chaining

$count = $pdo->query("SELECT count(\*) FROM table")->fetchColumn();

### Getting data out of statement in dozens different formats. fetchAll()

That's most interesting function, with most astonishing features. Mostly thanks to its existence one can call PDO a wrapper, as this function can automate many operations otherwise performed manually.

PDOStatement::fetchAll() returns an array that consists of **all the rows** returned by the query. From this fact we can make two conclusions:

1. This function should not be used if many\* rows have been selected. In such a case, a conventional while loop should be used to fetch rows one by one instead of getting them all as an array at once.  
   \* "many" means more than is suitable to be shown on the average web page.
2. This function is mostly useful in a modern web application that never outputs data right away during fetching, but rather passes it to template.

You'd be amazed, in how many different formats this function can return data in (and how little an average PHP user knows of them), all controlled by PDO::FETCH\_\* variables. Some of them are:

#### Getting a plain array.

By default, this function will return just simple enumerated array consists of all the returned rows. Row formatting constants, such as PDO::FETCH\_NUM, PDO::FETCH\_ASSOC, PDO::FETCH\_OBJ etc can change the row format.

$data = $pdo->query('SELECT name FROM users')->fetchAll(PDO::FETCH\_ASSOC);

var\_export($data);

/\*

array (

0 => array('John'),

1 => array('Mike'),

2 => array('Mary'),

3 => array('Kathy'),

)\*/

#### Getting a column.

It is often very handy to get plain one-dimensional array right out of the query, if only one column out of many rows being fetched. Here you go:

$data = $pdo->query('SELECT name FROM users')->fetchAll(PDO::FETCH\_COLUMN);

/\* array (

0 => 'John',

1 => 'Mike',

2 => 'Mary',

3 => 'Kathy',

)\*/

#### Getting key-value pairs.

Also extremely useful format, when we need to get the same column, but indexed not by numbers in order but by another field. Here goes PDO::FETCH\_KEY\_PAIR constant:

$data = $pdo->query('SELECT id, name FROM users')->fetchAll(PDO::FETCH\_KEY\_PAIR);

/\* array (

104 => 'John',

110 => 'Mike',

120 => 'Mary',

121 => 'Kathy',

)\*/

Note that you have to select only two columns for this mode, first of which have to be unique.

#### Getting rows indexed by unique field

Same as above, but getting not one column but full row, yet indexed by an unique field, thanks to PDO::FETCH\_UNIQUE constant:

$data = $pdo->query('SELECT \* FROM users')->fetchAll(PDO::FETCH\_UNIQUE);

/\* array (

104 => array (

'name' => 'John',

'car' => 'Toyota',

),

110 => array (

'name' => 'Mike',

'car' => 'Ford',

),

120 => array (

'name' => 'Mary',

'car' => 'Mazda',

),

121 => array (

'name' => 'Kathy',

'car' => 'Mazda',

),

)\*/

Note that first column selected have to be unique (in this query it is assumed that first column is id, but to be sure better list it explicitly).

#### Getting rows grouped by some field

PDO::FETCH\_GROUP will group rows into a nested array, where indexes will be unique values from the first columns, and values will be arrays similar to ones returned by regular fetchAll(). The following code, for example, will separate boys from girls and put them into different arrays:

$data = $pdo->query('SELECT sex, name, car FROM users')->fetchAll(PDO::FETCH\_GROUP);

array (

'male' => array (

0 => array (

'name' => 'John',

'car' => 'Toyota',

),

1 => array (

'name' => 'Mike',

'car' => 'Ford',

),

),

'female' => array (

0 => array (

'name' => 'Mary',

'car' => 'Mazda',

),

1 => array (

'name' => 'Kathy',

'car' => 'Mazda',

),

),

)

So, this is the ideal solution for such a popular demand like "group events by date" or "group goods by category". Some real life use cases:

* [How to multiple query results in order to reduce the query number?](http://stackoverflow.com/q/35317555/285587)
* [List records grouped by category name](http://stackoverflow.com/a/41263854/285587)

Other modes

Of course, there is a PDO::FETCH\_FUNC for the functional programming fans.

More modes are coming soon.

### Error handling. Exceptions

Although there are several error handling modes in PDO, the only proper one is PDO::ERRMODE\_EXCEPTION. So, one ought to always set it this way, either by adding this line after creation of PDO instance,

$dbh->setAttribute( PDO::ATTR\_ERRMODE, PDO::ERRMODE\_EXCEPTION );

or as a connection option, as demonstrated in the example above. And this is all you need for the basic error reporting.

#### Reporting PDO errors

TL;DR:  
Despite what all other tutorials say, you don't need a try..catch operator to report PDO errors. Catch an exception only if you have a handling scenario other than just reporting it. Otherwise just let it bubble up to a site-wide handler (note that you don't have to write one, there is a basic built-in handler in PHP, which is quite good).

The only exception (pun not intended) is the creation of the PDO instance, which in case of error might reveal the connection credentials (that would be the part of the stack trace). In order to hide them, we can wrap the connection code into a try..catch operator and then throw a new ErrorException that contains only the message but not the credentials.

A long rant on the matter:

Despite a widespread delusion, you should never catch errors to report them. A module (like a database layer) should not report its errors. This function has to be delegated to an application-wide handler. All we need is to raise an error (in the form of exception) - which we already did. That's all. Nor should you "always wrap your PDO operations in a try/catch" like the most popular tutorial from tutsplus recommends. Quite contrary, catching an exception should be rather an exceptional case (pun intended).

In fact, there is nothing special in PDO exceptions - they are errors all the same. Thus, you have to treat them exactly the same way as other errors. If you had an error handler before, you shouldn't create a dedicated one for PDO. If you didn't care - it's all right too, as PHP is good with basic error handling and will conduct PDO exceptions all right.

Exception handling is one of the problems with PDO tutorials. Being acquainted with exceptions for the first time when starting with PDO, authors consider exceptions dedicated to this library, and start diligently (but improperly) handling exceptions for PDO only. This is utter nonsense. If one paid no special attention to any exceptions before, they shouldn't have changed their habit for PDO. If one didn't use try..catch before, they should keep with that, eventually learning how to use exceptions and when it is suitable to catch them.

So now you can tell that the PHP manual is wrong, [stating that](http://php.net/manual/en/pdo.connections.php)

If your application does not catch the exception thrown from the PDO constructor, the default action taken by the zend engine is to terminate the script and display a back trace. This back trace will likely reveal the full database connection details, including the username and password.

However, there is no such thing as "the displaying of a back trace"! What zend engine really does is just convert an uncaught exception into a fatal error. And then this fatal error is treated like any other error - so it will be displayed only if appropriate php.ini directive is set. Thus, although you may or you may not catch an exception, it has absolutely nothing to do with displaying sensitive information, because it's a totally different configuration setting in response to this. So, do not catch PDO exceptions to report them. Instead, configure your server properly:

On a development server just turn displaying errors on:

ini\_set('display\_errors', 1);

While on a production server turn displaying errors off while logging errors on:

ini\_set('display\_errors', 0);

ini\_set('log\_errors', 1);

* keep in mind that there are other errors that shouldn't be revealed to the user as well.

#### Catching PDO exceptions

You may want to catch PDO errors only in two cases:

1. If you are writing a wrapper for PDO, and you want to augment the error info with some additional data, like query string. In this case, catch the exception, gather the required information, and re-throw another Exception.
2. If you have a certain scenario for handling errors in the particular part of code. Some examples are:
   * if the error can be bypassed, you can use try..catch for this. However, do not make it a habit. Empty catch in every aspect works as error suppression operator, [and so equally evil it is](http://programmers.stackexchange.com/questions/219788/is-error-suppressing-bad-practice).
   * if there is an action that has to be taken in case of failure, i.e. [transaction](https://phpdelusions.net/pdo" \l "transactions) rollback.
   * if you are waiting for a particular error to handle. In this case, catch the exception, see if the error is one you're looking for, and then handle this one. Otherwise just throw it again - so it will bubble up to the handler in the usual way.

E.g.:

try {

$pdo->prepare("INSERT INTO users VALUES (NULL,?,?,?,?)")->execute($data);

} catch (PDOException $e) {

$existingkey = "Integrity constraint violation: 1062 Duplicate entry";

if (strpos($e->getMessage(), $existingkey) !== FALSE) {

// Take some action if there is a key constraint violation, i.e. duplicate name

} else {

throw $e;

}

}

However, in general, no dedicated treatment for PDO exceptions is ever needed. In short, to have PDO errors properly reported:

1. Set PDO in exception mode.
2. Do not use try..catch to report errors.
3. Configure PHP for proper error reporting
   * on a live site set display\_errors=off and log\_errors=on
   * on a development site, you may want to set display\_errors=on
   * of course, error\_reporting has to be set to E\_ALL in both cases

As a result, you will be always notified of all database errors without a single line of extra code! [Further reading](https://phpdelusions.net/try-catch).

### Getting row count with PDO

You don't needed it.

Although PDO offers a function for returning the number of rows found by the query, PDOstatement::rowCount(), you scarcely need it. Really.

If you think it over, you will see that this is a most misused function in the web. Most of time it is used not to *count* anything, but as a mere flag - just to see if there was any data returned. But for such a case you have the data itself! Just get your data, using either fetch() or fetchAll() - and it will serve as such a flag all right! Say, to see if there is any user with such a name, just select a row:

$stmt = $pdo->prepare("SELECT 1 FROM users WHERE name=?");

$stmt->execute([$name]);

$userExists = $stmt->fetchColumn();

Exactly the same thing with getting either a single row or an array with rows:

$data = $pdo->query("SELECT \* FROM table")->fetchAll();

if ($data) {

// You have the data! No need for the rowCount() ever!

}

Remember that here you don't need the *count*, the actual number of rows, but rather a boolean flag. So you got it.

Not to mention that the second most popular use case for this function should never be used at all. One should never use the rowCount() to count rows in database! Instead, one has to ask a database to count them, and return the result in a **single** row:

$count = $pdo->query("SELECT count(1) FROM t")->fetchColumn();

is the only proper way.

In essence:

* if you need to know how many rows in the table, use SELECT COUNT(\*) query.
* if you need to know whether your query returned any data - check that data.
* if you still need to know how many rows has been returned by some query (though I hardly can imagine a case), then you can either use rowCount() or simply call count() on the array returned by fetchAll() (if applicable).

Thus you could tell that the [top answer for this question on Stack Overflow](http://stackoverflow.com/a/883382/285587/) is essentially pointless and harmful - a call to rowCount() could be never substituted with SELECT count(\*) query - their purpose is essentially different, while running an extra query only to get the number of rows returned by other query makes absolutely no sense.

### Affected rows and insert id

PDO is using the same function for returning both number of rows returned by SELECT statement and number of rows affected by queries - PDOstatement::rowCount(). Thus, to get the number of rows affected, just call this function after performing a query.

Another frequently asked question is caused by the fact that mysql won't update the row, if new value is the same as old one. Thus number of rows affected could differ from the number of rows matched by the WHERE clause. Sometimes it is required to know this latter number.

Although you can tell rowCount() to return the number of rows matched instead of rows affected by setting PDO::MYSQL\_ATTR\_FOUND\_ROWS option to TRUE, but, as this is a connection-only option and thus you cannot change it's behavior during runtime, you will have to stick to only one mode for the application, which could be not very convenient.

Note that PDO::MYSQL\_ATTR\_FOUND\_ROWS is not guaranteed to work, as it's described in the [comment](https://phpdelusions.net/pdo" \l "comment-276) below.

Unfortunately, there is no PDO counterpart for the mysql(i)\_info() function which output can be easily parsed and desired number found. This is one of minor PDO drawbacks.

An auto-generated identifier from a sequence or auto\_inclement field in mysql can be obtained from the [PDO::lastInsertId](http://php.net/manual/en/pdo.lastinsertid.php) function. An answer to a frequently asked question, "whether this function is safe to use in concurrent environment?" is positive: yes, it is safe. Being just an interface to MySQL C API [mysql\_insert\_id()](http://dev.mysql.com/doc/refman/5.7/en/mysql-insert-id.html) function it's perfectly safe.

### Prepared statements and LIKE clause

Despite PDO's overall ease of use, there are some gotchas anyway, and I am going to explain some.

One of them is using placeholders with LIKE SQL clause. At first one would think that such a query will do:

$stmt = $pdo->prepare("SELECT \* FROM table WHERE name LIKE '%?%'");

but soon they will learn that it will produce an error. To understand its nature one has to understand that, [like it was said above](https://phpdelusions.net/pdo" \l "bindable), *a placeholder have to represent a complete data literal only* - a string or a number namely. And by no means can it represent either a part of a literal or some arbitrary SQL part. So, when working with LIKE, we have to prepare our complete literal first, and then send it to the query the usual way:

$search = "%$search%";

$stmt = $pdo->prepare("SELECT \* FROM table WHERE name LIKE ?");

$stmt->execute([$search]);

$data = $stmt->fetchAll();

### Prepared statements and IN clause

Just like it was said above, it is impossible to substitute an arbitrary query part with a placeholder. Any string you bind through a placeholder will be put into query as a single string literal. For example, a string '1,2,3' will be bound as a string, resulting in

SELECT \* FROM table WHERE column IN ('1,2,3')

making SQL to search for just one value.

