**SQL EXERCISE - ADVANCED CONCEPTS**

**Exercise 1: Ranking and Window Functions**

Goal: Use ROW\_NUMBER(), RANK(), DENSE\_RANK(), OVER(), and PARTITION BY.

Scenario:

Find the top 3 most expensive products in each category using different ranking functions.

Steps:

1. Use ROW\_NUMBER() to assign a unique rank within each category.

2. Use RANK() and DENSE\_RANK() to compare how ties are handled.

3. Use PARTITION BY Category and ORDER BY Price DESC.

**SOLUTION:-**

-- 1. Create Database

IF DB\_ID('ProductDB') IS NULL

CREATE DATABASE ProductDB;

GO

-- 2. Use the database

USE ProductDB;

GO

-- 3. Create Products Table

IF OBJECT\_ID('Products', 'U') IS NOT NULL

DROP TABLE Products;

GO

CREATE TABLE Products (

ProductID INT PRIMARY KEY,

ProductName VARCHAR(100),

Category VARCHAR(50),

Price DECIMAL(10, 2)

);

GO

-- 4. Insert Sample Data

INSERT INTO Products (ProductID, ProductName, Category, Price) VALUES

(1, 'Phone A', 'Electronics', 1000),

(2, 'Phone B', 'Electronics', 900),

(3, 'Phone C', 'Electronics', 900),

(4, 'Phone D', 'Electronics', 850),

(5, 'Shirt A', 'Clothing', 500),

(6, 'Shirt B', 'Clothing', 450),

(7, 'Shirt C', 'Clothing', 500),

(8, 'Shirt D', 'Clothing', 400);

GO

-- 5. Top 3 by Category using ROW\_NUMBER()

PRINT '--- TOP 3 USING ROW\_NUMBER() ---';

SELECT \*

FROM (

SELECT \*,

ROW\_NUMBER() OVER (PARTITION BY Category ORDER BY Price DESC) AS RowNum

FROM Products

) AS RankedProducts

WHERE RowNum <= 3;

GO

-- 6. Top 3 by Category using RANK()

PRINT '--- TOP 3 USING RANK() ---';

SELECT \*

FROM (

SELECT \*,

RANK() OVER (PARTITION BY Category ORDER BY Price DESC) AS PriceRank

FROM Products

) AS RankedProducts

WHERE PriceRank <= 3;

GO

-- 7. Top 3 by Category using DENSE\_RANK()

PRINT '--- TOP 3 USING DENSE\_RANK() ---';

SELECT \*

FROM (

SELECT \*,

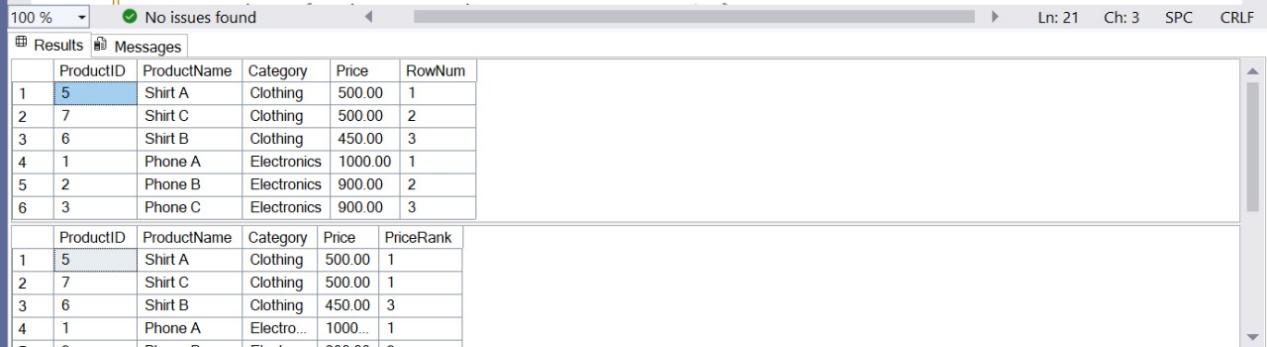
DENSE\_RANK() OVER (PARTITION BY Category ORDER BY Price DESC) AS DenseRank

