Hyperiondev

TASK

PHP - MySQL Database

Visit our website

Introduction

BRINGING DATA TO YOUR APP

In the previous level of this course, you had a lot of direct exposure to building databases and managing data, to the point where you may have been asking when you'll actually get to use this knowledge to build a web app. We are pleased to tell you that the day has finally come. In this task, we'll be discussing how to store, retrieve, and manipulate data in a web app using PHP.

HOOKING UP MYSQL

In past tasks, you used a language called SQL to manipulate databases on a platform called MySQL. SQL calls can be automated in PHP by using an extension called **mysqli** (short for MySQL Improved extension). When installing any PHP version from version 5.4 and above, the mysqli extension will be automatically installed.

You will, however, need to ensure that the extension is enabled in the php.ini file by locating the line of code that has **;extension=mysqli** then remove the semicolon prefixing the code and save the file. Ensure that you restart your computer after this change to ensure that the change takes effect.

```
926   ;extension=mbstring
927   ;extension=exif      ; Must be after mbstring as it depends on it
928   extension=mysqli
929   ;extension=oci8_12c  ; Use with Oracle Database 12c Instant Client
930   ;extension=oci8_19   ; Use with Oracle Database 19 Instant Client
931   ;extension=odbc
```

Now create a new database and user by running the following in a MySQL command-line client. For sake of simplicity, we're using an `example` as the basis for all the naming.

```
CREATE DATABASE example_db;

CREATE USER 'example_user'@'localhost' identified by 'Example123';

GRANT ALL ON example_db.* TO 'example_user'@'localhost';
```

With the database created, it's time to create the table we'll be using for this task. It will hold information on a list of to-do items.

```
CREATE TABLE todos (
  id INT AUTO_INCREMENT,
  title VARCHAR(255) NOT NULL,
  created DATETIME DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (id)
);
```

All the code in this task will be written in one script. The final code of the script has been provided to you, in case you have difficulty following the guide. You will also be building on the finished script for the compulsory tasks.

Create a file called **todo.php**, and place the following in it to have PHP connect to the database.

```
<?php
/* Attempt MySQL server connection. */
$mysqli = new mysqli("localhost", "example_user", "Example123", "example_db");

// Check connection
if ($mysqli === false) {
    die("ERROR: Could not connect. " . $mysqli->connect_error);
}

// Print host information
echo "Connect Successfully. Host info: " . $mysqli->host_info;

?>
```

You will see that an object of the mysqli class is created and that this argument took the following properties as arguments:

The **server name** i.e **localhost**.
The **username** which has access to the database i.e. **example_user**.
The **password** for the user i.e. **Example123**.
Lastly, the **name of the database** that you are connecting to i.e. **example_db**.

If the connection is successful, a message is displayed that shows the type of connection, in this case it is a TCP/IP connection. If the connection fails, an error will be displayed.

Note! **Never commit passwords to any repository**. This is a big security risk and potentially exposes your systems to unauthorised access. A better practice is to have a file with all your secrets that you don't commit and then import that file in other scripts. For instance, create a file called `secrets.php` with your database information in the following format:

```php
<?php
// Never commit this file!

define('DB_UID', 'example_user');
define('DB_PWD', 'Example123');
```

This defines constants that will be available anywhere it's imported to. Import and use it as such:

```php
require 'secrets.php';
$mysqli = new mysqli("localhost", DB_UID, DB_PWD, "example_db");
```

INSERTING DATA

Below the database connection, create an HTML form to submit a to-do item:

```php
?>

<form method="post" action="todo.php">
   <input type="text" name="title" placeholder="To-do item">
   <button type="submit">Submit</button>
</form>
```

Now in the PHP code, after the connection success check, add the following code to add a to-do list item into the database if data was POSTed.

```php
if ($_SERVER['REQUEST_METHOD'] === 'POST') {

    $new_title = $_POST['title'];

    $sql = "INSERT INTO todos(title) VALUES (?)";

    if ($stmt = $mysqli->prepare($sql)) {
        $stmt->bind_param("s", $new_title);
        $stmt->execute();
        echo "Records inserted successfully.";
    } else {
        echo "ERROR: Could not prepare query: $sql. " . $mysqli->error;
    }

}
```

This code checks if the request was a POST, in which case it gets the to-do item title that was POSTed. It then sets up a sql query by assigning a string with a placeholder (indicated by the **?** symbol) and assigning it to the $sql variable. In this case the placeholder is used for the title that is sent from the form.

We then need to create a prepared statement.

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 are very useful, particularly in situations when you execute a particular statement multiple times with different values, for example, a series of INSERT statements.

Why not just put the value into the SQL string and execute it immediately? The reason is because of a vulnerability this introduces that can be exploited by SQL injection attacks. SQL injection (SQLi) is a method of exploiting a security vulnerability so that an attacker can interfere with communication with a database, enabling them to send SQL commands to the database that are then executed.

We recommend that you read this article about **SQL injection**.

Moral of the story: **never concatenate to SQL statement strings**.

Now open up **todo.php** in a browser and enter a few to-do items to ensure it works. Check the table in the database to ensure that the new entries were successful.

