**Week 2: Advanced SQL, Nunit and Moq**

**SQL Exercises: Advanced SQL Concepts**

**Exercise 1: RANKING AND WINDOW FUNCTIONS**

**QUERY:**

WITH RankedProducts AS (

SELECT

ProductID,

ProductName,

Category,

Price,

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

RANK() OVER (PARTITION BY Category ORDER BY Price DESC) AS RankNum,

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

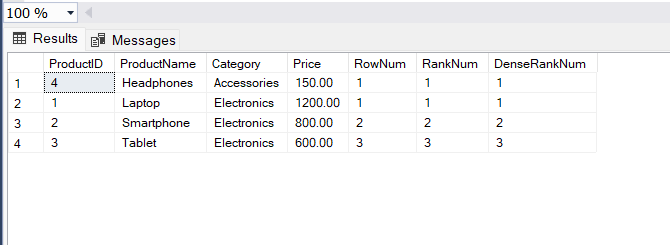
from

Products

)

select \* from RankedProducts where RowNum <= 3;

**Output:**

****

**SQL Exercise: Stored Procedure**

**Exercise 1: Create a Stored Procedure**

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

Create PROCEDURE sp\_GetEmployeesByDepartment

@DepartmentID INT

AS

BEGIN

select \* from Employees

where DepartmentID = @DepartmentID;

END;

**2. Select employee based on DepartmentID**

SELECT \* FROM Employees WHERE DepartmentID = 3;

**Output:**



**3. Stored procedure named ‘sp\_InsertEmployee’**

EXEC sp\_InsertEmployee

@FirstName = 'Sachin',

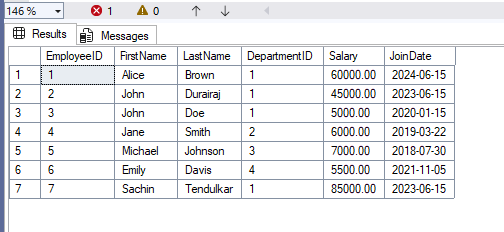
@LastName = 'Tendulkar',

@DepartmentID = 1,

@Salary = 85000.00,

@JoinDate = '2023-06-15';

**Output:**

****

**SQL Exercises: Functions**

**Exercise 7: Return data from Scalar function**

**1.Execute the `fn\_CalculateAnnualSalary` function for an employee with `EmployeeID = 1`.**

create FUNCTION dbo.fn\_CalculateAnnualSalary (@EmpID INT)

returns DECIMAL(12,2)

AS

BEGIN

DECLARE @Annual DECIMAL(12,2);

select @Annual = Salary \* 12

from Employees where EmployeeID = @EmpID;

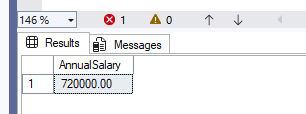
RETURN @Annual;

END;

GO

SELECT dbo.fn\_CalculateAnnualSalary(1) AS AnnualSalary;

**Output:**

****

**2. Verify the result**

SELECT e.EmployeeID,

e.FirstName,

e.LastName,

d.DepartmentName,

e.Salary AS MonthlySalary,

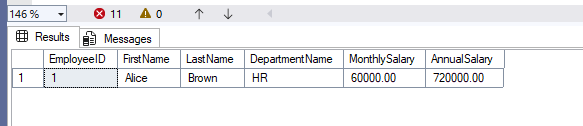
dbo.fn\_CalculateAnnualSalary(e.EmployeeID) AS AnnualSalary

FROM Employees e

JOIN Departments d ON d.DepartmentID = e.DepartmentID

WHERE e.EmployeeID = 1;

**Output:**



**SQL Exercise: Stored Procedure**

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

**1.Stored Procedure with DepartmentdID**

create PROCEDURE sp\_GetEmployeeCountByDepartment

@DepartmentID INT

AS

BEGIN

select COUNT(\*) AS EmployeeCount from Employees

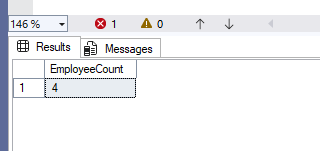
where DepartmentID = @DepartmentID;

END;

**2.Query for count the number of employees in the department**

EXEC sp\_GetEmployeeCountByDepartment @DepartmentID = 1;

