The Case of the Phantom Profile

Background Story

In the quiet town of Debugville, nestled within the sprawling Code Campus, there lived a web developer named Sam. Sam was a seasoned coder, well-versed in the mystical arts of PHP and SQL, but recently he had taken on a project that tested his every skill.

The task seemed simple enough at first: create a profile page for the students of Code Campus, where they could view and update their information. But soon after launching, the students began reporting bizarre issues. Sometimes, after updating their profiles, the page would show an error, or worse, revert to old data. There were whispers among the students that the page was haunted.

Sam knew this was no ordinary bug. He had to get to the bottom of this before the mysterious Phantom Profile caused chaos in the Campus.

Challenge

Sam needed to create a database table called ‘users’ with columns ‘id’, ‘name’, and ‘email’. Inserting data was up to him, though phpMyAdmin seemed like the easiest option.

He also needed to build a profile page that fulfilled the following requirements:

1. **GET Request**: When a user navigated to profile.php?userId=123, the page must retrieve the user’s data from the database using a **prepared statement**. This would ensure that only the correct profile information was displayed, and no pesky SQL injections could interfere.

2. **Output of Database Data**: The profile page should display all user details retrieved from the database in a readable format, allowing students to see their current information and confirm that updates were successful.

3. **Form Submission**: The page had to include a form for students to update their information, such as their name and email. When the user submitted the form, a **POST request** would be sent to updateProfile.php.

4. **Database Update with Prepared Statement**: The updateProfile.php script must update the database with the new information using a **prepared statement**. This would ensure safe and accurate updates, even with the most temperamental inputs.

5. **Post/Redirect/Get (PRG) Pattern**: After a successful update, instead of showing the same page again (and risking a double submission if the user refreshed), updateProfile.php would redirect the user back to the profile.php page. This would be achieved using the Post/Redirect/Get (PRG) pattern, providing a clean and user-friendly experience.

6. **Session Messages**: To inform students whether their updates were successful or if an error occurred, Sam needed to use **session variables**. After updating the profile, updateProfile.php would store a message in a session variable. Upon redirection to profile.php, this message would be displayed to the user.

Sample Scenario

Let’s imagine that student ID 101 belongs to Alice, who wants to update her email address.

1. Alice navigates to profile.php?userId=101.

The page executes a **GET request**, fetching her details from the database using a prepared statement. The page shows her name as “Alice” and her email as “alice@example.com”.

2. Alice updates her email to “alice.new@example.com” and clicks the “Save” button.

This sends a **POST request** to updateProfile.php. The script updates the database using a **prepared statement**, changing Alice’s email.

3. The updateProfile.php script then uses the **PRG pattern** to redirect Alice back to profile.php?userId=101.

The page loads with a success message, “Profile updated successfully!” displayed at the top.

4. Alice sees her updated email on the page: “alice.new@example.com”. Mission accomplished!