To make it right, one needs separated values, to make a query look like

SELECT \* FROM table WHERE column IN ('1','2','3')

Thus, for the comma-separated values, like for IN() SQL operator, one must create a set of ?s manually and put them into the query:

$arr = [1,2,3];

$in = str\_repeat('?,', count($arr) - 1) . '?';

$sql = "SELECT \* FROM table WHERE column IN ($in)";

$stm = $db->prepare($sql);

$stm->execute($arr);

$data = $stm->fetchAll();

Not very convenient, but compared to mysqli it's *[amazingly](https://phpdelusions.net/pdo/mysqli_comparison" \l "in)* concise.

In case there are other placeholders in the query, you could use array\_merge() function to join all the variables into a single array, adding your other variables in the form of arrays, in the order they appear in your query:

$arr = [1,2,3];

$in = str\_repeat('?,', count($arr) - 1) . '?';

$sql = "SELECT \* FROM table WHERE foo=? AND column IN ($in) AND bar=? AND baz=?";

$stm = $db->prepare($sql);

$params = array\_merge([$foo], $arr, [$bar, $baz]);

$stm->execute($params);

$data = $stm->fetchAll();

In case you are using named placeholders, the code would be a little more complex, as you have to create a sequence of the named placeholders, e.g. :id0,:id1,:id2. So the code would be:

// other parameters that are going into query

$params = ["foo" => "foo", "bar" => "bar"];

$ids = [1,2,3];

$in = "";

$i = 0;// we are using an external counter

// because the actual array key could be dangerous

foreach ($ids as $item)

{

$key = ":id".$i++;

$in .= "$key,";

$in\_params[$key] = $item; // collecting values into key-value array

}

$in = rtrim($in,","); // :id0,:id1,:id2

$sql = "SELECT \* FROM table WHERE foo=:foo AND id IN ($in) AND bar=:bar";

$stm = $db->prepare($sql);

$stm->execute(array\_merge($params,$in\_params)); // just merge two arrays

$data = $stm->fetchAll();

Luckily, for the named placeholders we don't have to follow the strict order, so we can merge our arrays in any order.

### Protecting table and field names

On Stack Overflow I've seen overwhelming number of PHP users implementing [the most fatal PDO code](https://phpdelusions.net/pdo/lame_update), thinking that only data values have to be protected. But of course it is not.

Unfortunately, PDO has no placeholder for identifiers (table and field names), so a developer must manually filter them out. Such a filter is often called a "white list" (where we only list allowed values) as opposed to a "black list" where we list disallowed values. Here is a brief example

$orders = ["name","price","qty"]; // the white list of allowed fierld names

$key = array\_search($\_GET['sort'], $orders); // see if we have such a name

$orderby = $orders[$key]; //if not, first one will be set automatically. smart enuf :)

the same approach should be used for the direction, although the code would be a bit simpler

$direction = \_GET['direction'] == 'DESC' ? 'DESC' : 'ASC';

having gottent these two variables this way will make them 100% safe

$query = "SELECT \* FROM `table` ORDER BY $orderby $direction"; // sound and safe

The same approach must be used every time a table of a field name is going to be used in the query.

### A problem with LIMIT clause

Another problem is related to the SQL LIMIT clause. When in [emulation mode](https://phpdelusions.net/pdo" \l "emulation) (which is on by default), PDO substitutes placeholders with actual data, instead of sending it separately. And with "lazy" binding (using array in execute()), PDO treats every parameter as a string. As a result, the prepared LIMIT ?,? query becomes LIMIT '10', '10' which is invalid syntax that causes query to fail.

There are two solutions:

One is [turning emulation off](https://phpdelusions.net/pdo" \l "emulation) (as MySQL can sort all placeholders properly). To do so one can run this code:

$conn->setAttribute( PDO::ATTR\_EMULATE\_PREPARES, false );

And parameters can be kept in execute():

$conn->setAttribute( PDO::ATTR\_EMULATE\_PREPARES, false );

$stmt = $pdo->prepare('SELECT \* FROM table LIMIT ?, ?');

$stmt->execute([$offset, $limit]);

$data = $stmt->fetchAll();

Another way would be to bind these variables explicitly while setting the proper param type:

$stmt = $pdo->prepare('SELECT \* FROM table LIMIT ?, ?');

$stmt->bindParam(1, $offset,PDO::PARAM\_INT);

$stmt->bindParam(2, $limit,PDO::PARAM\_INT);

$stmt->execute();

$data = $stmt->fetchAll();

One peculiar thing about PDO::PARAM\_INT: for some reason it does not enforce the type casting. Thus, using it on a number that has a string type will cause the aforementioned error:

$stmt = $pdo->prepare("SELECT 1 LIMIT ?");

$stmt->bindValue(1, "1", PDO::PARAM\_INT);

$stmt->execute();

But change "1" in the example to 1 - and everything will go smooth.

### Transactions

To successfully run a transaction, you have to make sure that error mode is set to exceptions, and learn three canonical methods:

* beginTransaction() to start a transaction
* commit() to commit one
* rollback() to cancel all the changes you made since transaction start.

Exceptions are essential for transactions because they can be caught. So in case one of the queries failed, the execution will be stopped and moved straight to the catch block, where the whole transaction will be rolled back.

So a typical example would be like

try {

$pdo->beginTransaction();

$stmt = $pdo->prepare("INSERT INTO users (name) VALUES (?)");

foreach (['Joe','Ben'] as $name)

{

$stmt->execute([$name]);

}

$pdo->commit();

}catch (Exception $e){

$pdo->rollback();

throw $e;

}

Please note the following important things:

* PDO error reporting mode should be set to PDO::ERRMODE\_EXCEPTION
* you have catch an Exception, not PDOException, as it doesn't matter what particular exception aborted the execution.
* you should re-throw an exception after rollback, to be notified of the problem the usual way.
* also make sure that a table engine supports transactions (i.e. for Mysql it should be InnoDB, not MyISAM)
* there are no Data definition language (DDL) statements that define or modify database schema among queries in your transaction, as such a query will cause an implicit commit

### Calling stored procedures in PDO

There is one thing about stored procedures any programmer stumbles upon at first: every stored procedure always returns one extra result set: one (or many) results with actual data and one just empty. Which means if you try to call a procedure and then proceed to another query, then **"Cannot execute queries while other unbuffered queries are active"** error will occur, because you have to clear that extra empty result first. Thus, after calling a stored procedure that is intended to return only one result set, just call [PDOStatement::nextRowset()](http://php.net/manual/en/pdostatement.nextrowset.php) once (of course after fetching all the returned data from statement, or it will be discarded):

$stmt = $pdo->query("CALL bar()");

$data = $stmt->fetchAll();

$stmt->nextRowset();

While for the stored procedures returning many result sets the behavior will be the same as with [multiple queries execution](https://phpdelusions.net/pdo" \l "multiquery):

$stmt = $pdo->prepare("CALL foo()");

$stmt->execute();

do {

$data = $stmt->fetchAll();

var\_dump($data);

} while ($stmt->nextRowset() && $stmt->columnCount());

However, as you can see here is another trick have to be used: remember that extra result set? It is so essentially empty that even an attempt to fetch from it will produce an error. So, we cannot use just while ($stmt->nextRowset()). Instead, we have to check also for empty result. For which purpose PDOStatement::columnCount() is just excellent.

This feature is one of essential differences between old mysql ext and modern libraries: after calling a stored procedure with mysql\_query() there was no way to continue working with the same connection, because there is no nextResult() function for mysql ext. One had to close the connection and then open a new one again in order to run other queries after calling a stored procedure.

Calling a stored procedure is a rare case where bindParam() use is justified, as it's the only way to handle OUT and INOUT parameters. The example can be found in the [corresponding manual chapter](http://php.net/manual/en/pdo.prepared-statements.php" \l "example-1009). However, for mysql it doesn't work. You have to resort to an [SQL variable and an extra call](http://stackoverflow.com/a/23749445/285587).

Note that for the different databases the syntax could be different as well. For example, to run a sored procedure against Microsoft SQL server, use the following format

$stmt = $pdo->prepare("EXEC stored\_procedure ? ?");

where ? marks are placeholders. Note that no braces should be used in the call.

### Running multiple queries with PDO

Note there are no reasons to stuff multiple queries in a single call, and generally you don't need this functionality. Running queries one by one is equal in every way to running them in a batch. The only use case for this functionality I can think of is when you need to execute an existing SQL dump and check for the results.

When in [emulation mode](https://phpdelusions.net/pdo" \l "emulation), PDO can run mutiple queries in the same statement, either via query() or prepare()/execute(). To access the result of consequent queries one has to use [PDOStatement::nextRowset()](http://php.net/manual/en/pdostatement.nextrowset.php):

$stmt = $pdo->prepare("SELECT ?;SELECT ?");

$stmt->execute([1,2]);

do {

$data = $stmt->fetchAll();

var\_dump($data);

} while ($stmt->nextRowset());

Within this loop you'll be able to gather all the related information from the every query, like affected rows, auto-generated id or errors occurred.

It is important to understand that at the point of execute() PDO will report the error for the first query only. But if error occurred at any of consequent queries, to get that error one has to iterate over results. Despite some [ignorant opinions](https://bugs.php.net/bug.php?id=61613), PDO can not and should not report all the errors at once. Some people just cannot grasp the problem at whole, and don't understand that error message is not the only outcome from the query. There could be a dataset returned, or some metadata like insert id. To get these, one has to iterate over resultsets, one by one. But to be able to throw an error immediately, PDO would have to iterate automatically, and thus discard some results. Which would be a clear nonsense.

Unlike mysqli\_multi\_query() PDO doesn't make an asynchronous call, so you can't "fire and forget" - send bulk of queries to mysql and close connection, PHP will wait until last query gets executed.

### Emulation mode. PDO::ATTR\_EMULATE\_PREPARES

One of the most controversial PDO configuration options is PDO::ATTR\_EMULATE\_PREPARES. What does it do? PDO can run your queries in two ways:

1. It can use a **real** or native prepared statement:  
   When prepare() is called, your query with placeholders gets sent to mysql as is, with all the question marks you put in (in case named placeholders are used, they are substituted with ?s as well), while actual data goes later, when execute() is called.
2. It can use **emulated** prepared statement, when your query is sent to mysql as proper SQL, with all the data in place, **properly formatted**. In this case only one roundtrip to database happens, with execute() call. For some drivers (including mysql) emulation mode is turned ON by default.

Both methods has their drawbacks and advantages but, and - I have to stress on it - both being **equally secure**, if used properly. Despite rather appealing tone of the popular [article on Stack Overflow](http://stackoverflow.com/a/12202218/285587), in the end it says that **if you are using supported versions of PHP and MySQL properly, you are 100% safe**. All you have to do is to set encoding in the DSN, as it shown in the [example above](https://phpdelusions.net/pdo" \l "dsn), and your emulated prepared statements will be as secure as real ones.

Note that when native mode is used, the data is never appears in the query, which is parsed by the engine as is, with all the placeholders in place. If you're looking into Mysql query log for your prepared query, you have to understand that it's just an artificial query that has been created solely for logging purpose, but not a real one that has been executed.

Other issues with emulation mode as follows:

#### When emulation mode is turned ON

one can use a handy feature of named prepared statements - a placeholder with same name could be used any number of times in the same query, while corresponding variable have to be bound only once. For some obscure reason this functionality is disabled when emulation mode is off:

$stmt = $pdo->prepare("SELECT \* FROM t WHERE foo LIKE :search OR bar LIKE :search");

$stmt->execute(['search'] => "%$search%");`

Also, when emulation is ON, PDO is able to run [multiple queries in one prepared statement](https://phpdelusions.net/pdo" \l "multiquery).

Also, as native prepared statements support only certain query types, you can run some queries with prepared statements only when emulation is ON. The following code will return table names in emulation mode and error otherwise:

$stmt = $pdo->prepare("SHOW TABLES LIKE ?");

$stmt->execute(["%$name%"]);

var\_dump($stmt->fetchAll());

#### When emulation mode is turned OFF

One could bother not with parameter types, as mysql will sort all the types properly. Thus, even string can be bound to LIMIT parameters, as it was noted in the [corresponding chapter](https://phpdelusions.net/pdo" \l "limit).

Also, this mode will allow to use the advantage of [single prepare-multiple execute](https://phpdelusions.net/pdo" \l "multiexec) feature.

It's hard to decide which mode have to be preferred, but for usability sake I would rather turn it OFF, to avoid a hassle with LIMIT clause. Other issues could be considered negligible in comparison.

