SE Lab 9 – Software Testing - Contents

Part 1: Unit Testing with C# .NET Core in Visual Studio

Introduction

Unit testing is a software testing technique where individual components or functions of a program are tested in isolation to ensure they work as expected. The goal is to validate that each unit of the software performs correctly.

1. Why Unit Testing?

- Early Bug Detection: Helps identify and fix bugs early in the development process.
- Code Quality: Encourages better design and more maintainable code.
- Documentation: Provides documentation on how the code is supposed to work.
- Refactoring Support: Makes it easier to refactor code with confidence that existing functionality is not broken.

2. What is Unit Testing?

Unit testing is a software testing technique where individual components or functions of a program are tested in isolation to ensure they work as expected. The goal is to validate that each unit of the software performs correctly.

2.1. Equivalence Partitioning (EP)

Equivalence partitioning involves dividing input data into partitions where test cases from each partition are expected to behave similarly. For example:

- Positive Numbers: Testing with positive values.
- Negative Numbers: Testing with negative values.
- Zero: Testing with zero.

2.2. Boundary Value Analysis (BVA)

Boundary value analysis involves testing at the boundaries of input ranges. For example:

- Upper Boundary: Testing with the maximum value.
- Lower Boundary: Testing with the minimum value.

3. Steps to Create a .NET Core Project and Test Project

3.1. Create a .NET Core Project

1. Open Visual Studio.
2. Go to File > New > Project.
3. Select ASP.NET Core Web Application or Console App (.NET Core), and click Next.
4. Name your project (e.g., CalculatorApp) and click Create.
5. Choose the appropriate template (e.g., Console Application) and click Create.

3.2. Create a Test Project

1. Right-click on the solution in Solution Explorer.
2. Select Add > New Project.
3. Choose xUnit Test Project (.NET Core) and click Next.
4. Name your test project (e.g., CalculatorApp.Tests) and click Create.

3.3. Add Project Reference

1. Right-click on the CalculatorApp.Tests project.
2. Select Add > Reference.
3. Check the box next to CalculatorApp and click OK.

Exercise 1: BasicMath Class

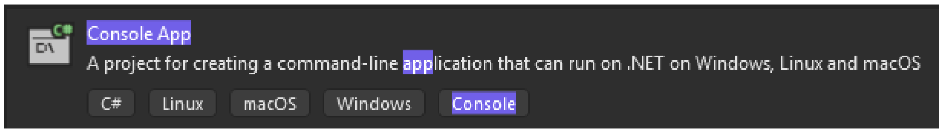
Steps to Create Unit Tests with C# .NET Core in Visual Studio

If no Invisual Studio then Install Visual Studio**:**

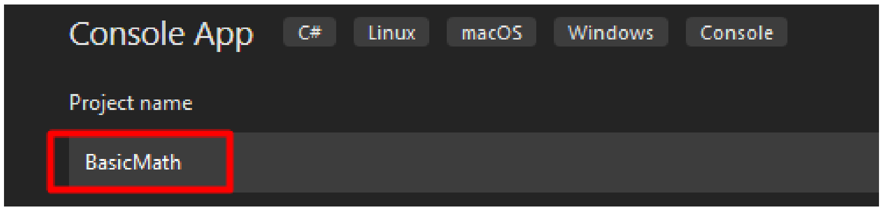
- Download and install Visual Studio Community edition from the official Visual Studio website.
- During installation, ensure you select the .NET Desktop Development workload.

1. Create a Console Project

- Open Visual Studio.
- Select Create a new project.
- **Choose Console App (.NET Core) and click Next.**


Console App
A project for creating a command-line application that can run on .NET on Windows, Linux and macOS

