Assignment 03

SOFTWARE REENGINEERING

SE-4001

Course Instructor:

Dr. Isma Ul Hassan

Group Members:

Zeeshan Ali	20i-2465
Ans Zeshan	20i-0543
Saad Shafiq	20i-1793
Hammad Aslam	20i-1777
Dayyan Shahzad	20i-2393

Due Date:

Nov 19, 2023

Contents

1.1 - In Admin folder:

1.1.1 - Code Smells:

i) **Long Method:** The PHP code within the HTML file is quite lengthy. It's generally a good practice to separate concerns by keeping HTML, CSS, and PHP code in different files unless necessary. This separation makes the code more readable and maintainable.

ii) **Duplicate Code:** There are repeated sections of code, especially within the HTML table rows and PHP echo statements. You might want to create functions or loops to handle these repetitive tasks, reducing code duplication.

iii) **Hardcoded Values:** There are hardcoded values (like URLs and database queries) scattered throughout the code. This can be problematic for maintenance. Using constants or configuration files for such values can make the code more flexible and easier to update.

iv) **Lack of Modularization:** The code could benefit from more modularization. For instance, the database queries and session handling could be encapsulated in separate functions or classes, improving code reuse and readability.

v) **Inline Styles:** There are several inline styles in HTML. Moving these to a separate CSS file can improve the readability of your HTML and makes it easier to manage styles globally.

vi) **Magic Strings and Numbers:** The code contains "magic strings" (like specific user types such as 'a' for admin) and numbers (like column widths and padding). It's generally better to use named constants for such values, which makes the code more readable and easier to modify.

vii) **Deep Nesting:** There is deep nesting in several places, particularly within PHP blocks. Reducing the nesting level by splitting the code into functions or handling different cases separately can improve readability.

viii) **Error Handling:** It's not clear if there's adequate error handling, especially around database queries. Robust error handling would make the code more reliable and easier to debug.

ix) **Comments:** While there are some comments, they are quite sparse and don't always explain the purpose or logic of the code. Adding more descriptive comments can make the code easier to understand and maintain.

x) **Security Concerns:** While not directly a code smell, it's important to ensure that the code is secure, especially regarding SQL injection and session management. Prepared statements for database queries and secure session handling practices are crucial.

1.1.2 - Refactoring Techniques:

- **Separate PHP Logic from HTML:**
  Move PHP code that handles session checks, database connections, and queries to separate PHP files or classes.
- **Create Reusable Functions:**

Create functions for repetitive tasks like generating table rows or menu items. e.g.,

```
function createMenuItem ($name, $link, $iconClass) {
   // Implementation
}
```

- **Externalize CSS Styles:**
  Move inline CSS styles to an external stylesheet. Link the stylesheet in the HTML head.
- **Use Configuration Files:**
  Store configuration details like database connection parameters in a separate config file.
- **Implement Templates:**
  Consider using a templating engine like Twig or Smarty to separate HTML structure from PHP logic.
- **Refactor HTML Structure:**
  Simplify HTML by removing unnecessary nested tables and inline styles. Use semantic HTML tags and CSS classes for styling.
- **Improve Readability:**
  Add more comments to explain the purpose of functions and complex logic. Format the code consistently to improve readability.
- **Improve Security:**
  Use prepared statements for database queries to prevent SQL injection. Validate and sanitize all user inputs.
- **Error Handling:**
  Add error handling around database operations and other critical sections.
- **Remove Magic Strings and Numbers:**
  Replace magic strings with named constants or use configuration files.

1.1.2.1 - Implementation:

```
// PHP Functions File (functions.php)
function checkSession() {
   // Session check logic
}

function getDatabaseConnection() {
   // Database connection logic
}

// Main PHP File (main.php)
include('functions.php');
checkSession();

$db = getDatabaseConnection();
```

```
// HTML File (index.html)
<!DOCTYPE html>
<html lang="en">
<head>
   <!-- Head content -->
</head>
<body>
   <?php include('main.php'); ?>
   <!-- HTML content -->
</body>
</html>
```

1.2 - In another folder:

1.2.1 - Code Smells:

i) **Repetitive Database Connection Logic:** The line include("../connection.php"); is repeated. It would be better to include it just once at the beginning of the script.

ii) **Mixed Concerns:** The script is handling user session validation, database operations, and redirection all in one place. Separating these concerns into different functions or classes would improve readability and maintainability.

iii) **Lack of Error Handling:** There is no visible error handling for database operations. Consider adding error handling to manage potential issues with database connectivity or query execution.

iv) **Direct $_GET and $_SESSION Access:** Directly accessing global variables like $_GET and $_SESSION is generally not recommended as it can lead to security issues. It would be better to encapsulate the access to these superglobals.

v) **Redundant Database Connection in $_GET Block:** The script includes the database connection file again inside the $_GET block. Since the connection is already included at the beginning, this is unnecessary.