### Mysqlnd and buffered queries. Huge datasets.

Recently all PHP extensions that work with mysql database were updated based on a low-level library called mysqlnd, which replaced old libmysql client. Thus some changes in the PDO behavior, mostly described above and one that follows:

There is one thing called [buffered queries](http://php.net/manual/en/mysqlinfo.concepts.buffering.php). Although you probably didn't notice it, you were using them all the way. Unfortunately, here are bad news for you: unlike old PHP versions, where you were using buffered queries virtually for free, modern versions built upon [mysqlnd driver](http://php.net/manual/en/book.mysqlnd.php) won't let you to do that anymore:

When using libmysqlclient as library PHP's memory limit won't count the memory used for result sets unless the data is fetched into PHP variables. With mysqlnd the memory accounted for will include the full result set.

The whole thing is about a resultset, which stands for all the data found by the query.

When your SELECT query gets executed, there are two ways to deliver the results in your script: buffered and unbuffered one. When buffered method is used, all the data returned by the query gets copied in the script's memory at once. While in unbuffered mode a database server feeds the found rows one by one.

So you can tell that in buffered mode a resultset is always burdening up the memory on the server even if fetching weren't started at all. Which is why it is not advisable to select huge datasets if you don't need all the data from it.

Nonetheless, when old libmysql-based clients were used, this problem didn't bother PHP uers too much, because the memory consumed by the resultset didn't count in the the memory\_get\_usage() and memory\_limit.

But with mysqlnd things got changed, and the resultset returned by the buffered query will be count towards both memory\_get\_usage() and memory\_limit, no matter which way you choose to get the result:

$pdo->query("create temporary table pdo\_memtest (i int)");

$pdo->query("insert into pdo\_memtest values (1),(1),(1),(1),(1),(1),(1),(1),(1),(1)");

$sql = "SELECT REPEAT(' ', 1024 \* 1024) as big FROM pdo\_memtest";

$pdo->setAttribute(PDO::MYSQL\_ATTR\_USE\_BUFFERED\_QUERY, FALSE);

$mem = memory\_get\_usage();

$stmt = $pdo->query($sql);

while($row = $stmt->fetch());

echo "Memory used: ".round((memory\_get\_usage() - $mem) / 1024 / 1024, 2)."M\n";

$pdo->setAttribute(PDO::MYSQL\_ATTR\_USE\_BUFFERED\_QUERY, TRUE);

$mem = memory\_get\_usage();

$stmt = $pdo->query($sql);

while($row = $stmt->fetch());

echo "Memory used: ".round((memory\_get\_usage() - $mem) / 1024 / 1024, 2)."M\n";

will give you

Memory used: 1M

Memory used: 20.04M

which means that with buffered query the memory is consumed even if you're fetching rows one by one!

So, keep in mind that if you are selecting a really huge amount of data, always set PDO::MYSQL\_ATTR\_USE\_BUFFERED\_QUERY to FALSE.

Of course, there are drawbacks. One is infamous

Cannot execute queries while other unbuffered queries are active

error message which means that until you won't retrieve all the selected rows from the unbuffered query, it will be impossible to run any other query against hte same database connection.

And a couple minor ones,

1. With unbuffered query you can't use rowCount() method (which is useless, as we learned [above](https://phpdelusions.net/pdo" \l "count))
2. Moving (seeking) the current resultset internal pointer back and forth (which is useless as well).

### Related articles:

* [Simple yet efficient PDO wrapper](https://phpdelusions.net/pdo/pdo_wrapper)
* [Usability problems of mysqli compared to PDO](https://phpdelusions.net/pdo/mysqli_comparison)
* [PDO Fetch Modes](https://phpdelusions.net/pdo/fetch_modes)
* [An SQL injection against which prepared statements won't help](https://phpdelusions.net/pdo/sql_injection_example)
* [Your first database wrapper's childhood diseases](https://phpdelusions.net/pdo/common_mistakes)
* [Fetching objects with PDO](https://phpdelusions.net/pdo/objects)
* [MCVE or How to debug database interactions with PDO](https://phpdelusions.net/pdo/mcve)
* [Authenticating a user using PDO and password\_verify()](https://phpdelusions.net/pdo/password_hash)
* [A cargo cult prepared statement](https://phpdelusions.net/pdo/cargo_cult_prepared_statement)
* [Whitelisting helper function](https://phpdelusions.net/pdo/whitelisting_helper_function)

# PDO Fetch Modes

1. [Classic Modes](https://phpdelusions.net/pdo/fetch_modes" \l "classic)
   * [PDO::FETCH\_BOTH](https://phpdelusions.net/pdo/fetch_modes" \l "FETCH_BOTH)
   * [PDO::FETCH\_NUM](https://phpdelusions.net/pdo/fetch_modes" \l "FETCH_NUM)
   * [PDO::FETCH\_ASSOC](https://phpdelusions.net/pdo/fetch_modes" \l "FETCH_ASSOC)
   * [PDO::FETCH\_OBJ](https://phpdelusions.net/pdo/fetch_modes" \l "FETCH_OBJ)
2. [PDO::FETCH\_LAZY](https://phpdelusions.net/pdo/fetch_modes" \l "FETCH_LAZY)
3. [Most useful modes](https://phpdelusions.net/pdo/fetch_modes" \l "best)
   * [PDO::FETCH\_COLUMN](https://phpdelusions.net/pdo/fetch_modes" \l "FETCH_COLUMN)
   * [PDO::FETCH\_KEY\_PAIR](https://phpdelusions.net/pdo/fetch_modes" \l "FETCH_KEY_PAIR)
   * [PDO::FETCH\_UNIQUE](https://phpdelusions.net/pdo/fetch_modes" \l "FETCH_UNIQUE)
   * [PDO::FETCH\_GROUP](https://phpdelusions.net/pdo/fetch_modes" \l "FETCH_GROUP)
4. [OOP](https://phpdelusions.net/pdo/fetch_modes" \l "object)
   * [PDO::FETCH\_CLASS](https://phpdelusions.net/pdo/fetch_modes" \l "FETCH_CLASS)
   * [PDO::FETCH\_PROPS\_LATE](https://phpdelusions.net/pdo/fetch_modes" \l "FETCH_PROPS_LATE)
   * [PDO::FETCH\_CLASSTYPE](https://phpdelusions.net/pdo/fetch_modes" \l "FETCH_CLASSTYPE)
   * [PDO::FETCH\_INTO](https://phpdelusions.net/pdo/fetch_modes" \l "FETCH_INTO)
   * [PDO::FETCH\_SERIALIZE](https://phpdelusions.net/pdo/fetch_modes" \l "FETCH_SERIALIZE)
5. [Miscellaneous](https://phpdelusions.net/pdo/fetch_modes" \l "misc)
   * [PDO::FETCH\_FUNC](https://phpdelusions.net/pdo/fetch_modes" \l "FETCH_FUNC)
   * [PDO::FETCH\_NAMED](https://phpdelusions.net/pdo/fetch_modes" \l "FETCH_NAMED)
   * [PDO::FETCH\_BOUND](https://phpdelusions.net/pdo/fetch_modes" \l "FETCH_BOUND)
   * [PDO::FETCH\_ORI\_NEXT at al](https://phpdelusions.net/pdo/fetch_modes" \l "FETCH_ORI_NEXT)
   * [Comments (9)](https://phpdelusions.net/pdo/fetch_modes" \l "comments)

Like it is said in the [main article](https://phpdelusions.net/pdo), PDO fetch modes (along with usable prepared statements) is a thing that makes PDO a wrapper, not yet another (though universal) database API. These modes could dramatically reduce the amount of code for routine operations, as they let you to get the data in the proper format right out of the query.

Although constants in question are partially described on the manual pages for [fetch()](http://php.net/manual/en/pdostatement.fetch.php) and [fetchAll()](http://php.net/manual/en/pdostatement.fetchall.php) methods, the full list can be found only on the [page with all PDO constants](http://php.net/manual/en/pdo.constants.php), where they are just scattered among other constants with descriptions that hardly exceed a few words. It doesn't seem too convenient and, most likely is the reason, why some interesting modes escaped attention of a general PHP audience. I decided to take the fetch modes out of the global list and split them into several categories for convenience, as total number of PDO fetch modes amounts to a decent figure of 23!

Moreover, some modes can be combined together, increasing the number of possible return formats even mere. However, one have to keep in mind that not all the constants' values are powers of 2, and thus some constants cannot be combined. For example, PDO::FETCH\_FUNC has a bit unexpected value of 10 (which is equal to PDO::FETCH\_CLASS|PDO::FETCH\_ASSOC combination - though quite useless by itself).

### Classic Modes

For starter let's list the modes that resemble the behavior of old good mysql functions.

#### PDO::FETCH\_BOTH

A counterpart for mysql\_fetch\_array(). The row is returned in the form of array, where data is duplicated, to be accessed via both numeric and associative indexes. This mode is set by default.

$user = $pdo->query("SELECT \* from users LIMIT 1")->fetch(PDO::FETCH\_BOTH);

/\*

array (

'id' => '104',

0 => '104',

'name' => 'John',

1 => 'John',

'sex' => 'male',

2 => 'male',

'car' => 'Toyota',

3 => 'Toyota',

)\*/

#### PDO::FETCH\_NUM

An old pal again, a mysql\_fetch\_row() counterpart, numeric indices only:

$user = $pdo->query("SELECT \* from users LIMIT 1")->fetch(PDO::FETCH\_NUM);

/\*

array (

0 => '104',

1 => 'John',

2 => 'male',

3 => 'Toyota',

)\*/

#### PDO::FETCH\_ASSOC

Same thing, but for mysql\_fetch\_assoc() - associative indices only. See also [PDO::FETCH\_NAMED](https://phpdelusions.net/pdo/fetch_modes" \l "FETCH_NAMED)

$user = $pdo->query("SELECT \* from users LIMIT 1")->fetch(PDO::FETCH\_ASSOC);

/\*

array (

'id' => '104',

'name' => 'John',

'sex' => 'male',

'car' => 'Toyota',

)\*/

#### PDO::FETCH\_OBJ

like mysql\_fetch\_object() but without class name provided. with no class name provided. Returns stdClass instance.

$user = $pdo->query("SELECT \* from users LIMIT 1")->fetch(PDO::FETCH\_OBJ);

/\*

stdClass::\_\_set\_state(array(

'id' => '104',

'name' => 'John',

'sex' => 'male',

'car' => 'Toyota',

))\*/

Note that PDO is not that silly with returning objects. [A whole dedicated section on object manipulation follows below](https://phpdelusions.net/pdo/fetch_modes" \l "object).

### PDO::FETCH\_LAZY

This constant is so remarkable that I decided to dedicate an entire section to it. For starter, the returned value being an instance of a special class PDORow, and this object bears a whole lot of peculiar features:

* First, this variable doesn't contain any row data, but just return it on demand (so the name suggests)
* Instead, it contains a single mysterious variable called queryString to which the SQL query is assigned. (Which reminds us a [PDOStatement](http://php.net/manual/ru/class.pdostatement.php) class):

$lazy = $pdo->query("SELECT name FROM users")->fetch(PDO::FETCH\_LAZY);

/\*

object(PDORow)#3 (2) {

["queryString"] => string(22) "SELECT name FROM users"

["name"] => string(4) "John"

}\*/

For the curious - yes, you can overwrite it :)

* Third, you cannot store this object in a session (or, in other words, this object is not serializable)
* Fourth, you can get the data in any way - by using a numeric index, an associative index or via the object property.
* Fifth, calling an undefined index/property doesn't raise an "Undefined property / index" error. NULL value is silently returned.
* Sixth, this variable is changing its state after consequent fetch() calls.
* Seventh, this constant cannot be used with fetchAll(), but only with fetch().

Let's do some tests. We'll try to request a considerable amount of data and watch the memory consumption, verifying a couple things along the way:

$stmt = $pdo->query("SELECT \*, REPEAT(' ', 1024 \* 1024) big FROM users");

echo 'start ', round(memory\_get\_usage() / 1024), PHP\_EOL;

$lazy = $stmt->fetch(PDO::FETCH\_LAZY);

echo 'lazy fetch ', round(memory\_get\_usage() / 1024), PHP\_EOL;

$big = $lazy[3];

echo 'lazy assign ', round(memory\_get\_usage() / 1024), PHP\_EOL;

echo 'lazy name ', $lazy[0], PHP\_EOL;

echo 'lazy undef ', var\_dump($lazy['undef']);

echo '------------', PHP\_EOL;

$num = $stmt->fetch(PDO::FETCH\_NUM);

echo 'num fetch ', round(memory\_get\_usage() / 1024), PHP\_EOL;

$big2 = $num[3];

echo 'num assign ', round(memory\_get\_usage() / 1024), PHP\_EOL;

$big2 .= ''; // to invoke a copy-on-write

echo 'num assign2 ', round(memory\_get\_usage() / 1024), PHP\_EOL;

echo 'lazy name ', $lazy[0], PHP\_EOL;

echo 'num undef ', var\_dump($num['undef']);

Output:

start 228

lazy fetch 228

lazy assign 1252

lazy name John

lazy undef NULL

------------

num fetch 2277

num assign 2277

num assign2 3301

lazy name Mike

num undef

Notice: Undefined index: undef in pdo.php on line 48

NULL

As you can see, this code is fetching all records from the users table, adding one column of the size 1Mb. First fetch is done with PDO::FETCH\_LAZY and it can be seen that it doesn't change the amount of memory consumed (note that this behavior can be changed depends on the [buffering mode used](https://phpdelusions.net/pdo" \l "buffering)). After that we are fetching another row, using one of conventional methods and watching the immediate memory consumption increase. So the memory save is evident. Also we observed the "Undefined variable" error absence, and the status change after a consequent fetch() call.

