# Introduction to working with PDO

This workbook provides a brief introduction to PHP Data Objects, PDO and how to work with PDO in JavaScript/jQuery. It includes explanations, code descriptions and code examples, which you should read. It also includes activity sections which ask you to carry out a set of actions, for example to create or modify some code. The activities reinforce what you have read, but also give you vital coding practice. The workbook is design to be read in sequence, if you skip parts it may make it harder to understand later parts.

This Workbook assumes that you are familiar with JavaScript and jQuery and have some knowledge of databases, SQL and php.

Whilst the workbook covers the essentials, you may find it useful to refer to other sources of information. The w3Schools website has some useful resources.

## PDO

PDO, PHP Data Objects, provides a lightweight interface for accessing databases in php. It provides a unified interface that can be used to access many different types of database. It provides benefits in terms of security (through prepared statements) usability (through helper functions) and reusability across different database types.

One of the main reasons for using PDO are prepared statements, which allow us to prepare SQL statements which include placeholders for variables and then pass in the variables separately. This provides separation between code and data. The data (well, actually only string and number literals) in prepared statements is also formatted correctly, automatically, helping to protect against malicious SQL injection and improperly formatted SQL more generally.

## Database Connection

You will have connected to a database from a webpage in previous modules, so will already know your connection credentials. If you do not remember them, they are probably stored somewhere in your webspace – look for a file called dbinfo.info.php or something similar. The contents will look something like:

<?PHP

$host="localhost";

$username="user\_12345678";

$password="psswrd";

$database="db\_12345678";

?>

In PDO we connect to a database using a method called DSN (Database Source Name). The DSN is a string of parameters, values and delimiters which specify they type and location of the database we wish to connect to. It contains no spaces or quote marks. For example:

"mysql:host=$host;dbname=$database;charset=utf8"

The username and password are provided separately, for example:

"mysql:host=$host;dbname=$database;charset=utf8", $username, $password

And additional options can follow, for example:

"mysql:host=$host;dbname=$database;charset=utf8", $username, $password, [PDO:: ATTR\_ERRMODE => PDO::ERRMODE\_EXCEPTION, PDO::ATTR\_EMULATE\_PREPARES => false]

**Activity**

1. Track down your connection credentials.
2. Connect to phpMyAdmin - https://at-web2.comp.glam.ac.uk/phpmyadmin/index.php
3. Create at least two tables that follow the specification in the coursework.

## Retrieving Table Names

The first task we will look at is retrieving the table names from the database. Consider the php below:

<?php

if(!isset($error)){

$error = new stdClass();

}

include "dbinfo.info.php";

try {

$pdo = new PDO("mysql:host=$host;dbname=$database;charset=utf8", $username, $password, [PDO:: ATTR\_ERRMODE => PDO::ERRMODE\_EXCEPTION, PDO::ATTR\_EMULATE\_PREPARES => false]);

} catch (PDOexception $e) {

$error->code = "error";

$error->message = $e->getMessage();

echo json\_encode($error);

$pdo = null;

return;

}

$tables = array();

$result = $pdo->query("SHOW TABLES");

while ($row = $result->fetch(PDO::FETCH\_NUM)){

$tables[] = $row[0];

}

echo json\_encode($tables);

$pdo = null;

?>

We are including the file that holds the details of the database and our connection credentials:

include "dbinfo.info.php";

We are connecting to the database using the information from the connection credentials file:

$pdo = new PDO("mysql:host=$host;dbname=$database;charset=utf8", $username, $password, [PDO:: ATTR\_ERRMODE => PDO::ERRMODE\_EXCEPTION, PDO::ATTR\_EMULATE\_PREPARES => false]);

If the connection fails, we are returning an error message. Otherwise, we are executing an SQL query SHOW TABLES on the database:

$result = $pdo->query("SHOW TABLES");

Note that we are not using a prepared statement here – there are no variables, so there is no point.

We then loop through the results, adding each result to an array:

while ($row = $result->fetch(PDO::FETCH\_NUM)){

$tables[] = $row[0];

}

We then return the array encoded as JSON data:

echo json\_encode($tables);

Note that we are also closing the database connection before returning (either successfully or with an error):

$pdo = null;

This isn't strictly necessary as the connection will be closed automatically when the script ends, but it is considered good practice.