**Output:**

****

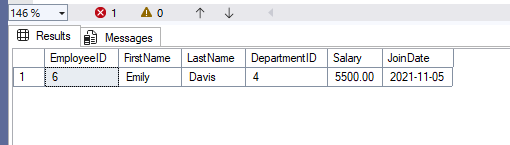
**SQL Exercise: Stored Procedure**

**Exercise 4: Execute a stored procedure**

**1.To Execute the stored procedure**

EXEC sp\_GetEmployeesByDepartment @DepartmentID = 4;

**Output:**

****

**SQL Exercise: Index**

**-- Exercise 1: Creating a Non-Clustered Index**

-- Goal: Create a non-clustered index on the ProductName column in the Products table and compare query execution time before and after index creation.

**-- Step 1: Query to fetch product details before index creation**

SELECT \* FROM Products WHERE ProductName = 'Laptop';

**-- Step 2: Create a non-clustered index on ProductName**

CREATE NONCLUSTERED INDEX idx\_ProductName

ON Products (ProductName);

**-- Step 3: Query to fetch product details after index creation**

SELECT \* FROM Products WHERE ProductName = 'Laptop';

**Output:**

A screenshot of a computer

AI-generated content may be incorrect.

**-- Exercise 2: Creating a Clustered Index**

-- Goal: Create a clustered index on the OrderDate column in the Orders table and compare query execution time before and after index creation.

**-- Step 1: Query to fetch orders before index creation**

SELECT \* FROM Orders WHERE OrderDate = '2023-01-15';

**-- Step 2: Create a clustered index on OrderDate**

CREATE CLUSTERED INDEX idx\_OrderDate

ON Orders (OrderDate);

**-- Step 3: Query to fetch orders after index creation**

SELECT \* FROM Orders WHERE OrderDate = '2023-01-15';

**Output:**

A screenshot of a computer

AI-generated content may be incorrect.

**-- Exercise 3: Creating a Composite Index**

-- Goal: Create a composite index on the CustomerID and OrderDate columns in the Orders table and compare query execution time before and after index creation.

**-- Step 1: Query to fetch orders before index creation**

SELECT \* FROM Orders WHERE CustomerID = 1 AND OrderDate = '2023-01-15';

**-- Step 2: Create a composite index on CustomerID and OrderDate**

CREATE NONCLUSTERED INDEX idx\_CustomerID\_OrderDate

ON Orders (CustomerID, OrderDate);

**-- Step 3: Query to fetch orders after index creation**

SELECT \* FROM Orders WHERE CustomerID = 1 AND OrderDate = '2023-01-15';

