RYECCO ADRIAN B. SAMBAJON
BSIS 2-A
WEB SYSTEMS

TROUBLESHOOTING

SCENARIO 1:

Changing in line #3 *$id = $_POST['id'] to - GET*
Explanation:
The page was opened using the URL so we should use GET

SCENARIO 2:

Changing in line #4 *$sql = "SELECT * FROM students WHERE first_name = $fname"; - '$fname' ";*
Explanation:
Names are words like *Ana*, not numbers. Words must be inside quotes ' ' in SQL.

SCENARIO 3:

The problem in scenario 3 is that someone could type 1 or 1=1 into the URL and break your program or steal information. In this scenario, we should use prepared statements.

```
$stmt = $conn->prepare("SELECT * FROM students WHERE age = ?");
$stmt->bind_param("i", $age);
$stmt->execute();
```

SCENARIO 4:

Checking first if fields are empty before inserting.

```
if (!empty($first) && !empty($last)) {
    // insert
}
```

In the php code in scenario 4, it inserted a student even if the form was empty.

SCENARIO 5:

Changing in line #3 *$email = $_POST['emial']; - ['email']*
The reason for the error is TYPO.

SCENARIO 6:

```php
<?php
// Connect to the database
$conn = mysqli_connect("localhost", "root", "", "class_db");

// Fix: Sanitize the input to ensure it is treated as a safe integer.
// intval() returns the integer value of a variable.
$id = intval($_GET['id']);
```

```
// Check if $id is a valid, non-zero ID before running the DELETE query
if ($id > 0) {
    // Now the SQL query uses a guaranteed integer value, making injection
impossible.
    $sql = "DELETE FROM students WHERE student_id = $id"; // Use
student_id as per table

    if (mysqli_query($conn, $sql)) {
        echo "Student with ID $id deleted successfully.";
    } else {
        echo "Error deleting record: " . mysqli_error($conn);
    }
} else {
    echo "Invalid student ID.";
}
?>
```

SCENARIO 7:

```
<?php
// Connect to the database
$conn = mysqli_connect("localhost", "root", "", "class_db");
$id = $_POST['id'];
$email = $_POST['email'];

// Sanitize/escape input (e.g., for security and proper quoting)
$id_safe = intval($id);
$email_safe = mysqli_real_escape_string($conn, $email);

// Fix: Add single quotes around the string variable $email_safe
$sql = "UPDATE students SET email='$email_safe' WHERE
student_id=$id_safe";

// Fix: Check the return value of mysqli_query
$res = mysqli_query($conn, $sql);

if ($res) {
    echo "Updated!";
} else {
    // Print the MySQL error message for troubleshooting
```

```
    echo "Error updating record: " . mysqli_error($conn);
}
?>
```

SCENARIO 8:

We can use a loop in this scenario.

```
 while ($row = mysqli_fetch_assoc($res)) {
    echo $row['email'];
}
```

SCENARIO 9:

The HTML link sends the parameter through the GET method. However, the buggy PHP code attempts to access the ID using $_POST['id']. Since no POST data is sent, this results in an Undefined index error

```
<?php
// Fix: Use $_GET to access parameters sent via the URL query string.
$id = $_GET['id'];
// Add validation/sanitization, e.g., $id = intval($_GET['id']);
// ... rest of the code to fetch student details
?>
<a href="view.php?id=3">View Student</a>
```

SCENARIO 10:

The variable $age was misspelled to $aeg.

```
<?php
$age = $_POST['age'];
$sql = "SELECT * FROM students WHERE age = $age"; // corrected variable
name
?>
```

SCENARIO 11:

The HTML form uses method=”GET” while the PHP form retrieve the data using $_POST[‘email’].

```
<form method="POST" action="save.php">
    <input name="email">
</form>
```

SCENARIO 12:

```php
<?php
// Retrieve the ID
$id = $_GET['id'];

// Fix: Sanitize and cast the input to an integer, then remove the quotes
from the SQL.
$id_int = intval($id);

// The query is correct for an integer column without quotes.
$sql = "SELECT * FROM students WHERE student_id = $id_int";

// ... rest of the code
?>
```

SCENARIO 13:

```php
<?php
// Assume you need to update a specific student, identified by ID
$newEmail = $_POST['email'];
$id = $_POST['id']; // Must be retrieved from POST/GET

// Sanitize/escape input
$newEmail_safe = mysqli_real_escape_string($conn, $newEmail);
$id_safe = intval($id);

// Fix: Add a WHERE clause to restrict the update to the target record.
$sql = "UPDATE students SET email='$newEmail_safe' WHERE student_id =
$id_safe";

mysqli_query($conn, $sql);
// ... rest of the code
?>
```

SCENARIO 14:

```php
<?php
// Connect to the database
$conn = mysqli_connect("localhost", "root", "", "class_db");
$data = $_POST;

// Sanitize/escape all string values before use
$first_name_safe = mysqli_real_escape_string($conn, $data['first_name']);
$last_name_safe = mysqli_real_escape_string($conn, $data['last_name']);
$email_safe = mysqli_real_escape_string($conn, $data['email']);

// Fix: Use correct array indexing AND ensure string values are surrounded
by single quotes in the SQL.
$sql = "INSERT INTO students (first_name, last_name, email)
        VALUES ('$first_name_safe', '$last_name_safe', '$email_safe')";

// ... rest of the code
?>
```

SCENARIO 15:

```php
<?php
$limit = 5;
$page = $_GET['page'];

// Fix 1: Sanitize input to ensure it is a safe, non-negative integer.
// Fix 2: Implement validation to prevent extremely large or nonsensical
numbers.
$page_int = intval($page);

// Ensure $page is not less than 1. If it is, default to 1.
$page_safe = max(1, $page_int);

// In a real app, you would also check if $page_safe exceeds the total
number of pages.
// ...

// Calculate offset (subtract 1 from page number for 0-based offset)
$offset = ($page_safe - 1) * $limit;

// Now the offset is a safe integer and within reasonable bounds.
```

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

// ... rest of the code
?>
```