**Activity**

1. Create a php file using the code above.
2. Create an HTML file to call the php and display the results.
3. Check the execution of the code.
4. Check that it correctly detects connection errors by changing your credentials to incorrect values.

## Retrieving Table Contents

The next task we will look at is retrieving the contents of a database table. Consider the getJSON call below:

$.getJSON("getDatabaseTable.php", {tableName:"books"}, function(data){…}

We can see that we are now passing the name of the table to our php function in our getJSON method. Consider the php below:

<?php

if (!isset($error)) {

$error = new stdClass();

}

include "dbinfo.info.php";

try {

$pdo = new PDO("mysql:host=$host;dbname=$database;charset=utf8", $username, $password, [PDO:: ATTR\_ERRMODE => PDO::ERRMODE\_EXCEPTION, PDO::ATTR\_EMULATE\_PREPARES => false]);

} catch (PDOException $e) {

$error->code = "error";

$error->message = "There was a problem connecting to the database";

echo json\_encode($error);

$pdo = null;

return;

}

$table = $\_GET["tableName"];

$stmt = $pdo->prepare("SELECT \* FROM {$table}");

$result = $stmt->execute();

if ($stmt->rowCount() > 0)

{

$tableData = array();

$tableData[] = $stmt->fetchAll(PDO::FETCH\_ASSOC);

echo json\_encode($tableData);

}

else

{

$error->code = "error";

$error->message = "The table: ".$table." contains no rows.";

echo json\_encode($error);

}

$stmt = null;

$pdo = null;

?>

The first part should look familiar, mainly including the connection credentials and connecting to the database.

We are then getting the table name which was passed in the getJSON method in the tableName variable.

$table = $\_GET["tableName"];

Next we are preparing a SQL statement which includes that name of the table. We should be cautious about this table name as it may be a route for SQL injection – malicious SQL code passed as an innocent looking parameter. We may not think this is an issue as we are constraining the user to select the table name from a select box, but remember they could execute the php script directly through the browser. This is why we are using a prepared statement. You may also want to check the table name against the tables in the database before proceeding. There is probably some other error checking needed here too.

$stmt = $pdo->prepare("SELECT \* FROM {$table}");

We are then executing the prepared statement.

$result = $stmt->execute();

If there are rows returned,

if ($stmt->rowCount() > 0)

we are creating a new array,

$tableData = array();

storing the returned data in the array,

$tableData[] = $stmt->fetchAll(PDO::FETCH\_ASSOC);

and returning it.

echo json\_encode($tableData);

If no rows were returned, we report an error.

Finally we close the prepared statement and the connection – not required, but good practice.

$stmt = null;

$pdo = null;

**Activity**

1. Create a php file using the code above.
2. Create an HTML file to call the php and display the results.
3. Check the execution of the code.
4. Check that it correctly detects when no rows are returned.

## Inserting Data into a Table

The final task we will look at is inserting new data into a table. In addition to passing the table name we will need to pass column names and values.

For the purposes of this example, we will assume that we have a database table, called 'test', with the following structure:

|  |  |
| --- | --- |
| **Name** | **Type** |
| Id | Auto-increment int |
| First | text |
| Second | text |
| Third | text |

Note that this table corresponds to the specification given in the coursework.

We will create an Object containing our (field)name: value pairs,

var addData = new Object();

addData.First = 'new';

addData.Second = 'also';

addData.Third = 'test';

Note that the names in the Object match the fieldnames in the table. The values are the strings that we wish to store in the database table. Note particularly that we are not providing a value for Id because it is an auto-increment.