- **Name the project BasicMath and click Create.**



Project name: BasicMath

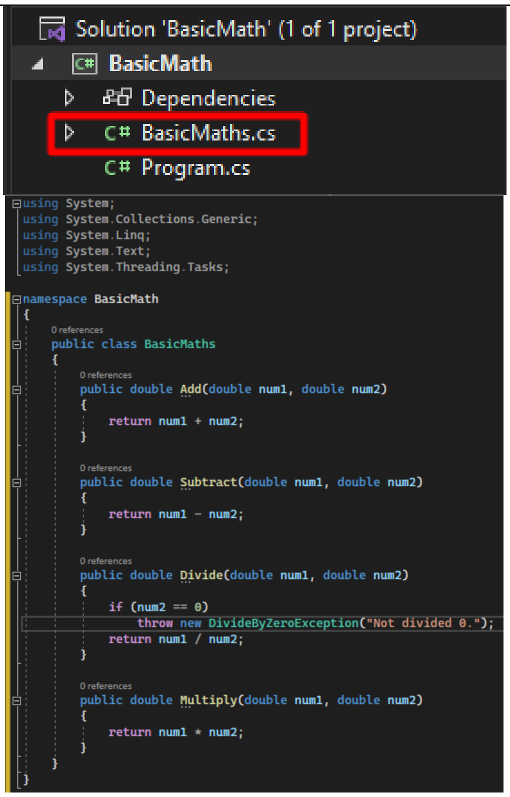
2. **Define the BasicMaths Class:**
   - In the Solution Explorer, right-click on the BasicMath project and select Add > Class.
   - Name the class BasicMaths.cs and click Add.
   - Replace the content with the following code:

```
namespace BasicMath
{
   public class BasicMaths
   {
      public double Add(double num1, double num2)
      {
         return num1 + num2;
      }

      public double Subtract(double num1, double num2)
      {
         return num1 - num2;
      }

      public double Divide(double num1, double num2)
      {
         if (num2 == 0)
            throw new DivideByZeroException("Division by
zero is not allowed.");
         return num1 / num2;
      }

      public double Multiply(double num1, double num2)
      {
         return num1 * num2;
      }
   }
}
```



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace BasicMath
{
    0 references
    public class BasicMaths
    {
        0 references
        public double Add(double num1, double num2)
        {
            return num1 + num2;
        }

        0 references
        public double Subtract(double num1, double num2)
        {
            return num1 - num2;
        }

        0 references
        public double Divide(double num1, double num2)
        {
            if (num2 == 0)
                throw new DivideByZeroException("Not divided 0.");
            return num1 / num2;
        }

        0 references
        public double Multiply(double num1, double num2)
        {
            return num1 * num2;
        }
    }
}
```

3. **Create a Unit Test Project**
   - In the Solution Explorer, right-click on the solution and select Add > New Project.

- Choose Unit Test Project (.NET Core) and click Next.
- Name the project BasicMathTests and click Create.
- Add a reference to the BasicMath project:
    - Right-click on the BasicMathTests project, select Add > Project Reference.
    - Check the BasicMath project and click OK.

4. Write Unit Tests

- **In the BasicMathTests project, open the BasicMathsTests.cs file.**
- **Replace the content with the following code to define test methods using [DataTestMethod]:**

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using BasicMath;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace BasicMathTests
{
   [TestClass]
   public class BasicMathsTests
   {
      [DataTestMethod]
      [DataRow(1, 1, 2)] // EP: Positive numbers
      [DataRow(-1, -1, -2)] // EP: Negative numbers
      [DataRow(0, 0, 0)] // EP: Zero
      [DataRow(int.MaxValue, 1, (double)int.MaxValue + 1)] // BVA: Upper boundary
      [DataRow(int.MinValue, -1, (double)int.MinValue - 1)] // BVA: Lower boundary
      public void Test_AddMV(int a, int b, double expected)
      {
         BasicMaths bm = new BasicMaths();
         double actual = bm.Add(a, b);
         Assert.AreEqual(expected, actual);
      }

      // Similar tests can be written for Subtract, Divide, and Multiply methods








   }
}
```

Another methods with Xunit:

You can create CalculatorTests.cs and copy code or copy this code to BasicMathsTests.cs

```
using Xunit;
using CalculatorApp;

namespace BasicMathTests
{
    public class CalculatorTests
    {
        // Test method for Add operation using inline data
        [Theory]
        [InlineData(1, 2, 3)] // Test case 1: 1 + 2 = 3
        [InlineData(-1, -2, -3)] // Test case 2: -1 + -2 = -3
        [InlineData(0, 0, 0)] // Test case 3: 0 + 0 = 0
        [InlineData(int.MaxValue, 1, int.MinValue)] // Test case 4: Overflow scenario
        public void Add_MultipleValues_ReturnsCorrectSum(int a, int b, int expected)
        {
            // Arrange: Create an instance of Calculator
            var calculator = new Calculator();
            // Act: Call the Add method
            int result = calculator.Add(a, b);
            // Assert: Verify the result
            Assert.Equal(expected, result);
        }
        // Test method for Subtract operation using inline data
        [Theory]
        [InlineData(5, 3, 2)] // Test case 1: 5 - 3 = 2
        [InlineData(-5, -3, -2)] // Test case 2: -5 - -3 = -2
        [InlineData(0, 0, 0)] // Test case 3: 0 - 0 = 0
        [InlineData(int.MinValue, 1, int.MinValue - 1)] // Test case 4: Underflow
scenario
        public void Subtract_MultipleValues_ReturnsCorrectDifference(int a, int b,
int expected)
        {
            // Arrange: Create an instance of Calculator
            var calculator = new Calculator();
            // Act: Call the Subtract method
            int result = calculator.Subtract(a, b);
            // Assert: Verify the result
            Assert.Equal(expected, result);
        }
        // Test method for Multiply operation using inline data
        [Theory]
        [InlineData(2, 3, 6)] // Test case 1: 2 * 3 = 6
        [InlineData(-2, -3, 6)] // Test case 2: -2 * -3 = 6
        [InlineData(0, 5, 0)] // Test case 3: 0 * 5 = 0
        [InlineData(int.MaxValue, 1, int.MaxValue)] // Test case 4: Max value
scenario
```

```
        public void Multiply_MultipleValues_ReturnsCorrectProduct(int a, int b, int
expected)
        {
            // Arrange: Create an instance of Calculator
            var calculator = new Calculator();
            // Act: Call the Multiply method
            int result = calculator.Multiply(a, b);
            // Assert: Verify the result
            Assert.Equal(expected, result);
        }
        // Test method for Divide operation using inline data
        [Theory]
        [InlineData(6, 3, 2)] // Test case 1: 6 / 3 = 2
        [InlineData(-6, -3, 2)] // Test case 2: -6 / -3 = 2
        [InlineData(0, 1, 0)] // Test case 3: 0 / 1 = 0
        [InlineData(int.MaxValue, 1, int.MaxValue)] // Test case 4: Max value
scenario
        public void Divide_MultipleValues_ReturnsCorrectQuotient(int a, int b, double
expected)
        {
            // Arrange: Create an instance of Calculator
            var calculator = new Calculator();
            // Act: Call the Divide method
            double result = calculator.Divide(a, b);
            // Assert: Verify the result
            Assert.Equal(expected, result);
        }
    }
}
```

5. **Run the Tests**
    - **Open the Test Explorer from the Test menu.**
    - **Click Run All to execute the tests.**

Exercise 2: Multiple Values and Data Source

Data sources

Data sources for unit tests can be from various formats such as:

- CSV Files: As shown in the example above.
- Excel Files: Using libraries like ExcelDataReader.
- Database Servers: Connecting to a database and fetching test data.

Using Data Source from .csv. Ensure you have the following CSV files in your test project directory:

add_testdata.csv

```
A,B,Expected
1,2,3
-1,-2,-3
0,0,0
2147483647,1,-2147483648
```

subtract_testdata.csv

```
A,B,Expected
5,3,2
-5,-3,-2
0,0,0
-2147483648,1,-2147483649
```

multiply_testdata.csv

```
A,B,Expected
2,3,6
-2,-3,6
0,5,0
2147483647,1,2147483647
```

divide_testdata.csv

```
A,B,Expected
6,3,2
-6,-3,2
0,1,0
2147483647,1,2147483647
```