GETTING DATA

After the PHP code and before the HTML form, insert the following code to get the list of to-do items:

```
// First setup the heading and the table headers
?>
<h2>To-do list items</h2>
<table><tbody>
<tr><th>Item</th><th>Added on</th></tr>

<?php
// create a sql query string for querying the database to populate the table
$sql = "SELECT title, created FROM todos";

// execute the query and check that it is successful
if ($result = $mysqli->query($sql)) {
    // check that there were rows returned
    if ($result->num_rows > 0) {
        /* a loop that goes through each row of the
        query result and fetches it as an associative array
        i.e. an array with named keys for each column */
        while ($row = $result->fetch_array()) {
            /* uses the data from each row of the query result to
            create new table rows and populate with data */
            echo "<tr>";
                echo "<td>" . $row['title'] . "</td>";
                echo "<td>" . $row['created'] . "</td>";
            echo "</tr>";
        }
        // close off the table
        echo "</table>";
```

```
        // Free result set to free up memory associated with the result.
        $result->free();
    } else {
        echo "No records matching your query were found.";
    }
} else {
    echo "ERROR: Could not execute $sql. " . $mysqli->error;
}

?>
```

In the code above, we are first setting up the heading, and the opening tag of the table with a row for the headers. The table is closed once all of the data that is queried from the database is used to populate the rows.

We first create a SQL query using a string to select the data we require from the database.

We then execute that query using the query method from the mysqli object. This is in an if statement so that if there is an error, we can output the error in the else block. We then have a nested if statement that checks whether there is data returned i.e. that there is actually data for that query. This also has an else statement that will trigger an output declaring that there were no results.

We then move onto a while loop that will continue looping for as long as the fetch_array method of teh result object returns a result. This will occur for each row of data from the database result. Each time the fetch_array method is executed, it returns an associative array for each row of the results. The associative array will in this case have keys for “title” and “created” columns and these are then used to populate each row of the table used to populate the data to the browser.

We then close off the table and use the free method of the results object to clear the memory of the associated results from the query.

UPDATING DATA

Let’s allow the user to mark tasks as complete. There are many ways of doing this, but the simplest is changing the task retrieval code to also create a button for each

task that, when clicked, POSTs to the server to have it marked as done. Change the result set loop from above to the following:

```
<tr><th>Item</th><th>Added on</th><th>Complete</th></tr>

<?php
// create a sql query string for querying the database to populate the table
$sql = "SELECT id, title, created FROM todos";

// execute the query and check that it is successful
if ($result = $mysqli->query($sql)) {
    // check that there were rows returned
    if ($result->num_rows > 0) {
        /* a loop that goes through each row of the
        query result and fetches it as an associative array
        i.e. an array with named keys for each column */
        while ($row = $result->fetch_array()) {
            /* uses the data from each row of the query result to
            create new table rows and populate with data */
            echo "<tr>";
                echo "<td>" . $row['title'] . "</td>";
                echo "<td>" . $row['created'] . "</td>";
                echo '<td><form method="post" action="todo.php">
                <input type="hidden" name="id" value="'.$row['id'].'">
                <button type="submit">Done</button>
                </form></td>';
            echo "</tr>";
        }
        // close off the table
        echo "</table>";
```

Updates made (highlighted in dark red):

- Added a table column with a heading for the new buttons.
- Updated SQL query to also fetch each task's ID.
- Added a form with a hidden input that holds each task's ID. The form also has a submit button that will post the ID to the script. Inputs are declared hidden by setting the **`type`** attribute to **`hidden`**, and it's the idiomatic way of sending constant form data.

We should now update the POST handling code to check if an ID was sent and, if so, to update the DB record.

```
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    if (isset($_POST['id'])) {
        // to-do mark task done
    } else {
    // insert new task (earlier code)
    }
}
```

Add the following code in place of the to-do comment to mark the task as complete. For the sake of simplicity, we'll just be deleting the task, but you can use any kind of update statement if your table is set up appropriately.

```
        $id = $_POST['id'];
        $sql = "DELETE FROM todos WHERE id = (?)";

        if ($stmt = $mysqli->prepare($sql)) {
            $stmt->bind_param("s", $id);
            $stmt->execute();
            echo "Records deleted successfully.";
        } else {
            echo "ERROR: Could not prepare query: $sql. " . $mysqli->error;
        }
```

This code should seem very familiar. It is basically the same as the insertion code from earlier, but uses a delete prepared statement with the POSTed ID instead of an insert statement with the POSTed title.

The following code that you have already created should replace the comment containing "insert new task" (earlier code). You can do this by cutting and pasting.

```
        $new_title = $_POST['title'];

        $sql = "INSERT INTO todos(title) VALUES (?)";

        if ($stmt = $mysqli->prepare($sql)) {
            $stmt->bind_param("s", $new_title);
            $stmt->execute();
            echo "Records inserted successfully.";
```

```
        } else {
            echo "ERROR: Could not prepare query: $sql. " . $mysqli->error;
        }
```

Test it out in your browser. You should now have a working to-do app with functionality to add, remove, and view tasks!

Compulsory Task 1

Alter the **todo.php** script to keep and flag completed tasks instead of deleting them.

- Add a column into the to-dos table to keep track of whether a task is complete.
- Change the update logic in PHP to not delete the task, but change this column value of the task appropriately to mark it as complete.
- Change the retrieval logic to show completed tasks by making it appear crossed out in HTML (hint: use `<del>` tags).

Compulsory Task 2

Extend your work in Compulsory Task 1 to hide completed tasks by default and allow the user to see them at will.

- Change the retrieval logic to only get uncompleted tasks.
- Add a button below the tasks labelled “Show completed” that will reload the page and display all tasks (completed and uncompleted).
    - After this, the button should show “Hide completed” with appropriate functionality.
    - For this feature, you’ll need to use a hidden input form with data to indicate what your script should do when POSTed to.

Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved or think we’ve done a good job?

Click here to share your thoughts anonymously.