**Output:**

**A screenshot of a computer

AI-generated content may be incorrect.**

1.NUnit-Handson

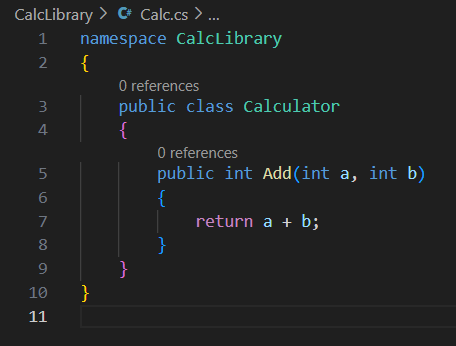
**Given:**

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

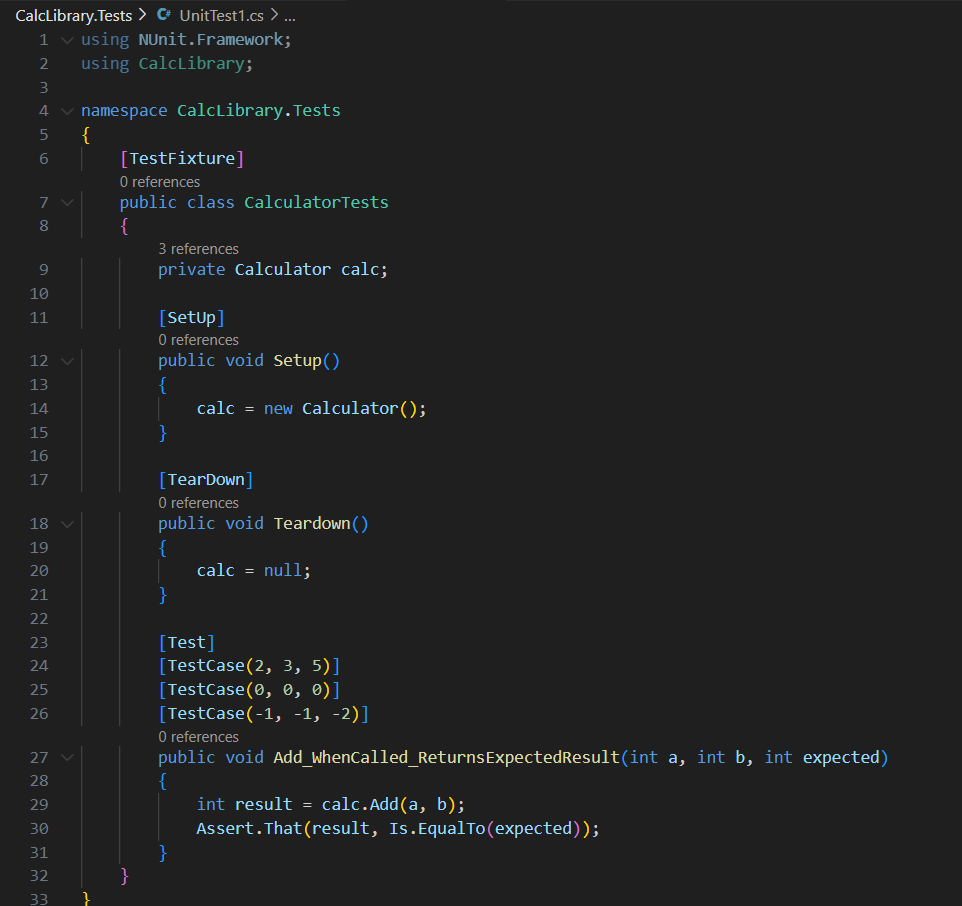
* Create a Unit test project(.Net Framework) in the solution provided.
* Add the CalcLibrary project as reference
* Create a class “CalculatorTests” to write all the test cases for the methods in the solution
* Use the ‘TestFixture’, ‘SetUp’ and ‘TearDown’ attributes, to declare, initialize and cleanup 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

**Code Implementation:**

**Calc.cs**

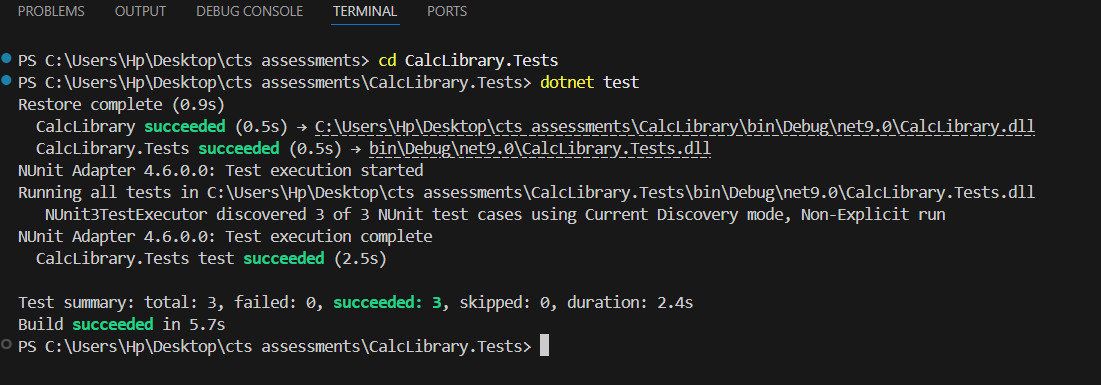
****

**CalculatorTests.cs**



* Used [TestFixture] to declare the test class for NUnit.
* Used [SetUp] to initialize the Calculator object before each test.
* Used [TearDown] to clean up the object after each test (optional but good practice).
* Tested the Add() method with different inputs using [TestCase].
* Used Assert.That to compare actual and expected results.
* Ensured the method works correctly for positive, negative, and zero inputs.

**Output:**



1. 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.

**Task 1:**

* Created a **Class Library** named CustomerCommLib.
* Defined an **interface IMailSender** with a SendMail() method.
* Implemented the interface in a class called **MailSender** using SmtpClient to send real emails.
* Created a class **CustomerComm** that **injects IMailSender** through its constructor.
* The method SendMailToCustomer() calls the injected SendMail() method.
* This structure makes the code **testable** by allowing **mocking** of the SendMail() method during unit testing.