Implementation

- Right-click on the BasicMath.Tests project in Solution Explorer.
- Select Add > New Item.
- Choose Class and name it CalculatorCsvTests.cs. Click Add

```
using CsvHelper;
using System.Collections.Generic;
using System.Globalization;
using System.IO;
using Xunit;
using CalculatorApp;

namespace BasicMathTests
{
    public class CalculatorCsvTests
    {
        // Method to read test data from a CSV file
        public static IEnumerable<object[]> GetTestData(string fileName)
        {
            using (var reader = new StreamReader(fileName))
            using (var csv = new CsvReader(reader, CultureInfo.InvariantCulture))
            {
                foreach (var record in csv.GetRecords<TestData>())
                {
                    yield return new object[] { record.A, record.B, record.Expected };
                }
            }
        }

        // Test method for Add operation using CSV data
        [Theory]
        [MemberData(nameof(GetTestData), parameters: "add_testdata.csv")]
        public void Add_CsvData_ReturnsCorrectSum(int a, int b, int expected)
        {
            // Arrange: Create an instance of Calculator
            var calculator = new Calculator();
            // Act: Call the Add method
            int result = calculator.Add(a, b);
            // Assert: Verify the result
            Assert.Equal(expected, result);
        }

        // Test method for Subtract operation using CSV data
        [Theory]
        [MemberData(nameof(GetTestData), parameters: "subtract_testdata.csv")]
        public void Subtract_CsvData_ReturnsCorrectDifference(int a, int b, int expected)
        {
            // Arrange: Create an instance of Calculator
            var calculator = new Calculator();
            // Act: Call the Subtract method
            int result = calculator.Subtract(a, b);
            // Assert: Verify the result
            Assert.Equal(expected, result);
        }
        // Test method for Multiply operation using CSV data
        [Theory]
        [MemberData(nameof(GetTestData), parameters: "multiply_testdata.csv")]
        public void Multiply_CsvData_ReturnsCorrectProduct(int a, int b, int expected)
        {
            // Arrange: Create an instance of Calculator
            var calculator = new Calculator();
            // Act: Call the Multiply method
            int result = calculator.Multiply(a, b);
            // Assert: Verify the result
            Assert.Equal(expected, result);
        }

        // Test method for Divide operation using CSV data
        [Theory]
        [MemberData(nameof(GetTestData), parameters: "divide_testdata.csv")]
        public void Divide_CsvData_ReturnsCorrectQuotient(int a, int b, double expected)
        {
            // Arrange: Create an instance of Calculator
            var calculator = new Calculator();
            // Act: Call the Divide method
            double result = calculator.Divide(a, b);
            // Assert: Verify the result
            Assert.Equal(expected, result);
        }

        // Class to represent the test data structure
        public class TestData
        {
            public int A { get; set; }
            public int B { get; set; }
            public double Expected { get; set; }
        }
    }
}
```

Run the Tests

- Open the Test Explorer from the Test menu.
- Click Run All to execute the tests.

Exercise 3: Bank Account

BankAccount Class

Scenario with Multiple Values and .csv Files

1. **Create a .csv file named BankAccountTestData.csv with the following content:**

```
customerName, initialBalance, debitAmount, expectedBalance

Trung Pham,1000,200,800

Ky Pham,500,100,400

Thai Pham,300,50,250

Ky Trung Pham,100,150,Insufficient funds
```

2. **Requirements:**

- Create a Console Project named BankAccountApp and define the BankAccount class with methods Debit, Credit, and Withdraw.
- Create a Unit Test Project named BankAccountTests and write test methods for each operation using [DataTestMethod] and [DataSource].
- Define Test Cases using Equivalence Partitioning (EP) and Boundary Value Analysis (BVA) for each method.
- Run the Tests and ensure all methods pass the test cases.

```
//BankAccount.cs
namespace BankAccountApp
{
    public class BankAccount
    {
        public string CustomerName { get; private set; }
        public decimal Balance { get; private set; }

        public BankAccount(string customerName, decimal initialBalance)
        {
            CustomerName = customerName;
            Balance = initialBalance;
        }
        public void Debit(decimal amount)
        {
            if (amount > Balance)
            {
                throw new InvalidOperationException("Insufficient funds");
            }
            Balance -= amount;
        }
        public void Credit(decimal amount)
```

```
            {
                Balance += amount;
            }
            public void Withdraw(decimal amount)
            {
                Debit(amount);
            }
        }
    }
```

```
//BankAccountTests.cs
    using Xunit;
    using BankApp;
    using System;

    namespace BankAccountTests
    {
        public class BankAccountTests
        {
            [Theory]
            [InlineData("John Doe", 1000, 200, 800)]
            [InlineData("Jane Smith", 500, 100, 400)]
            [InlineData("Alice Johnson", 300, 50, 250)]
            public void Debit_ValidAmount_UpdatesBalance(string customerName, decimal
    initialBalance, decimal debitAmount, decimal expectedBalance)
            {
                // Arrange
                var account = new BankAccount(customerName, initialBalance);

                // Act
                account.Debit(debitAmount);

                // Assert
                Assert.Equal(expectedBalance, account.Balance);
            }

            [Theory]
            [InlineData("Bob Brown", 100, 150)]
            public void Debit_InsufficientFunds_ThrowsException(string customerName,
    decimal initialBalance, decimal debitAmount)
            {
                // Arrange
                var account = new BankAccount(customerName, initialBalance);

                // Act & Assert
                Assert.Throws<InvalidOperationException>(() =>
        account.Debit(debitAmount));
            }

            [Theory]
            [InlineData("John Doe", 1000, 200, 1200)]
            [InlineData("Jane Smith", 500, 100, 600)]
            [InlineData("Alice Johnson", 300, 50, 350)]
            public void Credit_ValidAmount_UpdatesBalance(string customerName, decimal
        initialBalance, decimal creditAmount, decimal expectedBalance)
            {
                // Arrange
                var account = new BankAccount(customerName, initialBalance);

                // Act
                account.Credit(creditAmount);

                // Assert
                Assert.Equal(expectedBalance, account.Balance);
            }

            [Theory]
            [InlineData("John Doe", 1000, 200, 800)]
            [InlineData("Jane Smith", 500, 100, 400)]
            [InlineData("Alice Johnson", 300, 50, 250)]
            public void Withdraw_ValidAmount_UpdatesBalance(string customerName, decimal
        initialBalance, decimal withdrawAmount, decimal expectedBalance)
            {
                // Arrange
                var account = new BankAccount(customerName, initialBalance);

                // Act
                account.Withdraw(withdrawAmount);

                // Assert
                Assert.Equal(expectedBalance, account.Balance);
            }
        }
    }
```

```
//BankAccountCsvTests.cs
    using CsvHelper;
    using System.Collections.Generic;
    using System.Globalization;
    using System.IO;
    using Xunit;
    using BankApp;

    namespace BankAccountTests
    {
        public class BankAccountCsvTests
        {
            public static IEnumerable<object[]> GetTestData()
            {
                using (var reader = new StreamReader("BankAccountTestData.csv"))
                using (var csv = new CsvReader(reader, CultureInfo.InvariantCulture))
                {
                    foreach (var record in csv.GetRecords<TestData>())
                    {
                        yield return new object[] { record.CustomerName,
    record.InitialBalance, record.DebitAmount, record.ExpectedBalance };
                    }
                }
            }

            [Theory]
            [MemberData(nameof(GetTestData))]
            public void Debit_CsvData_UpdatesBalance(string customerName, decimal
    initialBalance, decimal debitAmount, string expectedBalance)
            {
                // Arrange
                var account = new BankAccount(customerName, initialBalance);

                // Act & Assert
                if (expectedBalance == "Insufficient funds")
                {
                    Assert.Throws<InvalidOperationException>(() =>
    account.Debit(debitAmount));
                }
                else
                {
                    account.Debit(debitAmount);
                    Assert.Equal(decimal.Parse(expectedBalance), account.Balance);
                }
            }

            public class TestData
            {
                public string CustomerName { get; set; }
                public decimal InitialBalance { get; set; }
                public decimal DebitAmount { get; set; }
                public string ExpectedBalance { get; set; }
            }
        }
    }
```

Part 2: Automation Testing with Selenium and C# .NET Core in Visual Studio

Introduction

Automation testing involves using software tools to execute pre-scripted tests on a software application before it is released into production. This helps ensure the application behaves as expected and meets the requirements. Automation testing is faster, more reliable, and repeatable compared to manual testing.