We will then convert this Object to JSON format,

var jsonAddData = JSON.stringify(addData);

We can now pass jsonAddData as a data element in our getJSON() call,

$.getJSON("insertDatabase.php", {tableName:"test", appendData:jsonAddData}, function(data){…}

Now, consider the php below:

<?php

if (!isset($error)) {

$error = new stdClass();

}

include "dbinfo.info.php";

try {

///$dbh = new PDO("mysql:host=$host;dbname=$database;charset=utf8", $username, $password);

$pdo = new PDO("mysql:host=$host;dbname=$database;charset=utf8", $username, $password, [PDO:: ATTR\_ERRMODE => PDO::ERRMODE\_EXCEPTION, PDO::ATTR\_EMULATE\_PREPARES => false]);

// check database connection

} catch (PDOException $e) {

$error->code = "error";

$error->message = "There was a problem connecting to the database";

echo json\_encode($error);

$pdo = null;

return;

}

$table = $\_GET["tableName"];

$appendData = $\_GET['appendData'];

$json\_array = json\_decode($appendData, true);

$query = "INSERT INTO ".$table." (";

$placeholder = "";

$values = array();

foreach($json\_array as $key => $value) {

$query .= $key.", ";

$placeholder .= "?, ";

$values[] = $value;

}

$query = rtrim($query, ', ');

$placeholder = rtrim($placeholder, ', ');

$query .= ") VALUES (".$placeholder.")";

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

$counter = 1;

foreach ($json\_array as $key => &$val) {

$stmt->bindParam($counter, $val);

$counter++;

}

$stmt->execute();

echo $pdo->lastInsertId();

$stmt = null;

$pdo = null;

?>

The first part should again look familiar, mainly including the connection credentials and connecting to the database.

We then get the table name, and get and decode the data we wish to add.

$table = $\_GET["tableName"];

$appendData = $\_GET['appendData'];

$json\_array = json\_decode($appendData, true);

We then go through quite an elaborate looking process before we prepare the query $stmt = $pdo->prepare($query); .

If we look at this carefully we will see that it builds three things at the same time.

* $query – the actual query string
* $placeholder – a list of placeholders for the values which will be bound
* $values – an array holding the values which will be bound to the placeholders

First, these are initialised,

$query = "INSERT INTO ".$table." (";

$placeholder = "";

$values = array();

* $query - the string is 'INSERT INTO test (' – remembering that our table is called 'test'.
* $placeholder – the string is an empty string
* $values – the array is an empty array

The next part loops through the name: value pairs adding the name followed by a comma to the $query string, the value to the $values array, and a placeholder for the value followed by a comma to the $placeholder string

foreach($json\_array as $key => $value) {

$query .= $key.", ";

$placeholder .= "?, ";

$values[] = $value;

}

* $query – is 'INSERT INTO test (First, Second, Third, '
* $placeholder – is '?, ?, ?, '
* $values – is [new, also, test]

At this point we have looped trough all the name: value pairs, so we trim off the unnecessary final comma and space from both $query and $placeholder.

$query = rtrim($query, ', ');

$placeholder = rtrim($placeholder, ', ');

* $query – is 'INSERT INTO test (First, Second, Third'
* $placeholder – is '?, ?, ?'

We now assemble the complete query by appending the $placeholder and some additional SQL to the $query

$query .= ") VALUES (".$placeholder.")";

* $query – is 'INSERT INTO test (First, Second, Third) VALUES (?, ?, ?)'
* $values – is [new, also, test]

So we can see that we have constructed a string containing our SQL, the appropriate table and column names and the correct number of placeholders. We also have the values which need to be bound to the placeholders, in the correct order in the array.

We then prepare the query statement

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

and bind each of the parameters in turn

$counter = 1;

foreach ($json\_array as $key => &$val) {

$stmt->bindParam($counter, $val);

$counter++;

}

Finally, we execute the query statement

$stmt->execute();

We tidy up by echoing the id of the row we inserted – we don't have to return this, but it is generally a good idea to return something useful. We also close the query statement and the database connection.

It is worth noting that this is a very bare bones solution, particularly in regards to error checking. It might be useful to check that data has been passed, table exists, specified columns exist, the right number of column values have been passed and so on.

**Activity**

1. Create a php file using the code above.
2. Create an HTML file to call the php.
3. Check the execution of the code and that the data is inserted correctly into different tables in your database.

This workbook has given you a rough and ready outline of the the types of function required by the coursework. The examples provide just one way of achieving the functions – there are other ways and there may be better ways. The examples also only provide limited error checking. You may wish to check for more errors and may wish to handle them in a more elegant way.