PHP PDO

Summary

	PDO	*MySQLi*
Database support	12 different drivers	MySQL only
API	OOP	OOP + procedural
Connection	Easy	Easy
Named parameters	Yes	No
Object mapping	Yes	Yes
Prepared statements (client side)	Yes	No
Performance	Fast	Fast
Stored procedures	Yes	Yes

Connection

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

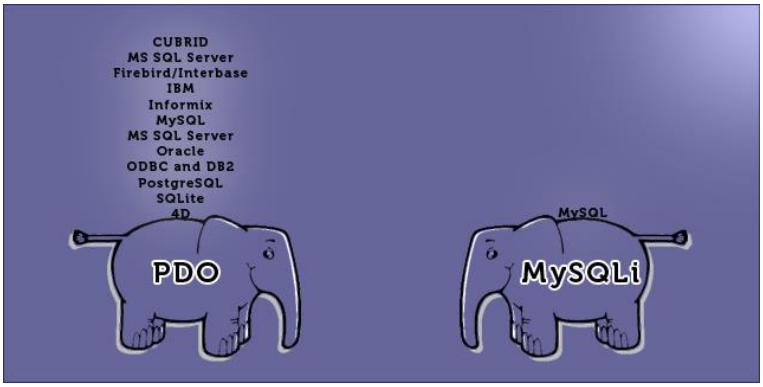
```
1  // PDO
2  $pdo = new PDO("mysql:host=localhost;dbname=database", 'username', 'password');
3
4  // mysqli, procedural way
5  $mysqli = mysqli_connect('localhost','username','password','database');
6
7  // mysqli, object oriented way
8  $mysqli = new mysqli('localhost','username','password','database');
```

Please note that these connection objects / resources will be considered to exist through the rest of this tutorial.

API Support

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

Database Support



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

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

```
var_dump(PDO::getAvailableDrivers());
```

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

Named Parameters

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

```
$params = array(':username' => 'test', ':email' => $mail, ':last_login' => time() - 3600);

$pdo->prepare('
    SELECT * FROM users
    WHERE username = :username
    AND email = :email
    AND last_login > :last_login');

$pdo->execute($params);
```

...opposed to the MySQLi way:

```
$query = $mysqli->prepare('
    SELECT * FROM users
    WHERE username = ?
    AND email = ?
    AND last_login > ?');

$query->bind_param('sss', 'test', $mail, time() - 3600);
$query->execute();

```

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

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

Object Mapping

Both PDO and MySQLi can map results to objects. This comes in handy if you don't want to use a custom database abstraction layer, but still want ORM-like behavior. Let's imagine that we have a `User` class with some properties, which match field names from a database.

```
01 class User {
02     public $id;
03     public $first_name;
04     public $last_name;
05 
06     public function info()
07     {
08         return '#'.$this->id.': '.$this->first_name.' '.$this->last_name;
09     }
10 }
```

Without object mapping, we would need to fill each field's value (either manually or through the constructor) before we can use the `info()` method correctly.

This allows us to predefine these properties before the object is even constructed! For instance:

```
01 $query = "SELECT id, first_name, last_name FROM users";
02 
03 // PDO
04 $result = $pdo->query($query);
05 $result->setFetchMode(PDO::FETCH_CLASS, 'User');
06 
07 while ($user = $result->fetch()) {
08     echo $user->info()."\n";
09 }
10 // MySQLI, procedural way
11 if ($result = mysqli_query($mysqli, $query)) {
12     while ($user = mysqli_fetch_object($result, 'User')) {
13         echo $user->info()."\n";
14     }
15 }
16 // MySQLi, object oriented way
17 if ($result = $mysqli->query($query)) {
18     while ($user = $result->fetch_object('User')) {
19         echo $user->info()."\n";
20     }
21 }
```

Security

```
SELECT * FROM
users
WHERE
username = 'Administrator'
AND
password = 'x' OR 'x' = 'x';
```

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

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

```
$_GET['username'] = "'; DELETE FROM users; /*"
```

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

```
// PDO, "manual" escaping
$username = PDO::quote($_GET['username']);

$pdo->query("SELECT * FROM users WHERE username = $username");

// mysqli, "manual" escaping
$username = mysqli_real_escape_string($_GET['username']);

$mysqli->query("SELECT * FROM users WHERE username = '$username'");

```

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

```
// PDO, prepared statement
$pdo->prepare('SELECT * FROM users WHERE username = :username');
$pdo->execute(array(':username' => $_GET['username']));

// mysqli, prepared statements
$query = $mysqli->prepare('SELECT * FROM users WHERE username = ?');
$query->bind_param('s', $_GET['username']);
$query->execute();

```

I recommend that you always use prepared statements with bound queries instead of PDO`::quote()` and `mysqli_real_escape_string()`.

Performance

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

Summary

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

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