From this we can conclude that PDORow object is just a link to the resultset. Having no own state, it is just reading the data from the current cursor position. Given all that, it is quite surprising to see this mode so rarely used.

### Most useful modes

Here I decided to list the most useful modes, that return the data in most demanded formats. All these modes make sense only with fetchAll().

Note that you can always combine several fetch modes to alter the result format. All the examples below are given with PDO::FETCH\_ASSOC set as a default fetch format.

#### PDO::FETCH\_COLUMN

It is often very handy to get plain one-dimensional array right out of the query, if only one column out of many rows being fetched. Here you go:

$data = $pdo->query('SELECT name FROM users')->fetchAll(PDO::FETCH\_COLUMN);

/\* array (

0 => 'John',

1 => 'Mike',

2 => 'Mary',

3 => 'Kathy',

) \*/

#### PDO::FETCH\_KEY\_PAIR

Also extremely useful format, when we need to get the same column as above, but indexed not by numbers in order but by an unique field:

$data = $pdo->query('SELECT name, car FROM users')->fetchAll(PDO::FETCH\_KEY\_PAIR);

/\* array (

'John' => 'Toyota',

'Mike' => 'Ford',

'Mary' => 'Mazda',

'Kathy' => 'Mazda',

)

The returned key-value format is excellent for the dictionary like data or simply for indexed values, like below

$data = $pdo->query('SELECT sex, count(\*) FROM users GROUP BY sex')->fetchAll(PDO::FETCH\_KEY\_PAIR);

/\* array (

'male' => 2,

'female' => 2,

)

Note that you have to select only two columns for this mode, first of which have to be unique.

#### PDO::FETCH\_UNIQUE

Same as above, but getting not one column but full row, yet indexed by an unique field

$data = $pdo->query('SELECT \* FROM users')->fetchAll(PDO::FETCH\_UNIQUE);

/\* array (

104 => array (

'name' => 'John',

'car' => 'Toyota',

),

110 => array (

'name' => 'Mike',

'car' => 'Ford',

),

120 => array (

'name' => 'Mary',

'car' => 'Mazda',

),

121 => array (

'name' => 'Kathy',

'car' => 'Mazda',

),

)\*/

here you get the data array indexed by id (Note that the first column selected have to be unique. In this query it is assumed that first column is id, but to be sure better list it up explicitly). Or you can use any other unique field as well:

$data = $pdo->query('SELECT name, users.\* FROM users')->fetchAll(PDO::FETCH\_UNIQUE);

/\* array (

'John' => array (

'sex' => 'male',

'car' => 'Toyota',

),

'Mike' => array (

'sex' => 'male',

'car' => 'Ford',

),

'Mary' => array (

'sex' => 'female',

'car' => 'Mazda',

),

'Kathy' => array (

'sex' => 'female',

'car' => 'Mazda',

),

) \*/

#### PDO::FETCH\_GROUP

This mode groups the returned rows into a nested array, where indexes will be unique values from the first column, and values will be arrays similar to ones returned by regular fetchAll(). The following code, for example, will separate boys from girls and put them into different arrays:

$data = $pdo->query('SELECT sex, name, car FROM users')->fetchAll(PDO::FETCH\_GROUP);

/\* array (

'male' => array ( 0 =>

array (

'name' => 'John',

'car' => 'Toyota',

),

1 => array (

'name' => 'Mike',

'car' => 'Ford',

),

),

'female' => array (

0 => array (

'name' => 'Mary',

'car' => 'Mazda',

),

1 => array (

'name' => 'Kathy',

'car' => 'Mazda',

),

),

) \*/

So, this is the ideal solution for such a popular demand like "group events by date" or "group goods by category".

This mode could be combined with PDO::FETCH\_COLUMN:

$sql = "SELECT sex, name FROM users";

$data = $pdo->query($sql)->fetchAll(PDO::FETCH\_GROUP|PDO::FETCH\_COLUMN);

/\*

array (

'male' =>

array (

0 => 'John',

1 => 'Mike',

),

'female' =>

array (

0 => 'Mary',

1 => 'Kathy',

),

)\*/

A hint: in case you need to select all the fields, but group by not the first one, the first idea that sporings in the mind won't work:

SELECT sex, \* FROM users

will return an error. To avoid it, just perpend the asterisk with the table name:

SELECT sex, users.\* FROM users

now it works like a charm!

### OOP

Of course, PDO's ability to manipulate objects is not limited to returning a silly stdObject instance. In the following section we will learn a whole bunch of object-oriented fetch modes.

#### PDO::FETCH\_CLASS

The cornerstone of object manipulation in PDO. Creates an instance of a class with a given name, mapping returned columns to the class' properties. This mode can be used to get either a single row or an array of rows from database. With fetchAll() the approach is quite familiar:

class User {};

$users = $pdo->query('SELECT name FROM users')->fetchAll(PDO::FETCH\_CLASS, 'User');

will give you an array consists of objects of a User class, with properties filled from returned data:

array(2) {

[0]=> object(User)#3 (1) {

["name"] => string(4) "John"

}

[1]=> object(User)#4 (1) {

["name"]=> string(4) "Mike"

}

}

While to get a single row you have two options. However, although you could use the the familiar fetch() method, like shown below:

class User {};

$stmt = $pdo->query('SELECT name FROM users LIMIT 1');

$stmt->setFetchMode(PDO::FETCH\_CLASS, 'User');

$user = $stmt->fetch();

it is recommended to use a dedicated fetchObject() method:

class User {};

$user = $pdo->query('SELECT name FROM users LIMIT 1')->fetchObject('User');

as there are several issues with using fetch() to get an object:

* you cannot pass constructor parameters to a newly created object
* if no class defined with such a name, an array will be silently returned, while with fetchObject() an error will be thrown
* obviously, the dedicated method takes less code to write.

No matter which method you choose, all the returned columns will be assigned to the corresponding class' properties according to the following rules:

* if there is a class property, which name is the same as a column name, the column value will be assigned to this property.
* if there is no such property, then a magic \_\_set() method will be called.
* if \_\_set() method is not defined for the class, then a public property will be created and a column value assigned to it.

For example, this code

class User

{

public $name;

}

$user = $pdo->query('SELECT \* FROM users LIMIT 1')->fetchObject('User');

will give you an object with all the properties automatically assigned, no matter were they ixist in the class or not:

object(User)#3 (4) {

["id"] => string(3) "104"

["name"] => string(4) "John"

["sex"] => string(4) "male"

["car"] => string(6) "Toyota"

}

From this you can tell that to avoid an automated property creation you need to use the magic \_\_set() method to filter the properties out. The simplest filtering technique would be just adding an empty \_\_set() method to your object. With it, only existing properties will be set:

class User

{

private $name;

public function \_\_set($name, $value) {}

}

$user = $pdo->query('SELECT \* FROM users LIMIT 1')->fetchObject('User');

array(1) {

[0]=> object(User)#3 (1) {

["name":"User":private]=> string(4) "John"

}

}

As you can see, in this mode PDO can assign values to private properties as well. Which is a bit unexpected but extremely useful.

Of course, for the newly created classes we may want to supply constructor parameters. So, let's add them to the examples above:

class User {

public function \_\_construct($car) {

$this->car = $car;

}

}

$users = $pdo->query('SELECT name FROM users LIMIT 1')

->fetchAll(PDO::FETCH\_CLASS, 'User', ['Caterpillar']);

// or using fetch()

$stmt = $pdo->query('SELECT name FROM users LIMIT 1');

$stmt->setFetchMode(PDO::FETCH\_CLASS, 'User', ['Caterpillar']);

$user = $stmt->fetch();

will give you

/\*

object(User)#3 (2) {

["name"] => string(4) "John"

["car"] => string(11) "Caterpillar"

} \*/

As you can see, in this mode PDO assign class properties before calling a constructor. To amend this behavior, use the following flag:

#### PDO::FETCH\_PROPS\_LATE

Not a fetch mode but a modifier flag. As it was noted above, by default PDO assigns class properties before calling a constructor. This behavior can be changed using this flag:

class User {

public function \_\_construct($car) {

$this->car = $car;

}

}

$stmt = $pdo->query('SELECT name, car FROM users LIMIT 1');

$stmt->setFetchMode(PDO::FETCH\_CLASS|PDO::FETCH\_PROPS\_LATE, 'User', ['Caterpillar']);

$user = $stmt->fetch();

/\*

object(User)#3 (2) {

["car"] => string(6) "Toyota"

["name"] => string(4) "John"

} \*/

As you can see, a value assigned in the constructor has been overwritten.

#### PDO::FETCH\_CLASSTYPE

One more modifier flag which tells PDO to get the class name from the first column's value. With this flag one can avoid using setFetchMode() with fetch():

$data = $pdo->query("SELECT 'User', name FROM users")

->fetch(PDO::FETCH\_CLASS | PDO::FETCH\_CLASSTYPE);

/ \*

object(User)#3 (1) {

["name"]=> string(4) "John"

} \*/

Besides, as it was [noted in the comments](https://phpdelusions.net/pdo" \l "comment-56) to the main article, this mode can be useful if objects of different classes can be created from the same query

$stmt = $pdo->query('SELECT sex, name FROM users');

$users = $stmt->fetchAll(PDO::FETCH\_CLASS | PDO::FETCH\_CLASSTYPE);

/\*

array(6) {

[0]=> object(Male)#3 (1) {

["name"]=> string(4) "John"

}

[1]=> object(Male)#4 (1) {

["name"]=> string(4) "Mike"

}

[2]=> object(Female)#5 (1) {

["name"]=> string(4) "Mary"

}

[3]=> object(Female)#6 (1) {

["name"]=> string(5) "Kathy"

}

}\*/

#### PDO::FETCH\_INTO

Unlike PDO::FETCH\_CLASS, doesn't create a new object but update the existing one. Works with setFetchMode() only, which takes the existing variable as a parameter. Obviously, useless with fetchAll().

class User

{

public $name;

public $state;

public function \_\_construct()

{

$this->name = NULL;

}

}

$user = new User;

$user->state = "up'n'running";

var\_dump($user);

$stmt = $pdo->query('SELECT name FROM users LIMIT 1');

$stmt->setFetchMode(PDO::FETCH\_INTO, $user);

$data = $stmt->fetch();

var\_dump($data, $user);

/\*

object(Foo)#2 (2) {

["name"] => NULL

["state"] => string(12) "up'n'running"

}

object(Foo)#2 (2) {

["name"] => string(4) "John"

["state"] => string(12) "up'n'running"

}

object(Foo)#2 (2) {

["name"] => string(4) "John"

["state"] => string(12) "up'n'running"

} \*/

As you can see, fetch() call is returning the same object, which seems redundant to me. Also note that unlike PDO::FETCH\_CLASS, this mode doesn't assign private properties.

#### PDO::FETCH\_SERIALIZE

One more flag for PDO::FETCH\_CLASS. Allegedly should return an object that was serialized and stored in a database. At the moment doesn't work.

Intended to do something like

class foo {}

$foo = new foo;

$foo->status="up'n'running";

$sFoo = serialize($foo);

// storing $sFoo in a database

// and then something like

$stmt = $pdo->query('SELECT sFoo FROM table');

$stmt->setFetchMode(PDO::FETCH\_CLASS|PDO::FETCH\_SERIALIZE, 'foo');

$foo = $stmt->fetch();

But the object returned from a database is not the same as it was before serialization! And thus unserialize fails. After toying around this mode with a friend, we submitted a [bug](https://bugs.php.net/bug.php?id=68802), but nothing seems changed since.

Either way, this flag looks a rather useless, as one could always unserialize manually:

class User {

protected $name = "John";

}

$serialized = serialize(new User);

var\_dump($serialized);

$stmt = $pdo->prepare('INSERT INTO sertest VALUES(?)')->execute([$serialized]);

$user = unserialize($pdo->query('SELECT \* FROM sertest')->fetchColumn());

var\_dump($user);

/\*

string(40) "O:4:"User":1:{s:7:"\*name";s:4:"John";}"

object(User)#2 (1) {

["name":protected]=>string(4) "John"

} \*/

### Miscellaneous

#### PDO::FETCH\_FUNC

For the closure lovers. Works with fetchAll() only. Not very convenient as you should list parameters for the every returned column manually. For example, a PDO::FETCH\_COLUMN emulator:

$data = $pdo

->query('SELECT name FROM users')

->fetchAll(PDO::FETCH\_FUNC, function($first) {return $first;});

#### PDO::FETCH\_NAMED

Almost exactly the same as PDO::FETCH\_ASSOC, but with one little difference. Many times I've seen a question, whether it's possible to distinguish different fields with the same names that returned by same query. With the only answer is using aliases in SQL or numeric indices instead of associative. However, PDO offers another way. If this mode is used, returned values are assigned the same way as with PDO::FETCH\_ASSOC, but if there are several columns with the same name in the result set, all values are stored in the nested array. For example, let's try to select from users and companies, while both tables has a column name. Using PDO::FETCH\_ASSOC, we'll lose one of the names:

$data = $pdo->query("SELECT \* FROM users, companies WHERE users.name=username")->fetch();

/\*

array(3) {

["name"] => string(10) "ACME, Inc."