## Task2 :

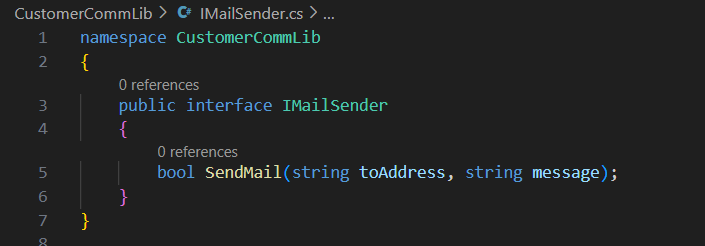
In this task, you will create unit test project which make use of NUnit framework and Moq.

* Create a new class library project called **CustomerComm.Tests** and add the following external dependencies to it using **NuGet Package Manager.**
  + NUnit
  + NUnit Test Adapter
  + Moq
* Add the references of assemblies as appropriate including **CustomerCommLib.**
* Write unit test code and **mock** the **MailSender (IMailSender)** class.
* Use **TestFixture**, **OneTimeSetUp** and **TestCase** attribute classes on top of test class, init method and test method respectively.
* **Configure** the mock object in such away that **SendMail()** method will accept any two string arguments and always return true when **SendMailToCustomer()** gets invoked.
* Finally **assert** the return value to “true”.