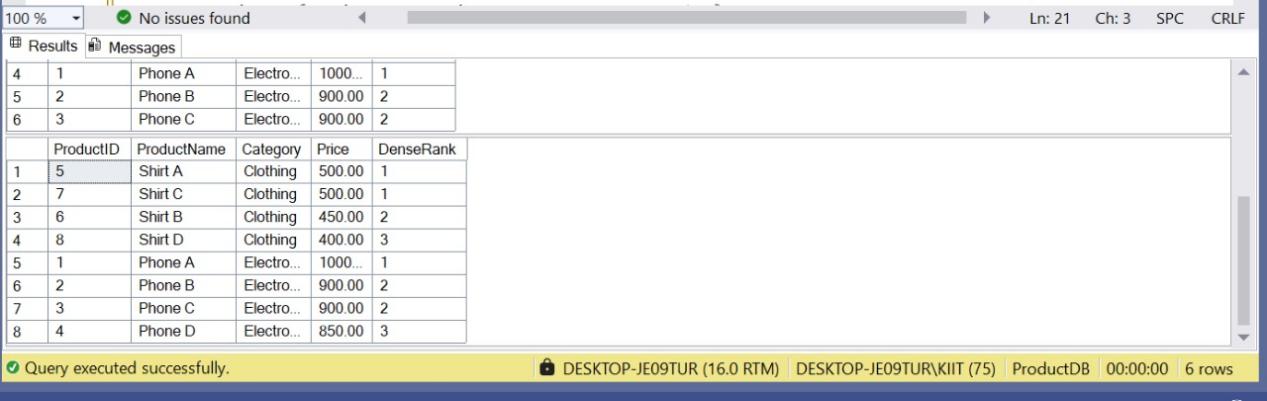
FROM Products

) AS RankedProducts

WHERE DenseRank <= 3;

GO





**SQL EXERCISE - STORED PROCEDURE**

**Exercise 1: Create a Stored Procedure**

Goal: Create a stored procedure to retrieve employee details by department.

Steps:

1. Define the stored procedure with a parameter for DepartmentID.

2. Write the SQL query to select employee details based on the DepartmentID.

3. Create a stored procedure named `sp\_InsertEmployee` with the following code:

CREATE PROCEDURE sp\_InsertEmployee

@FirstName VARCHAR(50),

@LastName VARCHAR(50),

@DepartmentID INT,

@Salary DECIMAL(10,2),

@JoinDate DATE

AS

BEGIN

INSERT INTO Employees (FirstName, LastName, DepartmentID, Salary, JoinDate)

VALUES (@FirstName, @LastName, @DepartmentID, @Salary, @JoinDate);

END;

**SOLUTION:-**

-- Step 1: Create and Use the Database

IF DB\_ID('EmployeeDB') IS NULL

CREATE DATABASE EmployeeDB;

GO

USE EmployeeDB;

GO

-- Step 2: Drop Tables If They Exist

IF OBJECT\_ID('Employees', 'U') IS NOT NULL DROP TABLE Employees;

IF OBJECT\_ID('Departments', 'U') IS NOT NULL DROP TABLE Departments;

GO

-- Step 3: Create Departments Table

CREATE TABLE Departments (

DepartmentID INT PRIMARY KEY,

DepartmentName VARCHAR(100)

);

-- Step 4: Create Employees Table (Fixed with IDENTITY)

CREATE TABLE Employees (

EmployeeID INT IDENTITY(1,1) PRIMARY KEY, -- Auto-increment EmployeeID

FirstName VARCHAR(50),

LastName VARCHAR(50),

DepartmentID INT FOREIGN KEY REFERENCES Departments(DepartmentID),

Salary DECIMAL(10,2),

JoinDate DATE

);

GO

-- Step 5: Insert Sample Data into Departments

INSERT INTO Departments (DepartmentID, DepartmentName) VALUES

(1, 'HR'),

(2, 'Finance'),

(3, 'IT'),

(4, 'Marketing');

