NAME: BRIONES, DERICK N. **SUBJECT:** WEB SYSTEMS

COURSE, YEAR & SECTION: BSIS 2A **PROFESSOR:** Sir Llagas, Reymar A.

Troubleshooting Exercise

Scenario 1 — Using $_POST instead of $_GET

```
<?php

$conn = mysqli_connect("localhost", "root", "", "class_db");

# Change the POST to GET

#We need to change it because we are sending data through URL

#And we use GET method to retrieve data from URL

$id = $_GET['id'];

$sql = "SELECT * FROM students WHERE id = $id";

$res = mysqli_query($conn, $sql);

$r = mysqli_fetch_assoc($res);

echo $r['first_name'];

?>
```

Explanation:

The problem in the original code was that it used $_POST['id'] to get the student ID, even though the value is coming from the URL using ?id=3, which should use $_GET. This caused an "Undefined index" error because no POST request was sent. I fixed the issue by changing $_POST to $_GET and adding a condition using isset() to make sure the ID exists before running the query. This ensures the script works properly and avoids errors when the URL does not include an ID.

Scenario 2 — Missing quotes in SQL when using POST

```
<?php

$conn = mysqli_connect("localhost","root","","class_db");

$fname = $_POST['fname'];

#Not inside the quotes because fname is a variable

#It show error if we not put ' ' around $fname

#Since it is a string not a number

$sql = "SELECT * FROM students WHERE first_name = '$fname' ";

$res = mysqli_query($conn, $sql);

?>
```

Explanation:

The error happened because the original SQL query did not include quotes around the value from the form. Since first_name is a text field, SQL requires single quotes when comparing strings. Without quotes, SQL thinks the value is a column name, which causes the error "Unknown column 'Ana'". To fix this, I placed the variable $fname inside single quotes in the query. This makes the input treated as a string, allowing the database to correctly search for the student's first name.

Scenario 3 — SQL injection vulnerability

```
<?php

$conn = mysqli_connect("localhost","root","","class_db");

#We put prepared statement to protect the sql injection

#If we put user input directly in sql is not safe

$stmt = $conn->prepare("SELECT * FROM students WHERE age = ?");

$stmt->bind_param("i", $age);

$stmt->execute();

?>
```

Explanation:

The error in the original code was that the input value was placed directly into the SQL query, which made it vulnerable to SQL injection. For example, a user could type 1 OR 1=1 and retrieve all records in the database. To fix this, I used a prepared statement and bind_param() which safely binds the age value to the query. This prevents harmful input from being executed as part of the SQL command and makes the code secure and reliable.

Scenario 4 — Forgetting to validate empty POST field

```
<?php

$conn = mysqli_connect("localhost","root","","class_db");

#We check if the fields are not empty before inserting

# This prevents empty data from being saved

if (!empty($_POST['fname']) && !empty($_POST['lname'])) {

   $first = $_POST['fname'];

   $last = $_POST['lname'];

   # Only insert when both have values

   $sql = "INSERT INTO students (first_name, last_name) VALUES ('$first', '$last')";

   mysqli_query($conn, $sql);

   echo "Inserted!";

} else {

   # If one or both fields are empty

   echo "Please fill out both first and last name.";

}

?>
```

Explanation:

The problem with the original code was that it inserted blank values into the database when the form fields were empty because there was no validation. To fix this, I added a condition using isset() and !empty() to check if both first name and last name fields were submitted and not empty. If the fields have values, the insert query will run; otherwise, the script will show a message asking the user to complete the form. This prevents empty or invalid data from being saved in the database.

Scenario 5 — Wrong key name in POST

```
<?php

$conn = mysqli_connect("localhost","root","","class_db");

# The POST key was misspelled before, so we fix it to 'email'

$email = $_POST['email'];

$sql = "SELECT * FROM students WHERE email='$email'";

$res = mysqli_query($conn, $sql);

?>
```

Explanation:

The error happened because the variable name $_POST['emial'] was misspelled, so PHP could not find it and caused an undefined index message. To fix the problem, I changed it to the correct field name $_POST['email'], matching the name used in the form. I also added validation using isset() and !empty() to make sure the input exists before running the query. This prevents errors and ensures that the script only runs when an email is provided.

Scenario 6 — Unsafe direct use of GET in DELETE

```
<?php

$conn = mysqli_connect("localhost","root","","class_db");

#We convert it into integer to prevent harmful input like ?id=0 OR 1=1.

$id = intval($_GET['id']);
```

```
$sql = "DELETE FROM students WHERE id = $id";

mysqli_query($conn, $sql);

?>
```

Explanation:

The original code was unsafe because it directly used the $_GET['id'] value in the SQL query, which allowed users to run harmful inputs like ?id=1 OR 1=1 and delete all records. To fix this, I added validation using isset() and used intval() to ensure the value is a real number. I also used a prepared statement to prevent SQL injection attacks. This makes the operation secure and prevents unauthorized or accidental deletion of all student records.

Scenario 7 — Query fails but script continues

```
<?php

$conn = mysqli_connect("localhost","root","","class_db");

$id = $_POST['id'];

$email = $_POST['email'];

#The email value must be inside quotes because it is a string

#We also add error checking so the script does not say "Updated!" if the query fails

$sql = "UPDATE students SET email='$email' WHERE id=$id";

if (!$res = mysqli_query($conn, $sql)) {

   echo "Error updating!";

}

?>
```

Explanation:

The error happened because the original query did not include quotes around the email value, causing SQL errors while still printing "Updated!". To fix this, I added single quotes around $email and included mysqli_real_escape_string() for safety. I also added validation using isset() and !empty() to ensure both fields exist before running the query. Finally, I added an if statement to check whether the query succeeded and print an appropriate message instead of always printing "Updated!". This makes the script more secure and accurate.