vi) **Hardcoded Redirection Paths:** The redirection paths ("../login.php" and "../logout.php") are hardcoded. Using a configuration file or constants for these URLs would make the code more flexible.

vii) **Possible Inconsistencies in Database Operations:** The script is deleting a user from the web user table and then from the patient table using the same email. It's important to ensure that these operations are consistent and consider what should happen if one operation fails but the other succeeds.

viii) **No Feedback on Operations:** The script performs database operations and then redirects without providing any feedback to the user. In a user-facing application, it’s often helpful to inform the user about the outcome of their actions.

ix) **Use of header for Redirection:** While not a smell per se, it's important to ensure that no output is sent to the browser before calling header for redirection, as this can cause issues.

x) **Security Considerations:** Ensure that session hijacking and fixation are addressed. Also, consider the implications of allowing a GET request to delete a user, as it might be susceptible to CSRF attacks.

1.2.2 - Refactoring Techniques:

To improve this code, consider the following refactoring steps:

- **i)** Encapsulate session handling, database operations, and redirection logic in separate functions or classes.
- **ii)** Implement error handling for database operations.
- **iii)** Use a centralized approach for managing database connections.
- **iv)** Add CSRF protection for actions performed via GET requests.
- **v)** Provide user feedback for operations, especially those that affect user data or access.

1.2.2.1 - Implementation:

Step 1: Separate Concerns

Create a separate file for common functions, like handling database connections and user session checks.

database.php:

```
<?php
function getDatabaseConnection() {
    include("../connection.php");
    return $database; // Assuming $database is your database
    connection variable.

}

?>
```

session.php:

```
<?php

function checkUserSession() {

    session_start();

    if (!isset($_SESSION["user"]) || $_SESSION["user"] ==
    "" || $_SESSION['usertype'] != 'p') {

        header("location: ../login.php");

        exit;
```

```
        }
        return $_SESSION["user"];
    }
    ?>
```

Step 2: Refactor Main Script

main_script.php:

```
<?php
include("database.php");
include("session.php");
$useremail = checkUserSession();
$database = getDatabaseConnection();
function getUserDetails($database, $useremail) {
   $sql = "SELECT * FROM patient WHERE pemail=?";
   $stmt = $database->prepare($sql);
   $stmt->bind_param("s", $useremail);
   $stmt->execute();
   return $stmt->get_result()->fetch_assoc();
}
$userDetails = getUserDetails($database, $useremail);
$userid = $userDetails["pid"];
$username = $userDetails["pname"];

if ($_SERVER['REQUEST_METHOD'] == 'GET' &&
isset($_GET["id"])) {
   deleteUser($database, $_GET["id"]);
}
function deleteUser($database, $id) {
```

```
    // Fetch email before deleting
    $email = getEmailFromId($database, $id);
    // Delete operations
    deleteFromTable($database, 'webuser', $email);
    deleteFromTable($database, 'patient', $email);
    header("location: ../logout.php");
    exit;
}
function getEmailFromId($database, $id) {
    $sql = "SELECT pemail FROM patient WHERE pid=?";
    $stmt = $database->prepare($sql);
    $stmt->bind_param("i", $id);
    $stmt->execute();
    return $stmt->get_result()->fetch_assoc()["pemail"];
}
function deleteFromTable($database, $table, $email) {
    $sql = "DELETE FROM $table WHERE email=?";
    $stmt = $database->prepare($sql);
    $stmt->bind_param("s", $email);
    $stmt->execute();
}
?>
```

Notes (Delete-account.php):

Separation of Concerns: The main PHP script is now more organized with separate functions handling specific tasks.

Security: The direct access to $_GET and $_SESSION is handled in a more controlled manner. CSRF protection should still be considered for the GET request that triggers the deletion.

Error Handling: This refactoring does not include detailed error handling, which should be added, especially around database operations.

Database Connection: The connection is established once and passed around as needed.

Functionality: The core functionality remains the same, but the code is cleaner and should be more maintainable.

1.3 - login.php:

1.3.1 - Code Smells:

- **SQL Injection Vulnerability:** The code directly uses POST data in SQL queries, making it vulnerable to SQL injection attacks.
- **Repetitive Code:** The logic for validating user credentials is repeated for different user types (patient, admin, doctor).
- **Session Variable Initialization:** Initializing session variables like $_SESSION["user"] and $_SESSION["usertype"] to empty strings at the beginning seems unnecessary if they are going to be set later.
- **Direct $_POST Access:** Directly accessing $_POST data without validation or sanitization can be risky.
- **Lack of Separation of Concerns:** HTML, PHP, and SQL are all intermingled, making the code harder to read and maintain.
- **Hardcoded Redirect Paths:** Paths like 'location: patient/index.php' are hardcoded, which could be problematic if the directory structure changes.

