**Technical Interview Project Question**

**Part 1: Clean the unclean code**

<?php

function addUser($user, $conn)

{

$stmt = $conn->prepare("INSERT INTO users (name, email, role) VALUES (?, ?, ?)");

$stmt->bind\_param("sss", $user['name'], $user['email'], $user['role']);

$stmt->execute();

echo $stmt->affected\_rows > 0 ? "User added successfully." : "Failed to add user.";

$stmt->close();

}

function deleteUser($userId, $conn)

{

$stmt = $conn->prepare("DELETE FROM users WHERE id = ?");

$stmt->bind\_param("i", $userId);

$stmt->execute();

echo $stmt->affected\_rows > 0 ? "User deleted successfully." : "Failed to delete user.";

$stmt->close();

}

function updateUser($user, $conn)

{

$stmt = $conn->prepare("UPDATE users SET name = ?, email = ?, role = ? WHERE id = ?");

$stmt->bind\_param("sssi", $user['name'], $user['email'], $user['role'], $user['id']);

$stmt->execute();

echo $stmt->affected\_rows > 0 ? "User updated successfully." : "Failed to update user.";

$stmt->close();

}

function listUsers($conn)

{

$result = $conn->query("SELECT \* FROM users");

if ($result->num\_rows > 0) {

while ($row = $result->fetch\_assoc()) {

echo "ID: " . $row["id"] . " - Name: " . $row["name"] . " - Email: " . $row["email"] . " - Role: " . $row["role"] . "<br>";

}

} else {

echo "No users found.";

}

}

function manageUser($user, $operation, $conn)

{

switch ($operation) {

case 'add':

addUser($user, $conn);

break;

case 'delete':

deleteUser($user['id'], $conn);

break;

case 'update':

updateUser($user, $conn);

break;

case 'list':

listUsers($conn);

break;

default:

echo "Invalid operation.";

}

}

// Example usage

$conn = new mysqli("localhost", "username", "password", "database");

if ($conn->connect\_error) {

die("Connection failed: " . $conn->connect\_error);

}

$user = [

'id' => 1,

'name' => 'John Doe',

'email' => 'john@example.com',

'role' => 'admin'

];

manageUser($user, 'add', $conn);

$conn->close();

**Explanation of Changes:**

1. **Separated Concerns**:
   * Each database operation (addUser, deleteUser, updateUser, listUsers) is moved to its own function. This separation adheres to the **Single Responsibility Principle** from SOLID, making the code modular and easier to maintain.
2. **Use of Prepared Statements**:
   * I replaced string concatenation in SQL queries with prepared statements ($conn->prepare()) to prevent SQL injection attacks and improve security. The use of bind\_param() ensures proper data binding.
3. **Improved Naming Conventions**:
   * Variables and functions are given meaningful names to make the code more readable ($stmt for statement objects, $userId for user IDs, etc.).
4. **Ternary Operators for Feedback**:
   * I used ternary operators to simplify result-checking logic, making the code more concise (e.g., checking if rows are affected).
5. **Error Handling**:
   * The database connection is checked with a more explicit error message ($conn->connect\_error). This prevents proceeding with the operation if the connection fails.
6. **Avoiding SQL Injection**:
   * By using prepared statements and binding parameters, this refactoring enhances security and prevents SQL injection vulnerabilities.

**Part 2**

**1. Functional and Non-Functional Requirements**

**Functional Requirements**

* **User Management:**
  + Admin can add, update, delete, and manage users (students, librarians).
  + Students can register, log in, and update their profiles.
  + Librarians can log in and manage library resources.
* **Book Management:**
  + Admin/librarian can add, update, and remove books.
  + Students can view available books.
  + Students can borrow and return books, and the system will track borrowed books with due dates.
* **Borrowing System:**
  + Students can borrow a book if available.
  + The system sends reminders to students before the due date.
  + Librarians can manually return books and update the status of borrowed books.
* **Search:**
  + Students can search for books by title, author, genre, or availability.
* **Penalties:**
  + Students receive notifications of overdue books and penalties for late returns.

**Non-Functional Requirements**

* **Performance:** The system should respond within 2 seconds for user queries and handle at least 100 concurrent users.
* **Security:**
  + Only authorized users can access the system.
  + Use encryption for passwords and sensitive data.
* **Usability:**
  + The system should be user-friendly with clear navigation for all types of users.
* **Scalability:** The system should be able to scale as the library adds more users and books.
* **Maintainability:** Code should be modular and adhere to best practices, making future changes easy.
* **Availability:** The system should have 99.9% uptime.