["sex"] => string(4) "male"

["username"] => string(4) "John"

}\*/

While if PDO::FETCH\_NAMED is used instead, everything will be kept intact:

$data = $pdo->query("SELECT \* FROM users, companies WHERE users.name=username")

->fetch(PDO::FETCH\_NAMED);

/\*

array(3) {

["name"]=> array(2) {

[0]=> string(4) "John"

[1]=> string(10) "ACME, Inc."

}

["sex"] => string(4) "male"

["username"] => string(4) "John"

}

I doubt whether this feature is useful at all, but at least it gives you a choice.

#### PDO::FETCH\_BOUND

An interesting mode, essentially different from others. It doesn't return any data by itself, but makes PDO assign values to variables that has been previously bound using bindColumn() - the behavior that can be familiar to mysqli users. [And example can be seen in the manual page](http://php.net/manual/en/pdostatement.bindcolumn.php" \l "example-982)

#### PDO::FETCH\_ORI\_NEXT at al

6 modes to operate the cursor. [An example from the manual](http://php.net/manual/en/pdostatement.fetch.php" \l "example-1053)

<https://phpdelusions.net/pdo/fetch_modes>

# Authenticating a user using PDO and password\_verify()

1. [Comments (11)](https://phpdelusions.net/pdo_examples/password_hash" \l "comments)

That's extremely popular question on various forums and Stack Overflow. An at the same time it's a very good example that can show you how to use PDO properly.

First of all make sure that your passwords are stored in the database using [password\_hash()](http://php.net/manual/en/function.password-hash.php) function.

Assuming we've already got a [valid PDO instance](https://phpdelusions.net/pdo" \l "dsn) in the variable called $pdo, while user's credentials are coming from POST request, here is the code you need:

$stmt = $pdo->prepare("SELECT \* FROM users WHERE email = ?");

$stmt->execute([$\_POST['email']]);

$user = $stmt->fetch();

if ($user && password\_verify($\_POST['pass'], $user['pass']))

{

echo "valid!";

} else {

echo "invalid";

}

As you can see, some tricks are used to make this code less bloated.

In the first line we are creating a PDO prepared statement, from a query where the actual data is substituted with a question mark - a placeholder.  
In the second line we are executing the query, sending the data apart from the query - so it can't do any harm, intentional or non-intentional.  
And in the third line we are simply fetching a row from a table.

The next line is a little trick: we are checking both whether our query returned any data at all, and - only in case it did! - verifying the password. Clean, concise and neat.

https://phpdelusions.net/pdo\_examples/password\_hash

# Whitelisting helper function

1. [Comments](https://phpdelusions.net/pdo/whitelisting_helper_function" \l "comments)

There are edge cases when we cannot use a prepared statement for a variable to be used in the query. Given placeholders can substitute only data literals (i.e. strings and numbers), we need something to protect other query pars, such as keywords or identifiers that may happen to be added to the query dynamically.

In this case we have to use the white list approach, filtering our data against a predefined list of allowed values. To ease the routine, I wrote a handy function that will let us to get the safe value in one line! Here it is:

function white\_list(&$value, $allowed, $message) {

if ($value === null) {

return $allowed[0];

}

$key = array\_search($value, $allowed, true);

if ($key === false) {

throw new InvalidArgumentException($message);

} else {

return $value;

}

}

The function accepts three parameters:

* the value to be checked. It is passed by reference so it won't raise an error in case a variable is not set. It would allow us to assign a default value if no value is provided
* the list of allowed values. The first one would serve as a default value
* the error message to throw so a programmer would know what caused the error

Having such a function at hand we can have our code much more tidy and concise:

$orderby = white\_list($\_GET['orderby'], ["name","price","qty"], "Invalid field name");

$direction = white\_list($\_GET['direction'], ["ASC","DESC"], "Invalid ORDER BY direction");

$query = "SELECT \* FROM `table` ORDER BY `$orderby` $direction"; // sound and safe!

https://phpdelusions.net/pdo/whitelisting\_helper\_function

# Top 10

1. [Mysql(i)\_real\_escape\_string prevents SQL injection](https://phpdelusions.net/top" \l "escaping)
2. [Second order SQL injection](https://phpdelusions.net/top" \l "second_order_sql_injection)
3. [Escaping user input to prevent SQL injection](https://phpdelusions.net/top" \l "escaping_user_input)
4. [Mysqli is for the newbies while PDO is for the advanced folks](https://phpdelusions.net/top" \l "mysqli_vs_pdo)
5. [Extensive use of the number of rows returned by a SELECT query](https://phpdelusions.net/top" \l "num_rows)
6. [Disabling errors for security reasons](https://phpdelusions.net/top" \l "zero_error_reporting)
7. [If (isset($var) && !empty($var))](https://phpdelusions.net/top" \l "empty)
8. [Try and catch to echo an error](https://phpdelusions.net/top" \l "try_catch)
9. [HTTP\_X\_FORWARDED\_FOR, HTTP\_X\_CLIENT\_IP etc. to get the "real" IP](https://phpdelusions.net/top" \l "ip)
10. [Single quotes are faster than double](https://phpdelusions.net/top" \l "quotes)
11. [You name it?](https://phpdelusions.net/top" \l "yours)
12. [Comments (8)](https://phpdelusions.net/top" \l "comments)

Initially, this site was launched as a place where I can treat PHP delusions: disprove numerous fallacies and superstitions circulating in the PHP community. Eventually I learned that it's not enough to disclose a misconception or a wrong practice - the right way should be shown as well, so for the time being I switched to writing educational articles. But seeing the same delusions expressed again and again on Stack Overflow or other forums around the Net, I decided to compile a sort of Top 10 list, the most frequent PHP delusions. Some of them are quite important, whereas others are but nitpicks. But they all indicate a cargo cult programming, a practice then someone mindlessly uses a code due to common belief/practice but don't really get the idea of it.

### Mysql(i)\_real\_escape\_string prevents SQL injection

I'd say that up to this day most PHP folks still believe in this nonsense, all thanks to a stupid note from the PHP manual: [*"This function must always (with few exceptions) be used to make data safe"*](http://php.net/manual/en/function.mysql-real-escape-string.php). "To make data safe", my foot!

In reality this function has absolutely nothing to do with safety or injections, merely escaping special characters in SQL string literals, making them immune to SQL injection as a side effect, but being utterly useless for any other query part, from a [table name](https://phpdelusions.net/pdo/sql_injection_example" \l "escaping) to a [numeric literal](https://phpdelusions.net/sql_injection" \l "whatis). To protect from SQL injection, one should follow two simple rules:

1. Any variable data literal (i.e. a string or a number) should be substituted with a parameter, whereas actual value should be sent to the query separately, through bind/execute process.
2. All other query parts that happen to be added through a variable, should be explicitly filtered through a hardcoded list of allowed values.

### Second order SQL injection

Many people take second order injection wrong, thanks to extremely confusing wording in [this answer on Stack Overflow](https://stackoverflow.com/a/134138/285587). People often make it that a prepared statement protects from the 1st order but not from 2nd. Which is plainly wrong.

In reality, it's quite the opposite: second order SQL injection happens only if you neglect a prepared statement. It happens when you are foolish enough to separate the sheep from the goat, to decide whether some certain data is safe or not, and therefore whether to use a prepared statement or not. Eventually such an approach will lead to second order injection. Whereas if you're using prepared statements all over the place, no 2nd or 99nd order injection is ever possible.

### Escaping user input to prevent SQL injection

A nasty one. OWASP up to this day [lists it as one of its recommendations](https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet" \l "Defense_Option_4:_Escaping_All_User-Supplied_Input), let alone hordes of PHP users reciting this sermon on a daily basis. Whereas you can tell from the above, it's a complete rubbish, in the meaning both parts are wrong:

* escaping. As we learned above, escaping has nothing to do with safety or injections. What we need is a prepared statement, not "escaping" for the data (and a white list for the everything else)
* user input is a vague and misleading term, implying that only what is coming from the user side should be treated somehow. But as we learned above, it's a nasty delusion that could lead to the 2nd order SQL injection. There are many reasons why there should never be such a term as "user input" in regard of SQL injection:
  + in a big layered application, a database layer just have no idea what the data source is, and have no means to tell "safe" data from "unsafe". All data should be processed uniformly.
  + protection from SQL injection also protects from syntax errors as well, so there is again not a single reason to avoid a prepared statement for the input that is from the super-trusted source.
  + there is a second order SQL injection when the data technically comes from within your own application
  + user input is a vague term that anyone treats differently, according to their knowledge. There is a [famous question](https://stackoverflow.com/q/22534183/285587) On Stack Overflow based on the impression that a hardcoded form value is not from the user but from the server. Go figure how many other false interpretations this notorious term could have.

### Mysqli is for the newbies while PDO is for the advanced folks

In reality it's the opposite. Given the main difference between the old mysql ext and mysqli is support for the prepared statements AND a whole lot of trouble when using prepared statements with mysqli (one can't even get a usual assoc array or num rows from a prepared statement without a certain trick) on one hand; and consistent, predictable and useful API that PDO offers on the other hand, mysqli should be only recommended for someone who certainly knows what are they doing and have a decent experience in programming.

### Extensive use of the number of rows returned by a SELECT query

It can be seen in almost every PHP script, yet every time you have an idea to use it, it's either superfluous or harmful.  
And for the only case when it can be of any use, it is not available at all.

* it is often used by newbies to get the number of rows matching some criteria. That's an awful idea that makes your database to actually select all the matched rows from the table and send them to PHP, along with this silly number. The right way would be to ask a database to return the only count, by issuing a SELECT count(\*) query.
* it is used to see whether a query returned any data. Which is pointless as there is always the data itself:
  + if you're selecting only one row, then just fetch that row and use it in any condition that's intended to tell whether there is any result or not
  + in case you are expecting multiple rows, then fetch them all into an array and then use this array exactly the same way as above. If, for some reason, you need the actual number of rows returned, you can use count() against this array, but I cannot think of any useful application for such a number.
* the only case I can think of where it can be of some use is when the amount of rows returned is so big that it would be unwise to store them all in array. But in such a case you are bound to use unbuffered query, and therefore the number of rows is simply [unavailable](https://phpdelusions.net/pdo" \l "buffered).

### Disabling errors for security reasons

Either local @s or error\_reporting(0) globally. This can be seen in too much code snippets. To prevent PHP from printing errors on-screen, people often disable error reporting at all. In fact, for the newly written code error reporting should be always E\_ALL, and without @s. Whereas to prevent errors from appearing in the browser, one needs to set another configuration option:

  ini\_set('display\_errors', 0);

and let PHP to log error instead. Further reading: [Error reporting in PHP](https://phpdelusions.net/articles/error_reporting)

### If (isset($var) && !empty($var))

Is essentially a grammar tautology that reads as if (isset($var) && isset($var) && $var). The full explanation is here: [Do you really need to check for both isset() and empty() at the same time?](https://phpdelusions.net/articles/empty)

### Try and catch to echo an error

Catching an exception only to report it is a sure overkill. Simply because uncaught exception is a fatal error already, and it will be reported by itself. Without that blunt try/catch/die sequence, making your code much cleaner.

Sadly, the PHP manual is especially bad at it, showing such examples all over the place. On the one hand, it is understandable as such an example is guaranteed to output an error message, regardless of the user settings. On the other hand, this when this approach gets mindlessly copy-pasted into the live code, it turns out to be [superfluous, user-unfriendly and harmful](https://phpdelusions.net/articles/error_reporting" \l "rules) .

### HTTP\_X\_FORWARDED\_FOR, HTTP\_X\_CLIENT\_IP etc. to get the "real" IP

Just read that [old story from Anthony Ferrara](https://blog.ircmaxell.com/2012/11/anatomy-of-attack-how-i-hacked.html) and never trust anything other than REMOTE\_ADDR in security matters.

TL;DR: everything beside REMOTE\_ADDR is just an HTTP header that can be easily spoofed by anyone.

There is an exception from this rule, though: if your PHP script is behind the trusted proxy, and you positively know which particular HTTP header/env variable set by that proxy contains an external IP, you can use that variable for sure. Better yet, configure your proxy to inject the remote IP directly into REMOTE\_ADDR (i.e. fastcgi\_param REMOTE\_ADDR $http\_x\_real\_ip; for nginx).

### Single quotes are faster than double

The main article: [What's wrong with popular articles telling you that foo is faster than bar?](https://phpdelusions.net/single_vs_double)

### You name it?

The above list is highly subjective, compiled from what I see every day on Stack Overflow. If you know a fallacy or a superstition widely shared in the community, please leave a comment below.

https://phpdelusions.net/top

# PDO Examples

1. [Comments (8)](https://phpdelusions.net/pdo_examples" \l "comments)

In this chapter you will find examples for the various PDO use cases.

### Related articles:

* [Authenticating a user using PDO and password\_verify()](https://phpdelusions.net/pdo_examples/password_hash)
* [How to connect to MySQL using PDO](https://phpdelusions.net/pdo_examples/connect_to_mysql)
* [PDO Examples](https://phpdelusions.net/pdo_examples)
* [SELECT query with PDO](https://phpdelusions.net/pdo_examples/select)
* [How to create a WHERE clause for PDO dynamically](https://phpdelusions.net/pdo_examples/dynamical_where)
* [How to create a prepared statement for UPDATE query](https://phpdelusions.net/pdo_examples/dynamical_update)
* [UPDATE query using PDO](https://phpdelusions.net/pdo_examples/update)
* [PDO Examples](https://phpdelusions.net/pdo_examples)
* [INSERT query using PDO](https://phpdelusions.net/pdo_examples/insert)
* [Select the number of rows using PDO](https://phpdelusions.net/pdo_examples/count)
* [How to check if email exists in the database?](https://phpdelusions.net/pdo_examples/check_email_exists)
* [Getting a nested array when multiple rows are linked to a single entry](https://phpdelusions.net/pdo_examples/nested_array)
* [Adding a field name in the ORDER BY clause based on the user's choice](https://phpdelusions.net/pdo_examples/order_by)
* [INSERT helper function for PDO Mysql](https://phpdelusions.net/pdo_examples/insert_helper)
* [How to execute 1000s INSERT/UPDATE queries with PDO?](https://phpdelusions.net/pdo_examples/multiple)
* [DELETE query using PDO](https://phpdelusions.net/pdo_examples/delete)

# How to connect to MySQL using PDO

1. [Example](https://phpdelusions.net/pdo_examples/connect_to_mysql" \l "example)
2. [Credentials exlained](https://phpdelusions.net/pdo_examples/connect_to_mysql" \l "credentials)
3. [Connection options explained](https://phpdelusions.net/pdo_examples/connect_to_mysql" \l "options)
4. [Handling errors](https://phpdelusions.net/pdo_examples/connect_to_mysql" \l "error_handling)
5. [Creating the connection](https://phpdelusions.net/pdo_examples/connect_to_mysql" \l "connection)
6. [Don'ts](https://phpdelusions.net/pdo_examples/connect_to_mysql" \l "donts)
7. [Accessing the newly created connection](https://phpdelusions.net/pdo_examples/connect_to_mysql" \l "access)
8. [Comments (12)](https://phpdelusions.net/pdo_examples/connect_to_mysql" \l "comments)

In this example we will learn how to properly connect to Mysql database using PDO. It is based on the information provided in the [main article on PDO](https://phpdelusions.net/pdo) but with additional explanations.

Surprisingly, there is no single state-of-the-art connection example in the PHP manual. Instead, different connection options are discussed in different chapters, which makes it hard for the learner to get a single robust example that is ready to use. Below you will find such an example, as well as the explanation of all the options used.

### Example

$host = '127.0.0.1';

$db   = 'test';

$user = 'root';

$pass = '';

$port = "3306";

$charset = 'utf8mb4';

$options = [

    \PDO::ATTR\_ERRMODE            => \PDO::ERRMODE\_EXCEPTION,

    \PDO::ATTR\_DEFAULT\_FETCH\_MODE => \PDO::FETCH\_ASSOC,

    \PDO::ATTR\_EMULATE\_PREPARES   => false,

];

$dsn = "mysql:host=$host;dbname=$db;charset=$charset;port=$port";

try {

     $pdo = new \PDO($dsn, $user, $pass, $options);

} catch (\PDOException $e) {

     throw new \PDOException($e->getMessage(), (int)$e->getCode());

}

### Credentials exlained

First of all we are defining variables that contain connection credentials. This set is familiar to anyone who were using the old mysql\_connect() function, save for $charset may be, which was rarely used (although it should have been).

* $host stands for the database host. In case of the local development, it is most likely be 127.0.0.1 or localhost. In case of the live site, the actual hostname should be provided by the site admin / hosting provider. Note that connecting through IP address could save you a headache or two, so if you have a trouble with "localhost", try to use 127.0.0.1 instead.
* $db is the name of the database in MySQL (the value that you were passing into mysql\_select\_db()). On your local server it could be anything, while on a live site again it should be given to you by the admin / provider.
* $user - a database user
* $pass - a database password
* $charset is a very important option. It is telling the database in which encoding you are sending the data in and would like to get the data back. Note that due to initially limited support of unicode in the utf8 MySQL charset, it is now recommended to use utf8mb4 instead.

Note it's a good idea to store connection variables ($host, $db etc.) in a separate file. This way you'll be able to have two versions of your code, one for the local server and one for the remote.

### Connection options explained

Next we are creating an array with PDO options that are either critically important or just make your experience with PDO much better.

* PDO::ATTR\_ERRMODE - this is a cornerstone option that should be always set to PDO::ERRMODE\_EXCEPTION. It tells PDO to throw an exception every time a query failed, so you won't have to get the error manually after every query call as it was used to be with mysql\_query()
* PDO::ATTR\_EMULATE\_PREPARES - this option tell PDO whether to use an [emulation mode](https://phpdelusions.net/pdo" \l "emulation) or not. It is agreed upon that in general it's better to turn it off, however in some cases it is convenient to have it turned on. Luckily, this setting could be changed in runtime using PDO::setAttribute() method, so let's make it turned off by default as a connection option, with the possibility to fall back later.
* PDO::ATTR\_DEFAULT\_FETCH\_MODE - this option is used simply for convenience. Although the fetch method can be always set right in the fetch function call (like $row = $stmt->fetch(PDO::FETCH\_ASSOC);), it is convenient to set it once for all and then just omit it in particular fetches. Besides, when [iterating over a statement using foreach](https://phpdelusions.net/pdo" \l "foreach) there is no place where we can set the fetch mode, so again it is convenient to set it beforehand. The two most popular fetch modes are PDO::FETCH\_ASSOC and PDO::FETCH\_OBJ which make PDO to fetch the resulting row as an associative array or as an object.

### Handling errors

An uncaught exception is converted to a PHP fatal error. It is not a problem by itself, errors are for the good, and we desperately need this one to get the idea what's wrong with our database. But such a converted error contains a stack trace added to the error message, which in case of PDO connection error would include the constructor parameters which happen to be the database credentials. Again, it shouldn't be a problem, as on a live site displaying errors should be always turned off anyway, but we are humans and we make mistakes. So, to avoid even a chance to reveal the credentials, we are catching the Exception and immediately re-throwing them. So, now the stack trace begins on the throw line and doesn't contain the database credentials.

### Creating the connection

Having all the options and credentials set, we can finally proceed to creating a connection. To do so we need to create an instance of PDO class for which we need to supply 4 parameters of which the first one, called "DSN" being most important.

DSN is a semicolon-delimited string, consists of param=value pairs, that begins from the driver name and a colon:

      mysql:host=localhost;dbname=test;port=3306;charset=utf8

driver^    ^ colon         ^param=value pair    ^semicolon

Note that it's important to follow the proper format - no spaces or quotes or other decorations have to be used in DSN, but only parameters, values and delimiters, as shown in the [manual](http://php.net/manual/en/pdo.construct.php).

Beside DSN, we are using $user, $pass and $options variables defined above.

We are wrapping the creation of the PDO instance into a try..catch statement in order to be aware of the possible error, but without the risk of revealing the database credentials.

### Don'ts

Beside things that you should do, there are always things that you should do not. Some of them we will discuss below.

* PDO::ATTR\_PERSISTENT. Although this option is rather popular in the copy-pasted cargo cult PDO examples, a learner should always avoid it. There are too many drawbacks (that are outside of the scope of this article) whereas no advantages for a small site at all. This option should be used after a strong consideration only, and by no means as a blindly copy-pasted option just in case.
* using die(), echo or any other output operator for the caught exception. As it is explained in the relevant article, [PHP error reporting](https://phpdelusions.net/articles/error_reporting), an error message should never be printed unconditionally, but only according to the site-wide settings. And this should be done in the site-wide handler only.

### Accessing the newly created connection

There is one thing that makes PDO more complex than old mysql\_connect related stuff. Although one was able to use mysql\_query anywhere in the code, without taking care of the connection which was magically supplied by PHP, with PDO one should always make sure that once created PDO instance is available in each part of their script.

There are many different strategies to have this done, but mostly used are:

* global. Although this method is unanimously frowned upon, in case your code is the usual procedural spaghetti so familiar to every PHP user, using global to access a PDO instance would be your least problem. So for the simplest method possible just create a PHP file with the code above, and then include in the every PHP script that needs a database connection. Then use $pdo variable everywhere you need. To make it accessible in functions, add global $pdo; inside.
* singleton approach is better but still frowned upon. But at least it can save you a hassle with global variables. An example can be found in the corresponding chapter about a [simple PDO wrapper](https://phpdelusions.net/pdo/pdo_wrapper" \l "singleton)
* constructor parameter is the most robust method in case your code is OOP. For the every class that needs a database connection, [make PDO a constructor parameter](https://phpdelusions.net/pdo/pdo_wrapper" \l "dependency_injection)

# SELECT query with PDO

1. [SELECT query without parameters](https://phpdelusions.net/pdo_examples/select" \l "query)
   * [Getting a single row](https://phpdelusions.net/pdo_examples/select" \l "query_single)
   * [Selecting multiple rows](https://phpdelusions.net/pdo_examples/select" \l "multiple)
2. [SELECT query with parameters](https://phpdelusions.net/pdo_examples/select" \l "prepare)
   * [SELECT query with positional placeholders](https://phpdelusions.net/pdo_examples/select" \l "prepare_positional)
   * [SELECT query with named placeholders](https://phpdelusions.net/pdo_examples/select" \l "prepare_named)
   * [Selecting multiple rows](https://phpdelusions.net/pdo_examples/select" \l "prepared_multiple)
   * [Comments (12)](https://phpdelusions.net/pdo_examples/select" \l "comments)

There are several ways to run a SELECT query using PDO, that differ mainly by the presence of parameters, type of parameters, and the result type. I will show examples for the every case so you can choose one that suits you best.

Just make sure you've got a properly configured [PDO connection variable](https://phpdelusions.net/pdo_examples/connect_to_mysql) that needs in order to run SQL queries with PDO and to inform you of the possible errors.

### SELECT query without parameters

If there are no variables going to be used in the query, we can use a conventional query() method instead of prepare and execute.

// select all users

$stmt = $pdo->query("SELECT \* FROM users");

This will give us an $stmt object that can be used to fetch the actual rows.

#### Getting a single row

If a query is supposed to return just a single row, then you can just call fetch() method of the $stmt variable:

// getting the last registered user

$stmt = $pdo->query("SELECT \* FROM users ORDER BY id DESC LIMIT 1");

$user = $stmt->fetch();

Note that in PHP you can "chain" method calls, calling a method of the returned object already, like:

$user = $pdo->query("SELECT \* FROM users ORDER BY id DESC LIMIT 1")->fetch();

#### Selecting multiple rows

There are two ways to fetch multiple rows returned by a query. The most traditional way is to use the fetch() method within a while loop:

$stmt = $pdo->query("SELECT \* FROM users");

while ($row = $stmt->fetch()) {

    echo $row['name']."<br />\n";

}

This method could be recommended if rows have to be processed one by one. For example, if such processing is the only action that needs to be taken, or if the data needs to be pre-processed somehow before use.

But the most preferred way to fetch multiple rows which would to be shown on a web-page is calling the great helper method called fetchAll(). It will put all the rows returned by a query into a PHP array, that later can be used to output the data using a template (which is considered much better than echoing the data right during the fetch process). So the code would be

$data = $pdo->query("SELECT \* FROM users")->fetchAll();

// and somewhere later:

foreach ($data as $row) {

    echo $row['name']."<br />\n";

}

### SELECT query with parameters

But most of time we have to use a variable or two in the query, and in such a case we should use a prepared statement (also called a parameterized query), first preparing a query with parameters (or placeholder marks) and then executing it, sending variables separately.

In PDO we can use both positional and named placeholders. For simple queries, personally I prefer positional placeholders, I find them less verbose, but it's entirely a matter of taste.