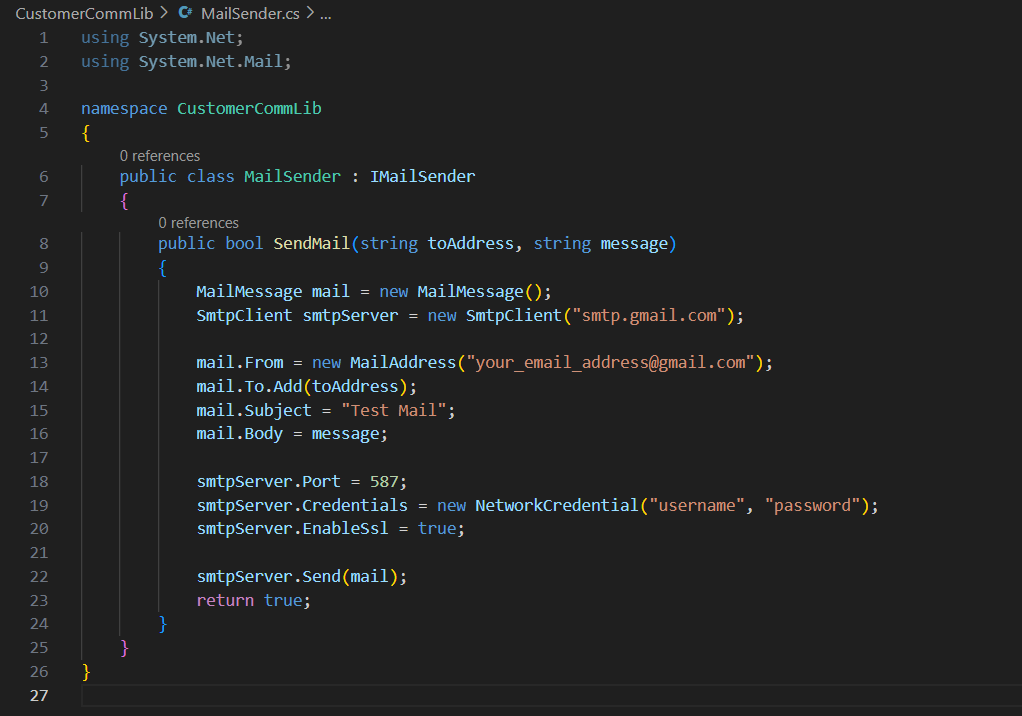
**Code Implementation:**

1. Write Code in CustomerCommLib

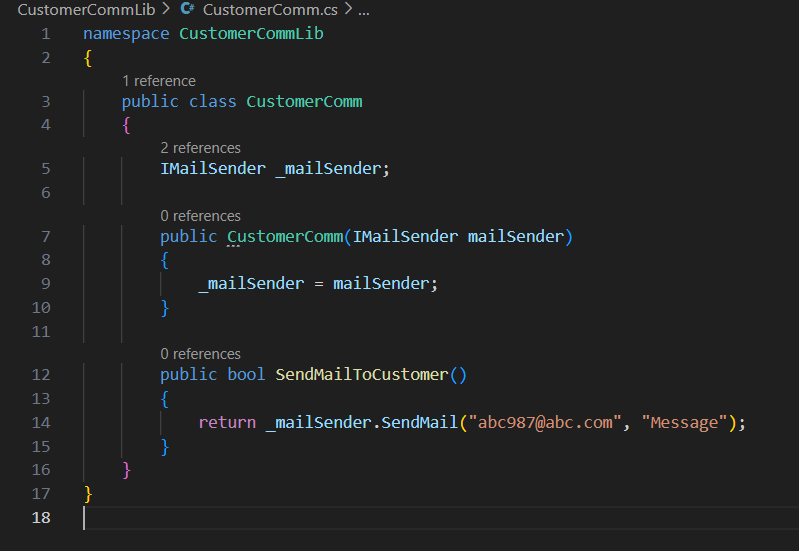
IMailSender.cs



MailSender.cs

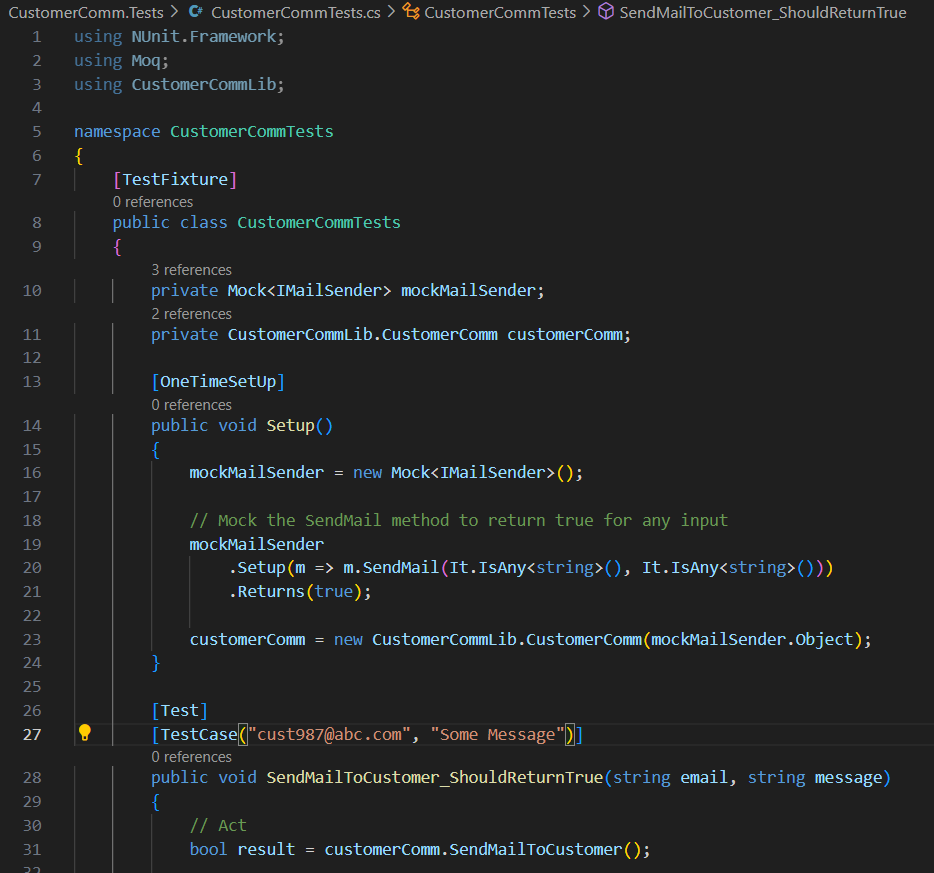


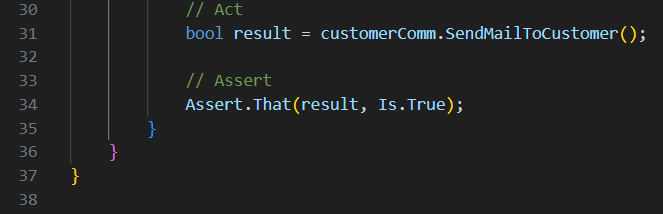
CustomerComm.cs



1. Write Test in CustomerComm.Tests

CustomerCommTests.cs





Output:

