**Task 3:**

Code quality improvements, bug fixes, and best practices – code smells identification.

CustomerService.cs

### CustomerService.cs:

using System;

using System.Data.SqlClient;

namespace BlazorApp1.Services

{

public class CustomerService

{

private readonly string connectionString = "YourConnectionString";

public CustomerObject GetCustomerById(int customerId)

{

SqlConnection connection = new SqlConnection(connectionString);

try

{

connection.Open();

SqlCommand command = new SqlCommand("SELECT Name FROM Customers WHERE Id =" + customerId + ";");

var result = command.ExecuteScalar();

return new CustomerObject

{

CustomerId = customerId,

CustomerName = result.ToString()

};

}

catch (Exception ex)

{

Console.WriteLine("Error: " + ex.Message);

return null;

}

finally

{

connection.Close();

}

}

}

public class CustomerObject

{

public double CustomerId;

public string CustomerName;

}

}

### 

Issues:

1. **Using Statement**: Utilization of the using statement for SqlConnection and SqlCommand ensures proper disposal of resources.

using (\_connection){}

1. **Parameterized SQL inline Queries**: Instead of inline SQL, a stored procedure (sp\_GetCustomerById) is called. This promotes better maintainability and security.

**Inline code blocks** – red lines are vulnerable. We need to pass with value sql command parameters Add with value can help here.

1. **Async Method**: GetCustomerByIdAsync is now an asynchronous method for better responsiveness.

Time consuming task - await \_connection.OpenAsync(); -– multithreading concept

1. **SOLID Pinciples – Loosely coupled - CustomerObject Class for mocking**: Need to move to a separate project for better decoupling and testability.
2. **Properties Instead of Public Fields**: CustomerId and CustomerName Get set properties.
3. **Constructor Injection**: The connection string is passed via the constructor - dependency management. DI to resolve interface, class register and resolving their dependencies.
4. **Error Handling**: Logging with Serilog or a logger framework should be implemented instead of console logs.
5. **Naming Convention**: camelCase for variables and Pascalcase for class names. Need to follow the naming convention as per SOLID principles and Microsoft clean architecture model.

To ensure code coverage and quality with SonarCube and SonarLint is the best option for code optimization.

Customer.razor:

### Customer.razor

@page "/customer"

@using BlazorApp1.Services

@attribute [StreamRendering]

<h3>Customer Information</h3>

<label>Customer Id: @Customer.CustomerName</label>

<label>Customer Name: @Customer.CustomerId</label>

@code {

private CustomerObject Customer { get; set; }

protected override void OnInitialized()

{

var customerService = new CustomerService();

Customer = customerService.GetCustomerById(5);

}

}

1. Dependecy injection via constructor: Instead of creating a new instance of CustomerService inside the component, it's better to inject it using constructor injection. customerService should come from DI via constructor with private readonly field.
2. Write Unit Tests: Will be helpful for unit test cases mocking, if we make DI for all dependency classes and interfaces.
3. **Error Handling**: Implement error handling for cases where Customer is null. (For example - when the database query fails.)
4. **Async/Await Pattern**: Making OnInitialized async task method and using asynchronous methods for database operations will prevent blocking the UI thread. (acts as a multi-threading processor)
5. @Customer.CustomerName – if customer is null or customer model properties ar e null, it should be addressed. ? operatior to be used to avoid null checks.
6. Display the error to end user in a customized manner, instead of console logs.