GO

-- Step 6: Insert Sample Data into Employees (no need to include EmployeeID)

INSERT INTO Employees (FirstName, LastName, DepartmentID, Salary, JoinDate) VALUES

('John', 'Doe', 1, 5000.00, '2020-01-15'),

('Jane', 'Smith', 2, 6000.00, '2019-03-22'),

('Michael', 'Johnson', 3, 7000.00, '2018-07-30'),

('Emily', 'Davis', 4, 5500.00, '2021-11-05');

GO

-- Step 7: Create Stored Procedure to Insert a New Employee

IF OBJECT\_ID('sp\_InsertEmployee', 'P') IS NOT NULL

DROP PROCEDURE sp\_InsertEmployee;

GO

CREATE PROCEDURE sp\_InsertEmployee

@FirstName VARCHAR(50),

@LastName VARCHAR(50),

@DepartmentID INT,

@Salary DECIMAL(10,2),

@JoinDate DATE

AS

BEGIN

INSERT INTO Employees (FirstName, LastName, DepartmentID, Salary, JoinDate)

VALUES (@FirstName, @LastName, @DepartmentID, @Salary, @JoinDate);

END;

GO

-- Step 8: Create Stored Procedure to Retrieve Employees by Department

IF OBJECT\_ID('sp\_GetEmployeesByDepartment', 'P') IS NOT NULL

DROP PROCEDURE sp\_GetEmployeesByDepartment;

GO

CREATE PROCEDURE sp\_GetEmployeesByDepartment

@DepartmentID INT

AS

BEGIN

SELECT

EmployeeID,

FirstName,

LastName,

DepartmentID,

Salary,

JoinDate

FROM Employees

WHERE DepartmentID = @DepartmentID;

END;