#### SELECT query with positional placeholders

// select a particular user by id

$stmt = $pdo->prepare("SELECT \* FROM users WHERE id=?");

$stmt->execute([$id]);

$user = $stmt->fetch();

#### SELECT query with named placeholders

// select a particular user by id

$stmt = $pdo->prepare("SELECT \* FROM users WHERE id=:id");

$stmt->execute(['id' => $id]);

$user = $stmt->fetch();

#### Selecting multiple rows

Fetching multiple rows from a prepared query would be identical to that from a query without parameters already shown:

$stmt = $pdo->prepare("SELECT \* FROM users LIMIT ?, ?");

$stmt->execute([$limit, $offset]);

while ($row = $stmt->fetch()) {

    echo $row['name']."<br />\n";

}

or

$stmt = $pdo->prepare("SELECT \* FROM users LIMIT :limit, :offset");

$stmt->execute(['limit' => $limit, 'offset' => $offset]);

$data = $stmt->fetchAll();

// and somewhere later:

foreach ($data as $row) {

    echo $row['name']."<br />\n";

}

# How to create a WHERE clause for PDO dynamically

1. [Create the query dynamically](https://phpdelusions.net/pdo_examples/dynamical_where" \l "dynamical)
2. [Create a static query with all conditions at once](https://phpdelusions.net/pdo_examples/dynamical_where" \l "static)
3. [Comments (6)](https://phpdelusions.net/pdo_examples/dynamical_where" \l "comments)

This is quite a common task when we need to create a search query based on the arbitrary number of parameters.

The main problem here is that we cannot create a query beforehand (well, actually we can and it will be shown later) and therefore we don't know which parameters to bind.

### Create the query dynamically

PDO doesn't have a functionality for this out of the box, but it can ease the task significantly, thanks again to its ability to bind the arbitrary number of parameters in the form of array passed to execute(). So we can use a conditional statement which will add a condition to the query and also a parameter to bind:

// always initialize a variable before use!

$conditions = [];

$parameters = [];

// conditional statements

if (!empty($\_GET['name']))

{

    // here we are using LIKE with wildcard search

    // use it ONLY if really need it

    $conditions[] = 'name LIKE ?';

    $parameters[] = '%'.$\_GET['name']."%";

}

if (!empty($\_GET['sex']))

{

    // here we are using equality

    $conditions[] = 'sex = ?';

    $parameters[] = $\_GET['sex'];

}

if (!empty($\_GET['car']))

{

    // here we are using not equality

    $conditions[] = 'car != ?';

    $parameters[] = $\_GET['car'];

}

if (!empty($\_GET['date\_start']) && $\_GET['date\_end'])

{

    // BETWEEN

    $conditions[] = 'date BETWEEN ? AND ?';

    $parameters[] = $\_GET['date\_start'];

    $parameters[] = $\_GET['date\_end'];

}

// the main query

$sql = "SELECT \* FROM users";

// a smart code to add all conditions, if any

if ($conditions)

{

    $sql .= " WHERE ".implode(" AND ", $conditions);

}

// the usual prepare/execute/fetch routine

$stmt = $pdo->prepare($sql);

$stmt->execute($parameters);

$data = $stmt->fetchAll();

### Create a static query with all conditions at once

We can actually use a conditional statement right in the query. A condition like (name = :name or :name is null) will return true if either the value matches the field's contents or if the value is null. It means that if we want the query to bypass a condition, we just have to pass null for the value. ong story short, here it goes:

$parameters['name'] = !empty($\_GET['name']) ? "%".$\_GET['name']."%" : null;

$parameters['sex']  = !empty($\_GET['sex'])  ? $\_GET['sex']  : null;

$parameters['car']  = !empty($\_GET['car'])  ? $\_GET['car']  : null;

$sql = "SELECT \* FROM users

WHERE (name LIKE :name or :name is null)

AND   (sex  = :sex  or :sex  is null)

AND   (car  != :car  or :car  is null)";

$stmt = $pdo->prepare($sql);

$stmt->execute($parameters);

$data = $stmt->fetchAll();

The only drawback here is that this code will work only if [emulation is turned off](https://phpdelusions.net/pdo" \l "emulation). So, to make it universal, we have to make it a little bit more verbose

$param['name'] = $param['name1'] = !empty($\_GET['name']) ? "%".$\_GET['name']."%" : null;

$param['sex'] = $param['sex1']  = !empty($\_GET['sex'])  ? $\_GET['sex']  : null;

$param['car'] = $param['car1']  = !empty($\_GET['car'])  ? $\_GET['car']  : null;

$sql = "SELECT \* FROM users

WHERE (name LIKE :name or :name1 is null)

AND   (sex  = :sex  or :sex1  is null)

AND   (car  != :car  or :car1  is null)";

$stmt = $pdo->prepare($sql);

$stmt->execute($param);

$data = $stmt->fetchAll();

# How to create a prepared statement for UPDATE query

1. [Comments (4)](https://phpdelusions.net/pdo_examples/dynamical_update" \l "comments)

Sometimes we need to update not all fields but only certain ones. Say, we have an array with values we want to add to the query.

The task is not very complex, but one have to keep in mind the [SQL injection against which prepared statements won't help](https://phpdelusions.net/pdo/sql_injection_example). To protect from that, all field names should be checked against a pre-compiled list of allowed values. The rest is simple. So the algorithm would be:

1. Define an array with allowed values
2. Loop over the source array and create a SET statement for SQL dynamically, based on the allowed fields list
3. Respective values should be added to the designated array to be used in the execute()

So, imagine we have $\_POST array with several fields. Here is the code to insert them into database safely:

// the list of allowed field names

$allowed = ["name","surname","email"];

// initialize an array with values:

$params = [];

// initialize a string with `fieldname` = :placeholder pairs

$setStr = "";

// loop over source data array

foreach ($allowed as $key)

{

    if (isset($\_POST[$key]) && $key != "id")

    {

        $setStr .= "`$key` = :$key,";

        $params[$key] = $\_POST[$key];

    }

}

$setStr = rtrim($setStr, ",");

$params['id'] = $\_POST['id'];

$pdo->prepare("UPDATE users SET $setStr WHERE id = :id")->execute($params);

}

# UPDATE query using PDO

1. [UPDATE query with positional placeholders](https://phpdelusions.net/pdo_examples/update" \l "positional)
2. [UPDATE query with named placeholders](https://phpdelusions.net/pdo_examples/update" \l "named)
3. [Comments (14)](https://phpdelusions.net/pdo_examples/update" \l "comments)

First of all, make sure you've got a properly configured [PDO connection variable](https://phpdelusions.net/pdo_examples/connect_to_mysql) that is needed in order to run SQL queries using PDO (and to inform you of the possible errors).

In order to run an UPDATE query with PDO just follow the steps below:

* create a correct SQL UPDATE statement
* replace all actual values with placeholders
* prepare the resulting query
* execute the statement, sending all the actual values to execute() in the form of array.

### UPDATE query with positional placeholders

As usual, positional placeholders are more concise and easier to use

$sql = "UPDATE users SET name=?, surname=?, sex=? WHERE id=?";

$stmt= $pdo->prepare($sql);

$stmt->execute([$name, $surname, $sex, $id]);

or you can chain execute() to prepare():

$sql = "UPDATE users SET name=?, surname=?, sex=? WHERE id=?";

$pdo->prepare($sql)->execute([$name, $surname, $sex, $id]);

### UPDATE query with named placeholders

In case you have a predefined array with values, or prefer named placeholders in general, the code would be

$data = [

    'name' => $name,

    'surname' => $surname,

    'sex' => $sex,

    'id' => $id,

];

$sql = "UPDATE users SET name=:name, surname=:surname, sex=:sex WHERE id=:id";

$stmt= $pdo->prepare($sql);

$stmt->execute($data);

or you can chain execute() to prepare():

$sql = "UPDATE users SET name=:name, surname=:surname, sex=:sex WHERE id=:id";

$pdo->prepare($sql)->execute($data);

Remember that you should't wrap every query into a try..catch statement. Instead, let a possible error to bubble up to either the built-in PHP's or your custom error handler.

# INSERT query using PDO

1. [INSERT query with positional placeholders](https://phpdelusions.net/pdo_examples/insert" \l "positional)
2. [INSERT query with named placeholders](https://phpdelusions.net/pdo_examples/insert" \l "named)
3. [INSERTing multiple rows](https://phpdelusions.net/pdo_examples/insert" \l "multiple)
4. [Comments (18)](https://phpdelusions.net/pdo_examples/insert" \l "comments)

First of all make sure you've got a properly configured [PDO connection variable](https://phpdelusions.net/pdo_examples/connect_to_mysql) that needs in order to run SQL queries with PDO and to inform you of the possible errors.

In order to run an INSERT query with PDO just follow the steps below:

* create a correct SQL INSERT statement
* replace all actual values with placeholders
* prepare the resulting query
* execute the statement, sending all the actual values in the form of array.

### INSERT query with positional placeholders

As usual, positional placeholders are more concise and easier to use

$sql = "INSERT INTO users (name, surname, sex) VALUES (?,?,?)";

$stmt= $pdo->prepare($sql);

$stmt->execute([$name, $surname, $sex]);

or you can chain execute() to prepare():

$sql = "INSERT INTO users (name, surname, sex) VALUES (?,?,?)";

$pdo->prepare($sql)->execute([$name, $surname, $sex]);

Important! You don't have to check the result of execute() (as it is often shown in low-quality tutorials). Such a condition will make no sense, as in case of error, a PDOException will be thrown and the script execution will be terminated, which means such a condition will never reach the else part.  
Neither a try ... catch operator should be used, unless you have a specific scenario to handle the error, such as a transaction rollback shown below. Please see the article about [error reporting](https://phpdelusions.net/articles/error_reporting) for the details.

### INSERT query with named placeholders

In case you have a predefined array with values, or prefer named placeholders in general, the code would be

$data = [

    'name' => $name,

    'surname' => $surname,

    'sex' => $sex,

];

$sql = "INSERT INTO users (name, surname, sex) VALUES (:name, :surname, :sex)";

$stmt= $pdo->prepare($sql);

$stmt->execute($data);

or you can chain execute() to prepare():

$sql = "INSERT INTO users (name, surname, sex) VALUES (:name, :surname, :sex)";

$pdo->prepare($sql)->execute($data);

Important! The only characters allowed in the placeholder names are Latin letters, numbers and underscores. No umlauts or dashes ar any other characters are allowed. It's also a good idea to keep all the letters lover case.

### INSERTing multiple rows

As it's explained in the [main article](https://phpdelusions.net/pdo" \l "multiexec), a once prepared statement could be executed multiple times, slightly reducing the overhead on the query parsing. So it makes sense to use this feature when we need to insert multiple rows into the same table. Just a couple notes before we begin:

* make sure that the [emulation mode](https://phpdelusions.net/pdo" \l "emulation) is turned off, as there will be no speed benefit otherwise, however small it is.
* it's a good idea to wrap our queries in a [transaction](https://phpdelusions.net/pdo" \l "transactions). In some circumstances it will greatly speed up the inserts, and it makes sense overall, to make sure that either all data has been added or none.

So in the end our code would be like

$data = [

    ['John','Doe', 22],

    ['Jane','Roe', 19],

];

$stmt = $pdo->prepare("INSERT INTO users (name, surname, age) VALUES (?,?,?)");

try {

    $pdo->beginTransaction();

    foreach ($data as $row)

    {

        $stmt->execute($row);

    }

    $pdo->commit();

}catch (Exception $e){

    $pdo->rollback();

    throw $e;

}

# Select the number of rows using PDO

1. [Comments (1)](https://phpdelusions.net/pdo_examples/count" \l "comments)

When you need to count rows that match some criteria in your database, under no circumstances you should select the actual rows and then use rowCount()!

Instead, you should always ask your database to count the rows and then return the only number, with a query like this:

SELECT count(1) FROM users

// or

SELECT count(\*) FROM goods WHERE category\_id = 1

(note that it doesn't matter which constant value to use as a count() function's parameter, be it \*, 0 or 1 or anything - all works the same as long as it's a constant as opposed to the field name. In case of the latter, only values of this field that are not null will be counted).

Luckily, PDO has a dedicated function to get that single number right off the query, [fetchColumn()](https://phpdelusions.net/pdo" \l "fetchcolumn).

If no variable is going to be used in the query, we could neatly chain this function right to the query() call:

$count = $pdo->query("SELECT count(\*) FROM table")->fetchColumn();

But if any variable is going to be used in the query, it should be always substituted with a parameter, and executed using a prepared statement:

$stmt = $pdo->prepare("SELECT count(\*) FROM goods WHERE category\_id = ?");

$stmt->execute([$category\_id]);

$count = $stmt->fetchColumn();

# How to check if email exists in the database?

1. [A common mistake](https://phpdelusions.net/pdo_examples/check_email_exists" \l "mistake)
2. [How to check whether a value exists?](https://phpdelusions.net/pdo_examples/check_email_exists" \l "check)
3. [Notes](https://phpdelusions.net/pdo_examples/check_email_exists" \l "notes)
4. [Comments (7)](https://phpdelusions.net/pdo_examples/check_email_exists" \l "comments)

### A common mistake

A very common newbie mistake is to check the result of the query, like

$check\_email = $pdo->query("SELECT \* FROM users WHERE email='test@example.com'");

if ($check\_email) {

    // found

}

Alas, it's entirely wrong. The result of the query has nothing to do with the number of rows found. An empty result is still a legitimate result, it cannot be false only because the query didn't find anything.

In fact, the code to check whether some value exists in the database would be the just a regular routine for select queries.

And of course, a prepared statement is obligatory. NEver ever put a data variable directly tho the query. Always use a placeholder instead!

### How to check whether a value exists?

To check whether a particular value exists in the database, you simply have to run just a regular SELECT query, fetch a row and see whether anything has been fetched.

