**Mutisya Elizabeth Waeni**

**ICS3B**

**107691**

# Internet Application Programming Term Paper

Table of Contents

[**How to run queries using prepared statements** 2](#_Toc40360652)

[Query Execution can also be done using the *bind\_param()* method: 2](#_Toc40360653)

[*The next step is to querying the Database:* 2](#_Toc40360654)

[*Named Placeholders.* 3](#_Toc40360655)

[**How prepared statements thwarts SQL injection** 3](#_Toc40360656)

[**Running Transactions using PDO** 4](#_Toc40360657)

[**Any configuration code changes you will need to make when moving your PHP app from development to production** 6](#_Toc40360658)

[**REFERENCES** 8](#_Toc40360659)

## **How to run queries using prepared statements**

## Query Execution can also be done using the *bind\_param()* method:

The first step is creating a MySQL Connection:

###### ***The first step is creating new MySQL Connection:***

This should be wrapped in a try/catch to handle a database connection error.

1. try {
2. $host = "your host”.
3. $dbname = "your\_database”.
4. $user = "your\_user”.
5. $pass = "qwerrtyuiop12345678”.
7. $dbh = **new** PDO("mysql:host=**$host**;dbname=**$dbname**", $user, $pass).
8. }
9. catch(PDOException $e) {
10. echo  $e->getMessage(); *// handle this better, this is just for the demo!*
11. }

 You have to specify the following information:

* MySQL data source name:  specifies the address of the MySQL database server. For example localhost
* MySQL database name: indicates the name of the database to which you want to connect.
* Username and password: specify username and password of the MySQL’s user that you use to connect to the MySQL database server. The account must have sufficient privileges to access the database specified above.

### *The next step is to querying the Database:*

Every query should be prepared hence one does not have to think of escaping data. One just has to bind the variables to the query.

For example, the example below shows how one can create a product post with the following data.

$title = "Product title”.

$product\_body = “Product body”.

$status = "availability\_status”.

#### Using unnamed placeholders.

$prepared\_statement = $dbh->("INSERT INTO products (title, product\_body, status) values (? ?, ?)").

They are indexed from 1 and then executed

$prepared\_statement->bindParam(1, $title).

$prepared\_statement->bindParam(2, $product\_body).

$prepared\_statement->bindParam(3, $status).

*Execution:*

$prepared\_statement->execute().

### 

### *Named Placeholders.*

On can also use a named placeholder that does not have to match anything else in the sql statement.

Although you will see a colon(:) is used before the names such as ":status"

$prepared\_statement = $dbh->("INSERT INTO blog\_posts (title, product\_body, status) values ( :title, :body, :product\_status )").

The next step is to bind them as follows:

$prepared\_statement->bindParam(":title", $title).

$prepared\_statement->bindParam(":body", $product\_body).

$prepared\_statement->bindParam(":product\_status", $status).

The last step is to execute it.

$prepared\_statement->execute().

## **How prepared statements thwarts SQL injection**

Prepared statements are resilient against SQL injection, because parameter values, which are transmitted later using a different protocol, need not be correctly escaped.

The main cause of an SQL injection problem is the mixing of the data and the code.

For example when we are creating a program dynamically, the data may interfere with the program code and even alter it, as shwn in the example below:

$expected\_data = 1;

$query = "SELECT \* FROM users where id=$expected\_data";

will produce a regular query

SELECT \* FROM users where id=1

while this code

$spoiled\_data = "1; DROP TABLE users;"

$query = "SELECT \* FROM users where id=$spoiled\_data";

will produce a malicious sequence

SELECT \* FROM users where id=1; DROP TABLE users;

It works because we are adding the data directly to the program body and it becomes a part of the program, so the data may alter the program, and depending on the data passed, we will either have a regular output or a table users deleted.

*While in the case of prepared statements the program is not altered,it remains intact. as shown below:*

**sending a program to the server first:**

$db->prepare("SELECT \* FROM users where id=?");

where the data is substituted by some *variable* called a parameter or a placeholder.

Note that exactly the same query is sent to the server, without any data in it! And then we're sending the data with the *second* request, essentially separated from the query itself:

$db->execute($data);

so it can't alter our program and do any harm.

## **Running Transactions using PDO**

transactions ensure that a set of data changes will only be made permanent if every statement is successful. Any query or code failure during a transaction can be caught and you then have the option to roll back the attempted changes.

PDO provides simple methods for beginning, committing, and rollbacking back transactions:

***Begin transaction*** *-* the transaction begins and the default auto commit nature of MySql is disabled. For example, an insert query will not work straight away.

***Commit -*** when one commits, it means that the transaction ran successfully and that the changes that have been made are final.

***Roll back -***  this basically means that if a transaction has failed or there is an error, that means that any changes that had been made will not be made and the information will go back to its previous state.