GO

-- Step 9: Insert a new employee for testing

EXEC sp\_InsertEmployee

@FirstName = 'Alice',

@LastName = 'Brown',

@DepartmentID = 2,

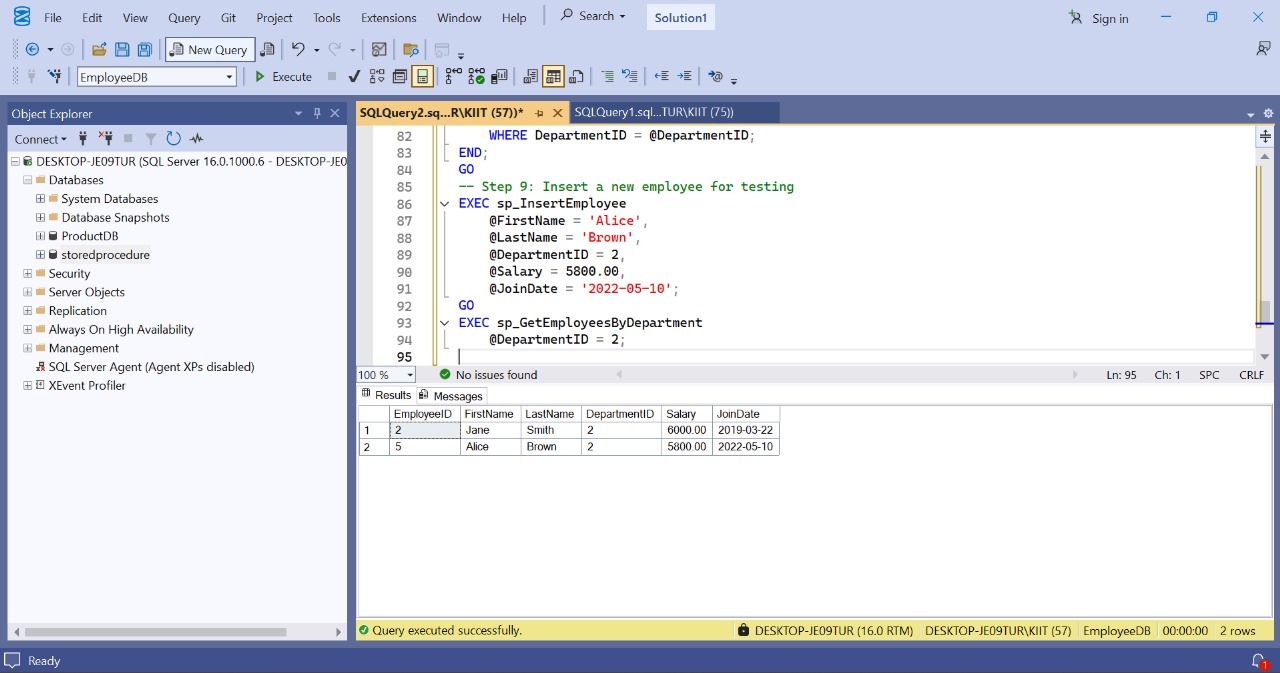
@Salary = 5800.00,

@JoinDate = '2022-05-10';

GO

EXEC sp\_GetEmployeesByDepartment

@DepartmentID = 2;



**Exercise 5: Return Data from a Stored Procedure**

Goal: Create a stored procedure that returns the total number of employees in a

department.

Steps:

1. Define the stored procedure with a parameter for DepartmentID.

2. Write the SQL query to count the number of employees in the specified department.

3. Save the stored procedure by executing the Stored procedure content

**SOLUTION:-**

-- Step 1: Create and Use the Database (if not already created)

IF DB\_ID('EmployeeDB') IS NULL

CREATE DATABASE EmployeeDB;

GO

USE EmployeeDB;

GO

-- Step 2: Drop the stored procedure if it already exists

IF OBJECT\_ID('sp\_CountEmployeesByDepartment', 'P') IS NOT NULL

DROP PROCEDURE sp\_CountEmployeesByDepartment;

GO

-- Step 3: Create the stored procedure

CREATE PROCEDURE sp\_CountEmployeesByDepartment

@DepartmentID INT

AS

BEGIN

SELECT COUNT(\*) AS EmployeeCount

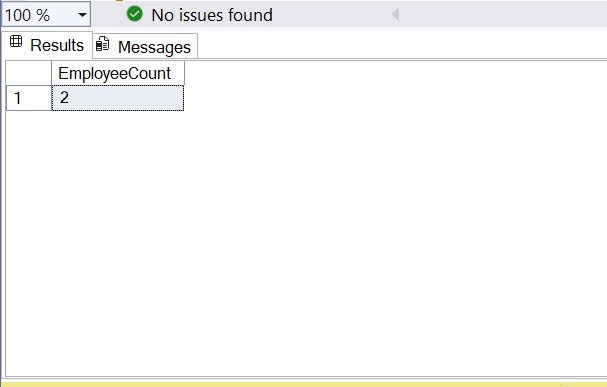
FROM Employees

WHERE DepartmentID = @DepartmentID;

END;

GO

EXEC sp\_CountEmployeesByDepartment @DepartmentID = 2;



**NUnit - Handson**

1. ****Explain the meaning of Unit Testing and its difference from Functional Testing****

****SOLUTION:-****

**Unit Testing**: Testing the smallest testable parts (units) of the application, such as individual methods or classes, in isolation.

Example: You tested the Add() method of your Calculator class.

**Functional Testing**: Tests the whole functionality or a feature of the system as a user would.

Example: Verifying that a button click performs addition correctly.

**Difference**:

| **Unit Testing** | **Functional Testing** |
| --- | --- |
| Developer-centric | QA/Tester-centric |
| Tests internal logic | Tests UI/API and integration |
| Fast execution | Slower due to more dependencies |

1. ****List Various Types of Testing****

****SOLUTION:-****

· Unit Testing

· Functional Testing

· Automated Testing

· Performance Testing

· (Optional: Integration, Regression, Smoke Testing)

1. ****Understand the Benefit of Automated Testing****

****SOLUTION:-****

· Fast feedback loop

· Easy to run frequently (CI/CD)

· Reduces regression bugs

· Saves time in long term

· Improves code confidence

1. ****Explain Loosely Coupled & Testable Design****

****SOLUTION:-****

* **Loosely Coupled**: Components are independent, changes in one do not break the other.

e.g., Injecting dependencies via constructors/interfaces.