Given $pdo contains a [valid PDO instance](https://phpdelusions.net/pdo_examples/connect_to_mysql), the code would be

$email = "test@example.com";

$stmt = $pdo->prepare("SELECT \* FROM users WHERE email=?");

$stmt->execute([$email]);

$user = $stmt->fetch();

if ($user) {

    // email found

} else {

    // or not

}

Here we are selecting a row matching our criteria, then fetching it and then checking whether anything has been selected or not.

### Notes

It also makes sense to select just a single constant value to reduce overhead a little, if we aren't going to use any selected data. For that purpose simple replace \* with 1. Having a function or a class' method for the purpose also would be handy:

function emailExists($pdo, $email) {

    $stmt = $pdo->prepare("SELECT 1 FROM users WHERE email=?");

    $stmt->execute([$email]);

    return $stmt->fetchColumn();

)

if (emailExists($pdo, $email)) {

    // found

}

here we are using a handy PDO's method fetchColumn that returns a selected value or false otherwise. Given 1 will be treated as true by if statement, we can return this method's result directly. However, in case you prefer strict typing, the return value could be cast to boolean:

    return (bool)$stmt->fetchColumn();

Also, as can be seen from above, a popular solution with getting the number of rows returned is just unnecessary. Calling $stmt->numRows() is just superfluous, not to mention it can be just unavailable due to some drivers or settings.

# Getting a nested array when multiple rows are linked to a single entry

1. [Getting a nested array from a single query with join](https://phpdelusions.net/pdo_examples/nested_array" \l "join)
2. [Comments](https://phpdelusions.net/pdo_examples/nested_array" \l "comments)

There is a situation when we have to select some data from the table and also get multiple rows from another table linked to an entry from the main table. A frequent example is an article and comments. Or a category and products from that category.

### Getting a nested array from a single query with join

The most direct solution for this task is to get all the data using a single SQL query with JOIN, such as,

SELECT c.id, c.name, c.url, p.id, p.name, p.price

FROM category c

JOIN product p ON p.category\_id=c.id

There is a drawback of course, as it will return a lot of duplicated data. If the data from the main table is large, like a news article, it would be a waste to use this approach. But in case the data from the main table is relatively small, it is not a problem, so woe could use a query like this:

In a rare case when we only need a single field from the main table, PDO already has a [solution](https://phpdelusions.net/pdo" \l "group). But in case you need more than one field from each table, follow this article further on.

To get the data in a nested array we will need to create an empty array first and then fetch the returned rows one by one and check if we already have an array element for the main entry

* if not, we should create the element, along with a nested array for the linked rows
* if yes - just add another entry in the linked rows section

In order to make it work we will need to index the resulting array with some unique value. Let it be the category id.

So here is the code that does exactly this:

$sql = "SELECT c.id as c\_id, c.name as c\_name, c.url,

               p.id as p\_id, p.name as p\_name, p.price

        FROM category c

        JOIN product p ON p.category\_id=c.id

        ORDER BY c.name, p.name

        LIMIT ?,?";

$stmt = $pdo->prepare($sql);

$stmt->execute([$start, $limit]);

$data = [];

while ($row = $stmt->fetch()) {

    $product = [

        'id' => $row['p\_id'],

        'name' => $row['p\_name'],

        'price' => $row['price'],

    ];

    if (!isset($data[$row['c\_id']])) {

        $row['c\_id'] = [

            'name' => $row['c\_name'],

            'url' => $row['url'],

            'products' => [$product]

        ];

    } else {

        $row['c\_id']['products'][] = $product;[

    }

}

# Adding a field name in the ORDER BY clause based on the user's choice

1. [Comments](https://phpdelusions.net/pdo_examples/order_by" \l "comments)

Unfortunately, PDO has no placeholder for identifiers (table and field names), therefore a developer must filter them out manually. Such a filter is often called a "white list" (where we only list allowed values) as opposed to a "black list" where we list disallowed values.

So we have to explicitly list all possible variants in the PHP code and then choose from them.

Here is an example with the "soft" approach - in case a field name is not found in the white list, the first one will be chosen

$orderby = $\_GET['orderby'] ?: "name"; // set the default value

$allowed = ["name","price","qty"]; // the white list of allowed field names

$key     = array\_search($orderby, $allowed); // see if we have such a name

$orderby = $allowed[$key]; //if not, the first one will be set automatically. smart enough :)

But it is generally considered a bad practice to let an invalid request, so it's better tho throw a an error. For this, just check if array\_search returned false and throw the appropriate custom exception/error:

$key     = array\_search($orderby, $allowed); // see if we have such a name

if ($key === false) {

    // given there is such a custom exception class

    throw new http400Error("Invalid order by value");

}

Exactly the same approach should be used for the direction. It could be either strict or soft. For the soft one the code could be a bit simpler due to the fact that we have to choose from two values only:

$direction = \_GET['direction'] == 'DESC' ? 'DESC' : 'ASC';

However, nowadays PHP is becoming more and more stricter language, so it's better to follow this way as well. Means we shouldn't silently use the default value if invalid value is provided.

Given the code for the strict checking would almost identical, it's a good reason to create a function:

function white\_list(&$value, $allowed, $message) {

    if ($value === null) {

        return $allowed[0];

    }

    $key = array\_search($value, $allowed, true);

    if ($key === false) {

        throw new InvalidArgumentException($message);

    } else {

        return $value;

    }

}

The function accepts three parameters:

* the checked value. It is passed by reference so it won't raise an error in case a variable is not set. It would allow us to assign a default value if no value is provided.
* the list of allowed values. the first one would serve as a default value
* the error message to throw so a programmer would know what caused the error.

Having such a function at hand we can have our code much more tidy and concise:

$orderby = white\_list($\_GET['orderby'], ["name","price","qty"], "Invalid field name");

$direction = white\_list($\_GET['direction'], ["ASC","DESC"], "Invalid ORDER BY direction");

$query  = "SELECT \* FROM `table` ORDER BY `$orderby` $direction"; // sound and safe!

# INSERT helper function for PDO Mysql

* + [First of all - escaping identifiers](https://phpdelusions.net/pdo_examples/insert_helper" \l "identifiers)
  + [PDO INSERT helper function](https://phpdelusions.net/pdo_examples/insert_helper" \l "helper)
  + [Notes](https://phpdelusions.net/pdo_examples/insert_helper" \l "notes)
  + [Comments](https://phpdelusions.net/pdo_examples/insert_helper" \l "comments)

It is often happens that we have an array consists of fields and their values that represents a row to be inserted into a database. And naturally it would be a good idea to have a function to convert such an array into a correct SQL INSERT statement and execute it. So here it goes.

#### First of all - escaping identifiers

When creating such a function, one must be acutely aware of the [**SQL Injection** through field names](https://phpdelusions.net/pdo/sql_injection_example). Because we are going to add not only values but also identifiers (i.e. table and field names), which cannot be substituted with placeholders, we will need another technique to protect them.

YES, we need to protect them. Your intended usage scenario, when all identifiers are intended to be hardcoded, is not an excuse. An application is never a static piece of code. It evolves. Other people join to the project, knowing absolutely nothing of your notions and intents. They see a function, consider it safe and use it. There even is [a real lot of people](https://stackoverflow.com/q/22534183/285587) thinking that certain form elements are "not coming from the client side". This is why the entire function must be safe, not some unspoken ideas on how it should be used.

There are basically two ways to protect identifiers

1. The preferred one is called whitelisting. It is a bullet-proof technique that protects both from injections and also from accessing the columns that are not intended to be updated by the user. But this method is a bit tedious and more suitable for the OOP approach, where we can list the table and the column names a the class properties and then use it in the insert method. After all, the perfect white-listing is implemented when we are writing a conventional SQL INSERT query without any helper functions, and I encourage you to strongly consider this approach.
2. Escaping. Yes, we can escape identifiers too, pretty much the same way as we used to escape strings. This method has its drawbacks, as it doesn't filter out columns that may be forbidden for the user to change. And also this method is not universal, as it must be different for different SQL flavors. Despite that, I would use this method, as I think it is more suitable for a simple helper function.

Hence, this function will need a helper function of its own (limited to MySQL):

function escape\_mysql\_identifier($field){

    return "`".str\_replace("`", "``", $field)."`";

}

#### PDO INSERT helper function

And now we can finally have a function that accepts a table name and an array with data and runs a prepared INSERT query against a database:

function prepared\_insert($pdo, $table, $data) {

    $keys = array\_keys($data);

    $keys = array\_map('escape\_mysql\_identifier', $keys);

    $fields = implode(",", $keys);

    $table = escape\_mysql\_identifier($table);

    $placeholders = str\_repeat('?,', count($keys) - 1) . '?';

    $sql = "INSERT INTO $table ($fields) VALUES ($placeholders)";

    $pdo->prepare($sql)->execute(array\_values($data));

}

Then it can be used like this

prepared\_insert($pdo, 'users', ['name' => $name, 'password' => $hashed\_password]);

What are we doing here?

* In the first three lines we are creating a correct list of correctly formatted column names, using keys from the $data array, to get a string like this:

`name`,`password`

* Then we are formatting the table name
* then we are creating a list of placeholders, like

(?,?)

* then we are creating the final SQL,

"INSERT INTO `users` (`name`,`password`) VALUES (?,?)";

* and finally preparing and executing it, using values from the $data array.

Note that I am using positional placeholders, because column names can potentially contain characters that are disallowed in the placeholder names, such as spaces or dashes. Besides, for a helper function it doesn't really matter which kind of placeholders we are using.

In the end, it will make a bullet-proof insert query protected from SQL injections.

#### Notes

Please note that this function prevents you from using SQL functions such as NOW(), DEFAULT(), inet\_aton() and such. In case you will need one, just fall back to raw SQL for this particular query. Or just get such values in PHP, such as date('Y-m-d H:i:s') instead of NOW(), etc. I strongly advise you against going down the rabbit hole of making such functions available - it will make your code too complex, clumsy an error-prone.

# How to execute 1000s INSERT/UPDATE queries with PDO?

1. [Comments](https://phpdelusions.net/pdo_examples/multiple" \l "comments)

The key to have your multiple inserts fast and reliable is to use transactions and a prepared statement.

A transaction will ensure that all operations either complete in their entirety or have no effect whatsoever. Another important advantage of using a transaction is that it makes multiple inserts dramatically faster, eliminating all possible delays that could be caused by separate query execution.

A prepared statement will add some little boost to the speed and of course ensure that there will be no errors or injections.

First of all, make sure you have a [proper mysqli connection](https://phpdelusions.net/mysqli/mysqli_connect), which, among other things, tells mysqli to throw an exception in case of error. Then just prepare your query, start a transaction, execute the queries and commit the transaction

include 'pdo.php';

// a sample array of data

$data = [

    ['col1' => 'foo1', 'col2' => 'bar1'],

    ['col1' => 'foo2', 'col2' => 'bar2'],

];

// prepare the SQL query once

$stmt = $pdo->prepare("INSERT INTO table SET col1 = ?, col2 = ?");

$pdo->beginTransaction();

// loop over the data array

foreach ($data as $row) {

    $stmt->execute([$row['col1'], $row['col2']]);

}

$pdo->commit();

note that it could be not just a single query but multiple different queries but the principle remains the same: just wrap all queries in a transaction and use prepared statements.

# DELETE query using PDO

1. [DELETE query with positional placeholders](https://phpdelusions.net/pdo_examples/delete" \l "positional)
2. [DELETE query with named placeholders](https://phpdelusions.net/pdo_examples/delete" \l "named)
3. [Comments](https://phpdelusions.net/pdo_examples/delete" \l "comments)

First of all, make sure you've got a properly configured [PDO connection variable](https://phpdelusions.net/pdo_examples/connect_to_mysql) that is needed in order to run SQL queries using PDO (and to inform you of the possible errors).

In order to run an DELETE query with PDO just follow the steps below:

* create a correct SQL DELETE statement
* replace all actual values with placeholders
* prepare the resulting query
* execute the statement, sending all the actual values to execute() in the form of array.

### DELETE query with positional placeholders

As usual, positional placeholders are more concise and easier to use

$sql = "DELETE FROM users WHERE id=?";

$stmt= $pdo->prepare($sql);

$stmt->execute([$id]);

or you can chain execute() to prepare():

$pdo->prepare("DELETE FROM users WHERE id=?")->execute([$id]);

### DELETE query with named placeholders

In case you have a predefined array with values, or prefer named placeholders in general, the code would be

$data = [

    'date' => $date,

    'used' => $used,

];

$sql = "DELETE FROM coupons WHERE used = :used AND date\_active < :date";

$stmt= $pdo->prepare($sql);

$stmt->execute($data);

or you can chain execute() to prepare():

$sql = "DELETE FROM coupons WHERE used = :used AND date\_active < :date";

$pdo->prepare($sql)->execute($data);

Remember that you should't wrap every query into a try..catch statement. Instead, let a possible error to bubble up to either the built-in PHP's or your [custom error handler](https://phpdelusions.net/articles/error_reporting).