

**Experiment No. 5**

**Title: Database and PHP integration using PDO**

# Batch: B2 Roll No.: 16010420117 Experiment No:5

**Aim:** Write PHP program for database activities using PDO.

**Resources needed:** Windows OS, Web Browser, Editor, XAMPP Server

# Pre Lab/ Prior Concepts:

Students should have prior knowledge of mysql database and PHP constructs.

# Theory:

The PHP Data Objects (PDO) extension defines a lightweight, consistent interface for accessing databases in PHP. Each database driver that implements the PDO interface can expose database- specific features as regular extension functions. Note that you cannot perform any database functions using the PDO extension by itself; you must use a database-specific PDO driver to access a database server.

PDO provides a data-access abstraction layer, which means that, regardless of which database you're using, you use the same functions to issue queries and fetch data. PDO does not provide a database abstraction; it doesn't rewrite SQL or emulate missing features. You should use a full-blown abstraction layer if you need that facility.

# Installation/Configuration Requirement

When installing PDO as a shared module, the php.ini file needs to be updated so that the PDO extension will be loaded automatically when PHP runs. You will also need to enable any database specific drivers there too; make sure that they are listed after the pdo.so line, as PDO must be initialized before the database-specific extensions can be loaded.

extension=php\_pdo.dll extension=php\_pdo\_firebird.dll extension=php\_pdo\_informix.dll extension=php\_pdo\_mssql.dll extension=php\_pdo\_mysql.dll extension=php\_pdo\_oci.dll extension=php\_pdo\_oci8.dll extension=php\_pdo\_odbc.dll extension=php\_pdo\_pgsql.dll extension=php\_pdo\_sqlite.dll

Connecting to mysql

<?php

$dbh = new PDO('mysql:host=localhost;dbname=test', $user, $pass);

?>

If there are any connection errors, a PDOException object will be thrown. You may catch the exception if you want to handle the error condition, or you may opt to leave it for an application global exception handler that you set up via set\_exception\_handler().

<?php try {

$dbh = new PDO('mysql:host=localhost;dbname=test', $user, $pass); foreach($dbh->query('SELECT \* from FOO') as $row) {

print\_r($row);

}

$dbh = null;

} catch (PDOException $e) {

print "Error!: " . $e->getMessage() . "<br/>"; die();

}

?>

# Closing the connection

Upon successful connection to the database, an instance of the PDO class is returned to your script. The connection remains active for the lifetime of that PDO object. To close the connection, you need to destroy the object by ensuring that all remaining references to it are deleted—you do this by assigning null to the variable that holds the object. If you don't do this explicitly, PHP will automatically close the connection when your script ends.

For eg.,

$dbh = null; should be added at the end of the file

# Persistent connection

Many web applications will benefit from making persistent connections to database servers. Persistent connections are not closed at the end of the script, but are cached and re-used when another script requests a connection using the same credentials. The persistent connection cache allows you to avoid the overhead of establishing a new connection every time a script needs to talk to a database, resulting in a faster web application.

<?php

$dbh = new PDO('mysql:host=localhost;dbname=test', $user, $pass, array( PDO::ATTR\_PERSISTENT => true

));

?>

If you wish to use persistent connections, you must set PDO::ATTR\_PERSISTENT in the array of driver options passed to the PDO constructor. If setting this attribute with PDO::setAttribute() after instantiation of the object, the driver will not use persistent connections.

Transactions are typically implemented by "saving-up" your batch of changes to be applied all at once; this has the nice side effect of drastically improving the efficiency of those updates. In other words, transactions can make your scripts faster and potentially more robust (you still need to use them correctly to reap that benefit).

# TRANSACTIONS AND AUTOCOMMIT

Unfortunately, not every database supports transactions, so PDO needs to run in what is known as "auto-commit" mode when you first open the connection. Auto-commit mode means that every query that you run has its own implicit transaction, if the database supports it, or no transaction if the database doesn't support transactions. If you need a transaction, you must use the

PDO::beginTransaction() method to initiate one. If the underlying driver does not support transactions, a PDOException will be thrown (regardless of your error handling settings: this is always a serious error condition). Once you are in a transaction, you may use PDO::commit() or PDO::rollBack() to finish it, depending on the success of the code you run during the transaction.

Example: In the following sample, let's assume that we are creating a set of entries for a new employee, who has been assigned an ID number of 23. In addition to entering the basic data for that person, we also need to record their salary. It's pretty simple to make two separate updates, but by enclosing them within the PDO::beginTransaction() and PDO::commit() calls, we are guaranteeing that no one else will be able to see those changes until they are complete. If something goes wrong, the catch block rolls back all changes made since the transaction was started, and then prints out an error message.

<?php try {

$dbh = new PDO('odbc:SAMPLE', 'db2inst1', 'ibmdb2', array(PDO::ATTR\_PERSISTENT => true));

echo "Connected\n";

} catch (Exception $e) {

die("Unable to connect: " . $e->getMessage());

}

try {

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

$dbh->beginTransaction();

$dbh->exec("insert into staff (id, first, last) values (23, 'Joe', 'Bloggs')");

$dbh->exec("insert into salarychange (id, amount, changedate) values (23, 50000, NOW())");

$dbh->commit();

} catch (Exception $e) {

$dbh->rollBack();

echo "Failed: " . $e->getMessage();

}

?>

# PREPARED STATEMENTS

Prepared Statement can be thought of as a kind of compiled template for the SQL that an application wants to run, that can be customized using variable parameters.