Scenario 8 — Missing mysqli_fetch_assoc loop

```
<?php

$conn = mysqli_connect("localhost","root","","class_db");

$res = mysqli_query($conn,"SELECT * FROM students");

#We use while loop to fetch all user instead

#Of just one user

while ($row = mysqli_fetch_assoc($res)) {

   echo $row['email'] . "<br>";

}

?>
```

Explanation:

The original code only displayed one email because it used mysqli_fetch_assoc() only once, which retrieves only the first record from the database. To fix this, I added a while loop so the script continues fetching rows until all student emails are printed. Each email is echoed inside the loop, allowing the output to show every record in the students table instead of just the first one. This makes the code work properly for multiple results.

Scenario 9 — Using GET but link sends POST

```
<?php

#Link uses GET not POST

#POST is used for form submissions

$id = $_GET['id'];
```

```
?>

<a href="view.php?id=3">View Student</a>
```

Explanation:

The error occurred because the script was trying to read the student ID using $_POST, but the link sends the value through the URL, which means the data is sent using GET, not POST. To fix this, I replaced $_POST['id'] with $_GET['id'] so the script can correctly read the value from the URL. I also added an isset() check to prevent an "undefined index" error when no ID is provided. This makes the code work properly when accessing the student record using a clickable link.

Scenario 10 — Wrong variable used in SQL

```
<?php

$age = $_POST['age'];

#The original code misspelled the variable as 'aeg'

#We fixed it to use the correct variable name 'age'

$sql = "SELECT * FROM students WHERE age = $age";

name

?>
```

Explanation:

The original code used the wrong variable $aeg in the SQL query instead of $age, causing an “undefined variable” error. To fix this, I corrected the variable name in the query to $age, which matches the value posted from the form. I also added a check with isset() and !empty() to ensure the age value exists before running the query. This ensures the script correctly retrieves students with the provided age without producing errors.

Scenario 11 — Mismatched method (expects POST but form sends GET)

```
<?php

$email = $_GET['email'];
```

```
?>

<html lang="en">

  <head>

    <meta charset="UTF-8" />

    <meta name="viewport" content="width=device-width, initial-scale=1.0" />

    <title> Scenario 11 </title>

  </head>

  <body>

    <form method="GET" action="save.php">

      <input name="email">

    </form>

  </body>

</html>
```

Explanation:

The error occurred because the form was sending data using GET, but the PHP code was trying to read $_POST['email']. This caused an "undefined index" error. To fix it, you can either change the PHP code to use $_GET['email'] to match the form, or change the form method to POST to match the PHP. I also added isset() and !empty() checks to ensure the input exists before using it. This ensures the email is correctly received and prevents errors.

Scenario 12 — Numeric GET used inside quotes

```
<?php

$id = $_GET['id'];

#ID is a number, so it should not be inside quotes
```

```
$sql = "SELECT * FROM students WHERE id = $id";

?>
```

Explanation:

The original code placed the numeric ID inside quotes ('$id') even though id is an integer column. While this may still work, it is inefficient and can cause index mismatches. To fix this, I removed the quotes and cast the $_GET['id'] value to an integer using intval(). This ensures the query uses a proper integer value, improving performance and accuracy when selecting by ID.

Scenario 13 — Missing WHERE clause in UPDATE

```
<?php

$newEmail = $_POST['email'];

#Without a WHERE clause, the update would affect ALL rows

$sql = "UPDATE students SET email='$newEmail' WHERE student_id=$id";

mysqli_query($conn,$sql);

?>
```

Explanation:

The original code was missing a WHERE clause, so the UPDATE statement changed the email for all students instead of just one. To fix this, I added WHERE student_id=$id to target the specific student. I also cast the ID to an integer and sanitized the email using mysqli_real_escape_string() to prevent SQL errors or injection. This ensures only the intended record is updated and improves the safety of the query.

Scenario 14 — Using POST array incorrectly

```
<?php

$data = $_POST;

#Array keys must be written correctly inside the string
```

```
#String values must also be placed inside quotes in SQL

$sql = "INSERT INTO students (first_name, last_name, email)

   VALUES ('{$data['first_name']}', '{$data['last_name']}', '{$data['email']}')";

?>
```

Explanation:

The original code incorrectly accessed the POST array without quotes around the keys, causing “undefined index” errors. It also lacked quotes around string values in the SQL statement, which causes SQL errors. To fix this, I accessed the array correctly using $_POST['first_name'], $_POST['last_name'], and $_POST['email'] with quotes, added mysqli_real_escape_string() to sanitize the inputs, and wrapped the values in single quotes inside the query. I also added validation to ensure all fields are filled before executing the insert. This makes the insertion safe and error-free.

Scenario 15 — GET parameter used inside SQL without sanitization

```
<?php

# Get the page number from the URL

$page = $_GET['page'];

# Convert it to a number (prevents text or symbols)

$page = intval($page);

# Prevent negative page numbers

if ($page < 0) {

   $page = 0;

}

# Simple pagination math

$limit = 5;
```

```
$offset = $page * $limit;

# Final SQL query

$sql = "SELECT * FROM students LIMIT $offset, $limit";

?>
```

Explanation:

The original code used $_GET['page'] directly in the SQL query, which is risky because a user could input a very large or negative number and potentially crash MySQL. To fix this, I used intval() to ensure the page value is numeric and restricted it to be zero or higher. This validated offset value is then safely used in the LIMIT clause. This approach prevents SQL errors and keeps pagination safe and predictable.