##### Examples of using transactions in code

Assume a scenario where you are building a shopping cart for an ecommerce website and you decide to keep the records in two tables. Now, one can create two tables, one called "orders" which contains columns of order\_id,name, address, telephone and created\_at, while the other one is a table called "order\_products" with the columns; order\_id, product\_id and quantity. The first table i.e. orders contains data about the orders while the "order\_products" table simply contains data about the products that have been ordered.

###### Inserting a new order using PDO's

We first need to insert a new record inside the orders table and the orders\_products.

At the start of a transaction, the method *begin* ***transaction()***is called before executing any queries.

// In this example we are using MySQL, but this applies to any database that has support for transactions  
$db = new PDO('mysql:host=' . $host . ';dbname=' . $dbname . ';charset=utf8', $username, $password).

// Make sure that PDO will throw an exception in case of error to make error handling easier  
$db->setAttribute(PDO:ATTR\_ERRMODE, PDO:ERRMODE\_EXCEPTION).

try {  
 // From this point and until the transaction is being committed every change to the database can be reverted  
 $db->beginTransaction();

To make any changes to the data in the tables, INSERT or UPDATE queries is used.

// Insert the metadata of the order into the database  
 $preparedStatement = $db->prepare(  
 'INSERT INTO `orders` (`order\_id`, `name`, `address`, `created\_at`)  
 VALUES (:name, :address, :telephone, :created\_at)'  
 );

To make changes to the data permanent, the **commit** method is used. If changes are being currently are not being committed, then they are not permanent and hence, one can simply redo the information by calling the ***rollback()***method of the object that has been created.

$preparedStatement->execute([  
 'name' => $name,  
 'address' => $address,  
 'telephone' => $telephone,  
 'created\_at' => time(),  
 ]);  
   
 // Get the generated `order\_id`  
 $orderId = $db->lastInsertId();

// Construct the query for inserting the products of the order  
 $insertProductsQuery = 'INSERT INTO `orders\_products` (`order\_id`, `product\_id`, `quantity`) VALUES';  
   
 $count = 0;  
 foreach ( $products as $productId => $quantity ) {  
 $insertProductsQuery .= ' (:order\_id' . $count . ', :product\_id' . $count . ', :quantity' . $count . ')';  
   
 $insertProductsParams['order\_id' . $count] = $orderId;  
 $insertProductsParams['product\_id' . $count] = $productId;  
 $insertProductsParams['quantity' . $count] = $quantity;  
   
 ++$count;  
 }  
   
 // Insert the products included in the order into the database  
 $preparedStatement = $db->prepare($insertProductsQuery);  
 $preparedStatement->execute($insertProductsParams);  
   
 // Make the changes to the database permanent  
 $db->commit();

}  
catch ( PDOException $e ) {   
 // Failed to insert the order into the database so we rollback any changes  
 $db->rollback();  
 throw $e;  
}

## **Any configuration code changes you will need to make when moving your PHP app from development to production**

Three files are needed when developing apps in PHP, the files and codes must be configured in order to access them from anywhere. The three files needed are:

***inc/config.php*** *-* this is the file that is to be included.

**inc/config.server.php** - this is where the production configuration is done.

**inc/config.dev.php** - this is where the development configuration is done.

**inc/config.default.php** - this file contains the default configurations of the app.

***Config.php***

In this file, the correct config file should be created according to the server name and the domains should be replaced in both the production and development domains. In additon, the default configurations can be included.

<?php  
if ($\_SERVER['SERVER\_NAME'] === 'production.com') {  
 include\_once 'config.server.php';  
} else if ($\_SERVER['SERVER\_NAME'] === 'localdomain.com') {  
 include\_once 'config.dev.php';  
}

include\_once 'config.default.php’.

each config file will have the following code:

// database config  
define('DB\_HOST', 'localhost');  
define('DB\_USERNAME', 'root');   
// ...

// domain config  
define('PROTOCOL', 'https://');  
// ...

// and any thing

The default config file can contain common configurations to production and development files.

define('MAX\_USERNAME\_LENGTH', 15).

any other configurations in other files, are included in the "config.php" file.

<?php  
include\_once '/path/to/config.php’.

# **REFERENCES**

PHP - Database Transactions with PDO: php Tutorial. (n.d.). Retrieved May 14, 2020, from <https://riptutorial.com/php/example/4186/database-transactions-with-pdo>

PHP - Database Transactions with PDO: php Tutorial. (n.d.). Retrieved May 14, 2020, from <https://riptutorial.com/php/example/4186/database-transactions-with-pdo>

Laravel features you may not know about. (1969, January 1). Retrieved from <https://webdevetc.com/programming-tricks/php/general-php/how-to-query-mysql-using-pdo-in-php>