Prepared statements offer two major benefits:

The query only needs to be parsed (or prepared) once, but can be executed multiple times with the same or different parameters. When the query is prepared, the database will analyze, compile and optimize its plan for executing the query. For complex queries this process can take up enough time that it will noticeably slow down an application if there is a need to repeat the same query many times with different parameters. By using a prepared statement the application avoids repeating the

analyze/compile/optimize cycle. This means that prepared statements use fewer resources and thus run faster.

The parameters to prepared statements don't need to be quoted; the driver automatically handles this. If an application exclusively uses prepared statements, the developer can be sure that no SQL injection will occur (however, if other portions of the query are being built up with unescaped input, SQL injection is still possible).

Prepared statements are so useful that they are the only feature that PDO will emulate for drivers that don't support them. This ensures that an application will be able to use the same data access paradigm regardless of the capabilities of the database.

Insert using Prepared Statement

<?php

$stmt = $dbh->prepare("INSERT INTO REGISTRY (name, value) VALUES (?, ?)");

$stmt->bindParam(1, $name);

$stmt->bindParam(2, $value);

// insert one row

$name = 'one';

$value = 1;

$stmt->execute();

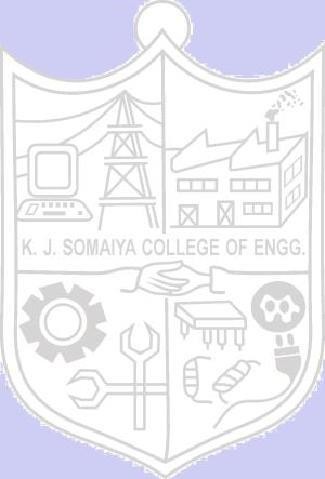
// insert another row with different values

$name = 'two';

$value = 2;

$stmt->execute();

?>



# Retrieval using Prepared statement

<?php

$stmt = $dbh->prepare("SELECT \* FROM REGISTRY where name = ?");

$stmt->execute([$\_GET['name']]); foreach ($stmt as $row) { print\_r($row);

}

?>

# Stored Procedure prepared statement

<?php

$stmt = $dbh->prepare("CALL sp\_takes\_string\_returns\_string(?)");

$value = 'hello';

$stmt->bindParam(1, $value, PDO::PARAM\_STR|PDO::PARAM\_INPUT\_OUTPUT, 4000);

// call the stored procedure

$stmt->execute();

print "procedure returned $value\n";

?>

PDO has following error handling strategies

# PDO::ERRMODE\_SILENT

PDO will simply set the error code for you to inspect using the PDO::errorCode() and PDO::errorInfo() methods on both the statement and database objects;

# PDO::ERRMODE\_WARNING

In addition to setting the error code, PDO will emit a traditional E\_WARNING message. This setting is useful during debugging/testing, if you just want to see what problems occurred without interrupting the flow of the application.

# PDO::ERRMODE\_EXCEPTION

In addition to setting the error code, PDO will throw a PDOException and set its properties to reflect the error code and error information. This setting is also useful during debugging, as it will effectively "blow up" the script at the point of the error, very quickly pointing a finger at potential problem areas in your code (remember: transactions are automatically rolled back if the exception causes the script to terminate).

Exception mode is also useful because you can structure your error handling more clearly than with traditional PHP-style warnings, and with less code/nesting than by running in silent mode and explicitly checking the return value of each database call.

Example #1 Create a PDO instance and set the error mode

<?php

$dsn = 'mysql:dbname=testdb;host=127.0.0.1';

$user = 'dbuser';

$password = 'dbpass';

$dbh = new PDO($dsn, $user, $password);

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

// This will cause PDO to throw a PDOException (when the table doesn't exist)

$dbh->query("SELECT wrongcolumn FROM wrongtable");

*The above example will output:*

*Fatal error: Uncaught PDOException: SQLSTATE[42S02]: Base table or view not found: 1146 Table 'testdb.wrongtable' doesn't exist in /tmp/pdo\_test.php:10*

*Stack trace:*

*#0 /tmp/pdo\_test.php(10): PDO->query('SELECT wrongcol...') #1 {main}*

*thrown in /tmp/pdo\_test.php on line 10*

Example #2 Create a PDO instance and set the error mode from the constructor

<?php

$dsn = 'mysql:dbname=test;host=127.0.0.1';

$user = 'googleguy';

$password = 'googleguy';

$dbh = new PDO($dsn, $user, $password, array(PDO::ATTR\_ERRMODE => PDO::ERRMODE\_WARNING));

// This will cause PDO to throw an error of level E\_WARNING instead of an exception (when the table doesn't exist)

$dbh->query("SELECT wrongcolumn FROM wrongtable");

**Activity:** Write PHP program to work with PDO database.

Complete the following functionalities

* Creating a database.
* Creating a table.
* Inserting values into the table (values to be extracted from user input) using prepared statement.
* Retreiving values from the table using prepared statement and displaying to the user.
* Using Exception handling to show error wherever possible.
* Update some values of the table based on user input.

# Output(Code with result Snapshot)

**Post Lab Questions:-**

1. Write in detail the importance of using Prepared Statements.
2. Write in detail about PDO constructor function.
3. Write a php program to return the id of the last inserted row.

# Outcomes:

**Conclusion: (Conclusion to be based on the objectives and outcomes achieved)**

# Signature of faculty in-charge with date References:

* 1. Thomson PHP and MySQL Web Development Addison-Wesley Professional , 5th Edition2016.
  2. <https://www.php.net/manual/en/book.pdo.php>