Intermediate PHP

Matt Warner

1 Connecting to a Database

PHP can act as a **client** and can make a connection to a database **server**. Once we make the connection we can send queries to it.

To connect to a server, a client needs to know where it is. We provide this information in the form of a ***dns*** (data source name).

The ***dsn*** holds our database type, hostname, and database name. The ***hostname*** contains the location of the database. If the DBMS is running on the same host (computer) as PHP, we use `localhost`. If it's running on a remote server, we use its *ip address* or *domain name*.

Note:-

Any examples provided in here will be using PostgreSQL as the database type. For the hostname, we will be using `localhost`.

Setting up the connection to the database is mostly just a bunch of boilerplate. To create the DSN, we first provide the **prefix**, where we specify which database system to use. For PostgreSQL, that is `pgsql`. For MariaDB, it would be mysql. Then we add a colon, followed by a key/value pair containing the hostname and database name, sperated by a semi-colon. The DSN uses a precise format, so it's important not to include spaces or extra characters.

```
1  $dsn = "pgsql:host=$hostname;dbname=$dbname";
```

We'll also need to store the `username` and `password of an account with the correct permissions for the database`.

```
1  $username = "matt";
2  $password = "pass";
```

We'll start the connection by creating a database object from the PDO class and instantiating it using keyword `new`. The PDO constructor takes the `$dsn`, `$username` and `$password` variables as its args in that order. If the connection is successful, the database object will be assigned to `$db`.

```
1  $db = new PDO($dsn, $username, $password);
```

To end the connection to the database, we set the `db` object to null.

```
1  $db = null;
```

Putting it all together, we have:

```
$hostname = "localhost";
$dbname = "testdb";
$username = "matt";
$password = "pass";

$dsn = "pgsql:host=$hostname;dbname=$dbname";

$db = new PDO($dsn, $username, $password);
```

2 Creating Queries

Now that we've created a database object, `$db`, we can call its methods to fetch data from the database.

Lets take a look at an example database. It contains a `books` table with a list of books with their `id`, `title`, `author`, and published `year`.

id	Title	Author	Year
1	Don Quixote	Miguel de Cervantes	1605
2	Robinson Crusoe	Daniel Defoe	1719
3	Pride and Prejudice	Jane Austen	1813
4	Emma	Jane Austen	1816
5	A Tale of Two Cities	Charles Dickens	1859

Table 1: A table of classic literature.

We'll begin with a query to fetch book titles. We'll use the `query()` method on the `$db` object and execute the query. We can do this by referencing the `$db` object with the array object operator (–>) before calling the `query()` method.

```
1 $bookquery = $db->query('SELECT title FROM books');
```

The sytax in parenthesis is just a regular SQL query that is understood by the database.

Let's fetch the result of the query and assign it to the `$book variable`. We do this by calling the `fetch()` method on `$bookquery`.

```
1 $book = $bookquery->fetch();
```

Although our `SELECT` statement above queries the database for *all* book titles, the `fetch()` method returns only *one* result. To return a list of *all* book titles, we can use the `fetchAll()` method instead:

```
$books = $bookQuery->fetchAll();
```

If we were to print all the element in the `$books` array, every element would appear twice. This is how the fetch `fetch()` method behaves by default. It stores both **numeric**, and **associative** indicies, leading to every element in the row appearing twice in the array.

When using `fetch()`, we can supply whats called a ***fetch mode***. We can supply either `PDO::FETCH_NUM` or `PDO_FETCH_ASSOC` as arguments. As their names suggest, PDO::FETCH_NUM will supply numeric indices, and PDO::FETCH_ASSOC will supply associative indices.

```
// No args.
$book = $book_query->fetch();
print_r("Fetch with no args<br><br>");
var_dump($book);

// Numeric.
print_r("<br><br>Fetch with PDO::FETCH_NUM<br><br>");
$book = $book_query->fetch(PDO::FETCH_NUM);
var_dump($book);

// Associative.
print_r("<br><br>Fetch with PDO::FETCH_ASSOC<br><br>");
$book = $book_query->fetch(PDO::FETCH_ASSOC);
var_dump($book);
```