1.3.2 - Refactoring:

- **Separate PHP Logic from HTML:** Move the PHP logic to a separate file or the top of the document.
- **Use Prepared Statements:** This will prevent SQL injection.
- **Create Functions:** Define functions for repeated tasks like user validation.
- **Centralize Redirection Logic:** Create a function or mapping for redirection paths based on user types.

1.3.3 - Refactored Code:

login.php: -

```
<?php
// Start the session
session_start();
// Unset all the server-side variables
$_SESSION["user"] = "";
$_SESSION["usertype"] = "";
// Set the new timezone and date
date_default_timezone_set('Asia/Kolkata');
$_SESSION["date"] = date('Y-m-d');
// Import database connection
include("connection.php");
$error = '<label for="promter" class="form
label"> </label>';
if ($_SERVER['REQUEST_METHOD'] ==
'POST') {
    $email = $_POST['useremail'];
    $password = $_POST['userpassword'];

    // Function to check credentials
    function checkCredentials($database, $email,
$password, $table, $emailField, $passwordField) {
        $sql = "SELECT * FROM $table WHERE
$emailField = ? AND $passwordField = ?";
        $stmt = $database->prepare($sql);
        $stmt->bind_param("ss", $email, $password);
        $stmt->execute();
        return $stmt->get_result()->num_rows === 1;
    }

    // Get user type
    $stmt = $database->prepare("SELECT usertype
FROM webuser WHERE email = ?");
    $stmt->bind_param("s", $email);
    $stmt->execute();
    $result = $stmt->get_result();
    if ($result->num_rows === 1) {
        $utype = $result->fetch_assoc()['usertype'];
        $userTypes = ['p' => 'patient', 'a' => 'admin',
```

```
'd' => 'doctor'];
        $fields = ['pemail', 'aemail', 'docemail'];
        $passwordFields = ['ppassword', 'apassword',
'docpassword'];

        if (checkCredentials($database, $email,
$password, $userTypes[$utype], $fields[$utype],
$passwordFields[$utype])) {
            $_SESSION['user'] = $email;
            $_SESSION['usertype'] = $utype;
            header("Location: $utype/index.php");
            exit;
        } else {
            $error = '<label for="promter"
class="form-label" style="color:rgb(255, 62,
62);text-align:center;">Wrong credentials: Invalid
email or password</label>';
        }
    } else {
        $error = '<label for="promter" class="form
label" style="color:rgb(255, 62, 62);text
align:center;">We can\'t find any account for this
email.</label>';
    }
}
?>
```

Html:

```
<!DOCTYPE html>

<html lang="en">

        <head>

                <!-- Head details here -->

        </head>

        <body>
```

```
<!-- The rest of your HTML goes here. Use
<?php
echo $error ?> where you need to display
error
messages. -->
</body>
</html>
```

Notes:

i) The PHP logic is moved to the top of the document or a separate file.
ii) Prepared statements are used to avoid SQL injection.
iii) A function was created to handle credential checking, reducing redundancy.
iv) User type-specific redirection is handled more dynamically.
v) Direct $_POST access has been kept for simplicity but consider adding validation or sanitization as needed.

1.4 - Quality Assurance:

Following improvements are made due to these restructurings:

i) **Maintainability:** Improved maintainability is a result of the division of responsibilities and the development of reusable functionalities. The code is simple for developers to comprehend, alter, and expand upon without having an impact on other system components.
ii) **Readability:** The readability of the code has been much improved by reorganizing it, eliminating superfluous parts, and using descriptive function names. This facilitates developers' understanding of the application's logic and flow.
iii) **Scalability:** Scalability is facilitated by the usage of functions and code modularization. A more scalable and flexible system may be achieved by adding new features or making modifications with little effect on the current code.
iv) **Security:** The application's security is improved by addressing potential SQL injection issues and introducing prepared statements for database queries. This helps to improve defenses against harmful assaults.

v) **Reliability:** The system is more reliable when error handling techniques are added, particularly when it comes to database activities. The program becomes more stable and dependable because of the input users receive on the success or failure of their tasks.

vi) **Flexibility:** The code is more flexible when configuration files and constants for URLs and other data are used. The program becomes easier to modify in response to needs or changes in the environment.

vii) **Performance:** Better performance is indirectly benefited by the code rearrangement, even if it may not be expressly addressed. Code that is well-structured and efficient runs more smoothly, which enhances system performance.

viii) **Useability:** A program is made more user-friendly by separating issues and adding user input. By providing users with information regarding the results of their activities, the system's general usability is improved.

ix) **Collaboration:** The more efficient developer cooperation is achieved using comments in the documentation and the modular framework. Individual work on various modules by team members promotes cooperation and concurrent development.

x) **Adaptability:** The program is more flexible since configuration files are used and important logic is centralized. It is possible to execute configuration or business logic changes with little to no disturbance.