* **Testable Design**: Easier to isolate logic and write tests.

e.g., CustomerComm(IMailSender) allows mock injection for testability.

### 5. ****Write Your First Testing Program to Validate a Calculator Addition Operation****

**SOLUTION:-**

* [TestFixture]: Marks the class containing test cases.
* [Test]: Marks the individual test method.

### ****Understand the Need of**** [SetUp]****,**** [TearDown]****,**** [Ignore] ****Attributes****

**SOLUTION:-**

| **Attribute** | **Purpose** |
| --- | --- |
| [SetUp] | Runs before each test (initialization) |
| [TearDown] | Runs after each test (cleanup) |
| [Ignore] | Temporarily skips a test |

### ****Explain the Benefit of Writing Parameterized Test Cases –**** [TestCase]

**SOLUTION:-**

* Avoids writing duplicate tests for similar logic
* Improves readability and maintenance

**Follow the steps listed below to write the NUnit test cases for the application.**

* **Create a Unit test project(.Net Framework) in the solution provided.**
* **Add the CalcLibrary project as reference**
* **Create a class “CalculatorTests” to write all the test cases for the methods in the solution**
* **Use the ‘TestFixture’, ‘SetUp’ and ‘TearDown’ attributes, to declare, initialize and cleanup activities respectively**
* **Create a Test method to check the addition functionality**
* **Use the ‘TestCase’ attribute to send the inputs and the expected result**
* **Use Assert.That to check the actual and expected result match**

**Step 1:** Create CalcLibrary1 Project

Go to File → New → Project.

Select Class Library (.NET Framework).

Name it: CalcLibrary1

Click Create

Delete Class1.cs.

Add a new class → Name it Calculator.cs → Add the following code:

namespace CalcLibrary1

{

public class Calculator

{

public int Add(int a, int b)

{

return a + b;

}

}

}

**Step 2:** Create a Unit Test Project

Right-click on the Solution → Add → New Project

Select Unit Test Project (.NET Framework)

Name it: CalcLibrary1.Tests

Click Create

**Step 3:** Add Reference to CalcLibrary1

Right-click CalcLibrary1.Tests → Add → Reference

Go to the Projects tab → Check the box for CalcLibrary1

Click OK

**Step 4:** Install NUnit Packages in Test Project

Right-click CalcLibrary1.Tests → Manage NuGet Packages

Install these packages:

NUnit

NUnit3TestAdapter

Microsoft.NET.Test.Sdk

**Step 5:** Create Test Class

Add a new class to CalcLibrary1.Tests → Name it CalculatorTests.cs

Add the following code:

csharp

Copy

Edit

using NUnit.Framework;

using CalcLibrary1;

namespace CalcLibrary1.Tests

{

[TestFixture]

public class CalculatorTests

{

private Calculator calculator;

[SetUp]

public void SetUp()

{

calculator = new Calculator();

}

[TearDown]

public void TearDown()

{

calculator = null;

}

[Test]

[TestCase(2, 3, 5)]

[TestCase(-1, 1, 0)]

[TestCase(0, 0, 0)]

public void Add\_WhenCalled\_ReturnsExpectedSum(int a, int b, int expected)

{

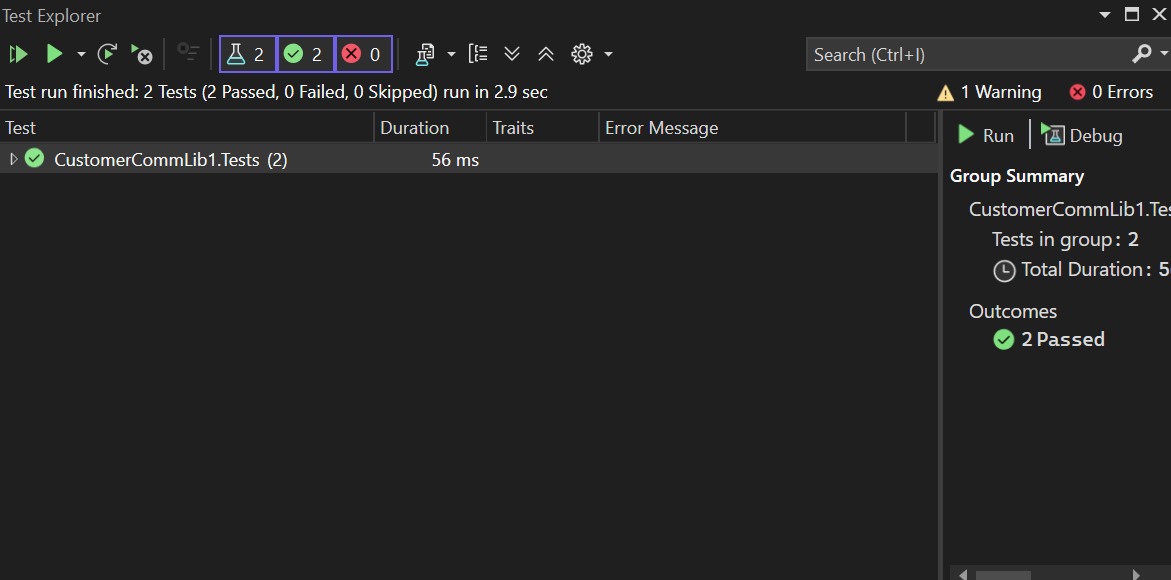
var result = calculator.Add(a, b);

Assert.That(result, Is.EqualTo(expected));

}

}

}

****

**Moq - Handson**

1. **Write Testable Code with Moq**

## **Scenario**

You are tasked to write a unit test code for the below scenario.

The application in which you are teamed up with, deals with a mail server communication in which your application tries to send mail to its users upon every transaction. Your role is to write unit testing the module that contains send mail functionality. You wanted to perform testing the module without sending any email.

After investigating the problem scenario, you found a solution and that is creating **mock** objects of these external dependencies in the unit testing project so that you can achieve speedier test execution and loose coupling of code.

**SOLUTION:-**

Step 1: Add Interface and Classes to CustomerCommLib1

a. Add a new file: IMailSender.cs

Right-click CustomerCommLib1 → Add → Class → Name it IMailSender.cs, then paste:

namespace CustomerCommLib1

{

public interface IMailSender

{

bool SendMail(string toAddress, string message);

}

}

b. Add a new file: MailSender.cs

Right-click → Add → Class → Name it MailSender.cs, then paste:

using System.Net;

using System.Net.Mail;

namespace CustomerCommLib1

{

public class MailSender : IMailSender

{

public bool SendMail(string toAddress, string message)

{

MailMessage mail = new MailMessage();

SmtpClient smtpServer = new SmtpClient("smtp.gmail.com");

mail.From = new MailAddress("your\_email@gmail.com");

mail.To.Add(toAddress);

mail.Subject = "Test Mail";

mail.Body = message;

smtpServer.Port = 587;

smtpServer.Credentials = new NetworkCredential("username", "password");

smtpServer.EnableSsl = true;

smtpServer.Send(mail);

return true;

}

}

}

c. Add a new file: CustomerComm.cs

Right-click → Add → Class → Name it CustomerComm.cs, then paste:

namespace CustomerCommLib1

{

public class CustomerComm

{

private readonly IMailSender \_mailSender;

public CustomerComm(IMailSender mailSender)

{

\_mailSender = mailSender;

}

public bool SendMailToCustomer()

{

\_mailSender.SendMail("cust123@abc.com", "Some Message");

return true;

}

}

}

Step 2: Create Unit Test Project (CustomerCommLib1.Tests)

a. Add new project

Right-click solution → Add → New Project

Select Unit Test Project (.NET Framework) → Name it: CustomerCommLib1.Tests

b. Add references

Right-click CustomerCommLib1.Tests → Add → Reference

Check ✅ CustomerCommLib1 under Projects and click OK

c. Add NuGet packages

Right-click CustomerCommLib1.Tests → Manage NuGet Packages

Install the following into the test project:

✅ NUnit

✅ NUnit3TestAdapter

✅ Moq

d. Add Test Class: CustomerCommTests.cs

Right-click CustomerCommLib1.Tests → Add → Class → Name it CustomerCommTests.cs, then paste:

using NUnit.Framework;

using Moq;

using CustomerCommLib1;

namespace CustomerCommLib1.Tests

{

[TestFixture]

public class CustomerCommTests

{

private Mock<IMailSender> \_mockMailSender;

private CustomerComm \_customerComm;

[OneTimeSetUp]

public void Init()

{

\_mockMailSender = new Mock<IMailSender>();

\_mockMailSender

.Setup(m => m.SendMail(It.IsAny<string>(), It.IsAny<string>()))

.Returns(true);

\_customerComm = new CustomerComm(\_mockMailSender.Object);

}

[Test]

public void SendMailToCustomer\_ShouldReturnTrue()

{

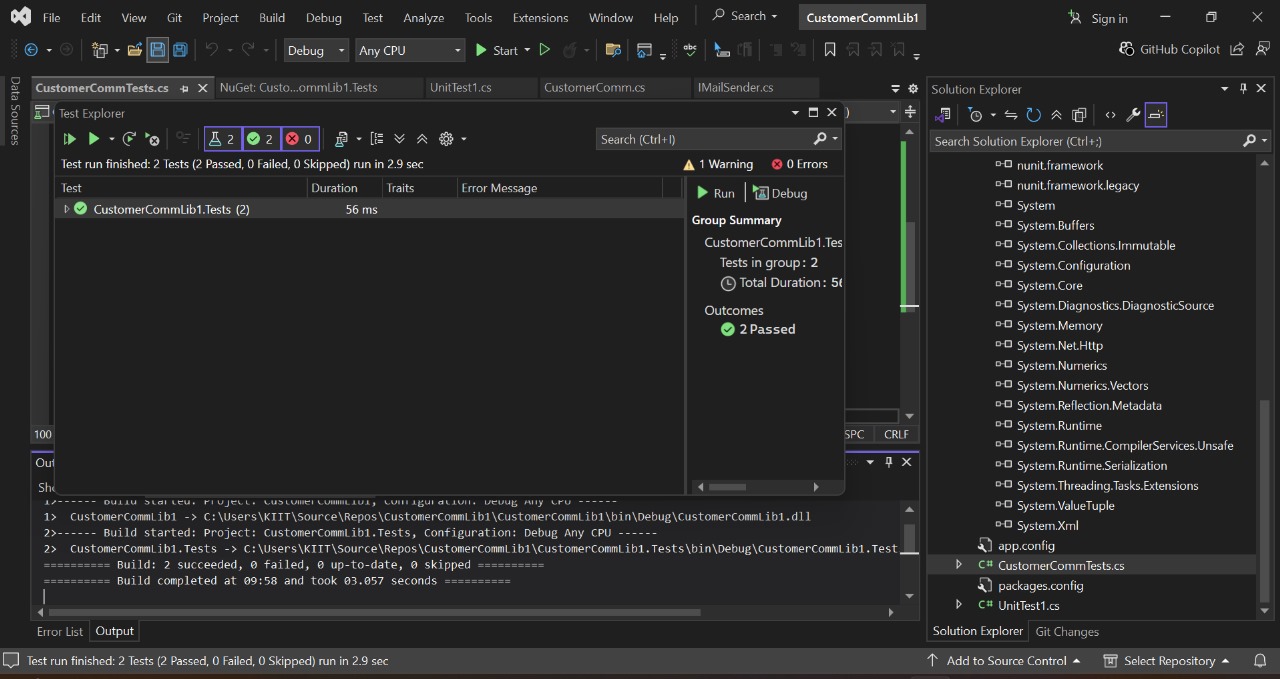
var result = \_customerComm.SendMailToCustomer();

Assert.That(result, Is.True);

}

}

}

****