PHP MySQL Prepared Statements

In this tutorial you will learn how to use prepared statements in MySQL using PHP.

What is Prepared Statement

A prepared statement (also known as parameterized statement) is simply a SQL query template containing placeholder instead of the actual parameter values. These placeholders will be replaced by the actual values at the time of execution of the statement.

MySQLi supports the use of anonymous positional placeholder (`?`), as shown below:

```
INSERT INTO persons (first_name, last_name, email) VALUES (?, ?, ?);
```

While, PDO supports both anonymous positional placeholder (`?`), as well as the named placeholders. A named placeholder begins with a colon (`:`) followed by an identifier, like this:

```
INSERT INTO persons (first_name, last_name, email)
VALUES (:first_name, :last_name, :email);
```

The prepared statement execution consists of two stages: prepare and execute.

- **Prepare** — At the prepare stage a SQL statement template is created and sent to the database server. The server parses the statement template, performs a syntax check and query optimization, and stores it for later use.
- **Execute** — During execute the parameter values are sent to the server. The server creates a statement from the statement template and these values to execute it.

Prepared statements is very useful, particularly in situations when you execute a particular statement multiple times with different values, for example, a series of `INSERT` statements. The following section describes some of the major benefits of using it.

Advantages of Using Prepared Statements

A prepared statement can execute the same statement repeatedly with high efficiency, because the statement is parsed only once again, while it can be executed multiple times. It also minimize bandwidth usage, since upon every execution only the placeholder values need to be transmitted to the database server instead of the complete SQL statement.

Prepared statements also provide strong protection against SQL injection, because parameter values are not embedded directly inside the SQL query string. The parameter values are sent to the database server separately from the query using a different protocol and thus cannot interfere with it. The server uses these values directly at the point of execution, after the statement template is parsed. That's why the prepared statements are less error-prone, and thus considered as one of the most critical element in database security.

The following example will show you how prepared statements actually work:

Example | Procedural | Object Oriented | PDO | Download

```php
<?php
/* Attempt MySQL server connection. Assuming you are running MySQL
server with default setting (user 'root' with no password) */
$link = mysqli_connect("localhost", "root", "", "demo");

// Check connection
if($link === false){
    die("ERROR: Could not connect. " . mysqli_connect_error());
}

// Prepare an insert statement
$sql = "INSERT INTO persons (first_name, last_name, email) VALUES (?,
?, ?)";

if($stmt = mysqli_prepare($link, $sql)){
    // Bind variables to the prepared statement as parameters
    mysqli_stmt_bind_param($stmt, "sss", $first_name, $last_name,
$email);

    /* Set the parameters values and execute
    the statement again to insert another row */
    $first_name = "Hermione";
    $last_name = "Granger";
    $email = "hermionegranger@mail.com";
    mysqli_stmt_execute($stmt);

    /* Set the parameters values and execute
    the statement to insert a row */
    $first_name = "Ron";
    $last_name = "Weasley";
    $email = "ronweasley@mail.com";
    mysqli_stmt_execute($stmt);

    echo "Records inserted successfully.";
} else{
    echo "ERROR: Could not prepare query: $sql. " .
mysqli_error($link);
}
```

```php
// Close statement
mysqli_stmt_close($stmt);

// Close connection
mysqli_close($link);
?>
```

As you can see in the above example we've prepared the `INSERT` statement just once but executed it multiple times by passing the different set of parameters.

Explanation of Code (Procedural style)

Inside the SQL `INSERT` statement (*line no-12*) of the example above, the question marks is used as the placeholders for the *first_name*, *last_name*, *email* fields values.

The `mysqli_stmt_bind_param()` function (*line no-16*) bind variables to the placeholders (`?`) in the SQL statement template. The placeholders (`?`) will be replaced by the actual values held in the variables at the time of execution. The *type* definition string provided as second argument i.e. the "sss" string specifies that the data type of each bind variable is string.

The type definition string specify the data types of the corresponding bind variables and contains one or more of the following four characters:

- **b** — binary (such as image, PDF file, etc.)
- **d** — double (floating point number)
- **i** — integer (whole number)
- **s** — string (text)

The number of bind variables and the number of characters in type definition string must match the number of placeholders in the SQL statement template.

Using Inputs Received through a Web Form

If you remember from the previous chapter, we've created an HTML form to insert data into database. Here, we're going to extend that example by implementing the prepared statement. You can use the same HTML form to test the following insert script example, but just make sure that you're using the correct file name in the `action` attribute of the form.

Here's the updated PHP code for inserting the data. If you see the example carefully you'll find we didn't use the `mysqli_real_escape_string()` to escape the user inputs, like we've done in the previous chapter example. Since in prepared statements, user inputs are never substituted into the query string directly, so they do not need to be escaped correctly.

Example | **Procedural** | Object Oriented | PDO | **Download**

```php
<?php
/* Attempt MySQL server connection. Assuming you are running MySQL
server with default setting (user 'root' with no password) */
$link = mysqli_connect("localhost", "root", "", "demo");

// Check connection
if($link === false){
    die("ERROR: Could not connect. " . mysqli_connect_error());
}

// Prepare an insert statement
$sql = "INSERT INTO persons (first_name, last_name, email) VALUES (?,
?, ?)";

if($stmt = mysqli_prepare($link, $sql)){
    // Bind variables to the prepared statement as parameters
    mysqli_stmt_bind_param($stmt, "sss", $first_name, $last_name,
$email);

    // Set parameters
    $first_name = $_REQUEST['first_name'];
    $last_name = $_REQUEST['last_name'];
    $email = $_REQUEST['email'];

    // Attempt to execute the prepared statement
    if(mysqli_stmt_execute($stmt)){
        echo "Records inserted successfully.";
    } else{
        echo "ERROR: Could not execute query: $sql. " .
mysqli_error($link);
    }
} else{
    echo "ERROR: Could not prepare query: $sql. " .
mysqli_error($link);
}

// Close statement
mysqli_stmt_close($stmt);

// Close connection
mysqli_close($link);
?>
```

Note: Though escaping user inputs is not required in prepared statements, you should always validate the type and size of the data received from external sources and enforces appropriate limits to protect against system resources exploitation.