Output:

```
Fetch with no args

array(16) { ["BookID"]=> int(1) [0]=> int(1) ["Title"]=> string(4) "1984" [1]=> string(4)
↪  "1984" ["AuthorID"]=> int(1) [2]=> int(1) ["PublisherID"]=> int(1) [3]=> int(1)
↪  ["GenreID"]=> int(2) [4]=> int(2) ["PublishDate"]=> string(10) "1949-06-08" [5]=>
↪  string(10) "1949-06-08" ["Pages"]=> int(328) [6]=> int(328) ["Price"]=> string(5)
↪  "15.99" [7]=> string(5) "15.99" }

Fetch with PDO::FETCH_NUM

array(8) { [0]=> int(2) [1]=> string(21) "To Kill a Mockingbird" [2]=> int(2) [3]=>
↪  int(2) [4]=> int(1) [5]=> string(10) "1960-07-11" [6]=> int(281) [7]=> string(5)
↪  "14.99" }

Fetch with PDO::FETCH_ASSOC

array(8) { ["BookID"]=> int(3) ["Title"]=> string(39) "Harry Potter and the Philosophers
↪  Stone" ["AuthorID"]=> int(3) ["PublisherID"]=> int(1) ["GenreID"]=> int(3)
↪  ["PublishDate"]=> string(10) "1997-06-26" ["Pages"]=> int(223) ["Price"]=> string(5)
↪  "12.99" }
```

`fetch()` will only ever return one row, if we wanted a list of *all* rows, we can use the `fetchAll()` method instead.

```
$books = $book_query->fetchAll(PDO::FETCH_ASSOC);
```

The line above will create `$books` as a two dimensional array, where each inner array will cotains the items from each row of the query. If we wanted to print out all the contents of the first row, we could write:

```
foreach($books[0] as $i) {
    echo $i . " ";
}

```

3 SQL Injection

Say we wanted to write a query that lets a user get a book's details by providing its ID.

```
// Get the ID from the frontend
$id = $_POST['id'];

// Like this?
$books_query = $db->query("SELECT * FROM books WHERE id = $id");
```

But what if instead of entering a number, a malicious user enters `1 or 1 = 1`?

Then the database will run the query:

```
SELECT * FROM books WHERE id = 1 or 1 = 1;
```

Since $1 = 1$ is always true, the database will return *every* row from the books table. While returning all books might not be a problem, an attacker can use the same technique to return a list of users, passwords, and other confidential information.

We can prevent SQL injection by telling the database which values should be treated *only* as data. We do this with the ***prepare*** statement.

A ***prepare*** statement is a pre-defined template containing SQL and optionally **placeholders**. We use placeholders to tell the database where to place the data we will provide when executing the statement.

```
$id = _POST['id'];
$books_query = $db->prepare('SELECT * FROM books WHERE id = :id');
```

The next step is to run the `execute()` method, and pass in an array with a key-value pair which maps our placeholders to variables.

```
$book_query->execute(['id'=>$id]);
```

Now, we can fetch the result:

```
$book = book_query->fetch(PDO::FETCH_ASSOC);
```

With this approach, the SQL and the data are sent to the database **seperatly**. The database first parses and compiles the SQL query into an execution plan. The database then binds the user-provided value `$id` as a parameter to the already compiled query. At this stage, the database treats `$id` as data, not as SQL code. Thus, the select statement might look something like this instead:

```
// Safe. Would not delete the table.
SELECT * FROM books WHERE id = '1; DROP TABLE books;'
```

As an added layer of protection, we can sanitize `$id` by using `filter_input()` to ensure we're only passing in numbers.

```
$id = filter_input(INPUT_POST, 'id', FILTER_SANITIZE_NUMBER_INT);
```