Refactoring

1. Ensure Single Responsibility for PHP Functions

Concept:

- Each function should do **only one thing**.
- This improves **readability, testability, and maintainability**.

Example Before Refactoring:

```
function loginUser($username, $password) {
    // Validate input
    if(empty($username) || empty($password)) return false;

    // Check database
    $result = $conn→query("SELECT * FROM users WHERE username='$usern
ame'");

    // Verify password
    if(password_verify($password, $result['password'])) {
        // Start session
        session_start();
        $_SESSION['user'] = $username;
        return true;
    }
    return false;
}
```

Problems:

- Handles **validation, DB access, password verification, and session creation** all in one function.

Refactored Approach:

```
function validateInput($username, $password) {
    return !empty($username) && !empty($password);
}

function getUserFromDB($conn, $username) {
    $stmt = $conn->prepare("SELECT * FROM users WHERE username=?");
    $stmt->bind_param("s", $username);
    $stmt->execute();
    return $stmt->get_result()->fetch_assoc();
}

function verifyPassword($inputPassword, $hashedPassword) {
    return password_verify($inputPassword, $hashedPassword);
}

function startUserSession($username) {
    session_start();
    $_SESSION['user'] = $username;
}
```

✅ **Benefits:** Each function has a clear responsibility. Easy to test independently.

2. Separate Authentication, Session Management, and Result Display

Why:

- Keeps **logic modular**
- Easier to update security features without touching unrelated code

Suggested Structure:

```
src/
├── auth/
│   ├── login.php
│   ├── logout.php
│   └── register.php
├── sessions/
│   └── session_manager.php
├── results/
│   └── display_results.php
```

Example: session_manager.php

```php
<?php
session_start();

function isLoggedIn() {
    return isset($_SESSION['user']);
}

function requireLogin() {
    if(!isLoggedIn()) {
        header("Location: login.php");
        exit();
    }
}

function logoutUser() {
    session_unset();
    session_destroy();
}
?>
```

- `auth/` → Handles login/logout
- `sessions/` → Manages sessions
- `results/` → Handles result display logic

3. Use PDO with Prepared Statements

Why:

- **Prevents SQL Injection**
- Provides a **clean, object-oriented DB interface**

Example:

```
// db.php
$dsn = 'mysql:host=db;dbname=student_portal;charset=utf8';
$pdo = new PDO($dsn, 'root', 'root', [
    PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION
]);

// Fetch student result
function getStudentResult($pdo, $student_id) {
    $stmt = $pdo->prepare("SELECT * FROM results WHERE student_id = :id");
    $stmt->execute(['id' => $student_id]);
    return $stmt->fetch(PDO::FETCH_ASSOC);
}
```

✅ **Benefits:** Safe from SQL Injection, easier to maintain than `mysqli`.

---

4. Implement CSRF Tokens for Forms

Why:

- Prevents **Cross-Site Request Forgery (CSRF)** attacks where a malicious site tricks a user into submitting forms.

Implementation Steps:

1. Generate a CSRF token in **session** when loading a form:

```
if (empty($_SESSION['csrf_token'])) {
    $_SESSION['csrf_token'] = bin2hex(random_bytes(32));
}
```

1. Include token in HTML form:

```
<form method="POST" action="update_result.php">
    <input type="hidden" name="csrf_token" value="<?php echo $_SESSION['csrf_token']; ?>">
    <input type="text" name="marks">
    <input type="submit" value="Update">
</form>
```

1. Verify token on form submission:

```
if ($_POST['csrf_token'] !== $_SESSION['csrf_token']) {
    die("Invalid CSRF token");
}
```