Why Automation Testing?

- Efficiency: Automates repetitive tasks, saving time and effort.
- Consistency: Ensures tests are executed in the same manner every time.
- Coverage: Allows for extensive testing across different browsers and devices.
- Speed: Accelerates the testing process, enabling faster feedback.

Key Frameworks and Tools

1. Selenium.WebDriver

Selenium WebDriver is a popular open-source tool for automating web browsers. It allows you to write scripts in various programming languages (e.g., C#, Java, Python) to interact with web elements and perform actions like clicking buttons, entering text, and navigating between pages.

Key Features:

- Supports multiple browsers (Chrome, Firefox, Edge, etc.).
- Can be integrated with various testing frameworks.
- Allows for cross-browser testing.
- Example:

```
using OpenQA.Selenium;
using OpenQA.Selenium.Chrome;

class SeleniumExample
{
  static void Main()
  {
    IWebDriver driver = new ChromeDriver();
    driver.Navigate().GoToUrl("https://www.example.com");
    driver.FindElement(By.Id("username")).SendKeys("user1");
```

```
            driver.FindElement(By.Id("password")).SendKeys("password1");
            driver.FindElement(By.Id("loginButton")).Click();
            driver.Quit();
        }
    }
```

2. NUnit

NUnit is a widely-used open-source unit testing framework for .NET languages. It is used to write and run tests, and it provides various attributes and assertions to facilitate testing.

Key Features:

- Supports data-driven tests using attributes like [TestCase] and [TestCaseSource].
- Provides setup and teardown methods for initializing and cleaning up test environments.
- Integrates well with other tools like Selenium.
- **Example:**

```
using NUnit.Framework;

[TestFixture]
public class CalculatorTests
{
    [Test]
    public void Add_TwoNumbers_ReturnsSum()
    {
        var calculator = new Calculator();
        int result = calculator.Add(2, 3);
        Assert.AreEqual(5, result);
    }
}
```

3. NUnit3TestAdapter

NUnit3TestAdapter is a Visual Studio extension that allows NUnit tests to be discovered, executed, and reported within the Visual Studio Test Explorer. It bridges the gap between NUnit and Visual Studio, making it easier to run and manage tests.

Key Features:

- Integrates NUnit tests with Visual Studio Test Explorer.
- Supports running tests from the command line using dotnet test.

4. Microsoft.NET.Test.Sdk

Microsoft.NET.Test.Sdk is a set of libraries and tools that enable running tests in .NET Core and .NET Framework projects. It provides the necessary infrastructure to discover and execute tests, and it integrates with various test frameworks like NUnit, xUnit, and MSTest.

Key Features:

- Supports multiple test frameworks.
- Provides test discovery and execution capabilities.
- Integrates with CI/CD pipelines.

5. Setting Up an Automation Test Project

1. **Create a New Project:**
   - **Open Visual Studio.**
   - **Create a new Class Library (.NET Core) project.**
2. **Add NuGet Packages:**
   - **Selenium.WebDriver**
   - **Selenium.WebDriver.ChromeDriver**
   - **NUnit**
   - **NUnit3TestAdapter**
   - **Microsoft.NET.Test.Sdk**
   - **CsvHelper (if needed for data-driven tests)**
3. **Write Test Scripts:**
   - **Use Selenium WebDriver to interact with web elements.**
   - **Use NUnit to write and organize test cases.**
   - **Use NUnit3TestAdapter to run tests in Visual Studio.**
4. **Run Tests:**
   - **Use Visual Studio Test Explorer to run and manage tests.**
   - **Generate test reports using tools like Allure or built-in Visual Studio reporting.**

Automation Test with Selenium

As automation testing involves using software tools to execute tests automatically, reducing the need for manual testing. Selenium is a popular tool for automating web application tests.

Overall Steps for this exercise:

1. Set Up Environment: Ensure all necessary tools and libraries are installed.
2. Create Login Page: A simple HTML page with username and password fields.
3. Create CSV File: Contains test data for user credentials.
4. Create Automated Test Project: A .NET Core project with Selenium WebDriver and CsvHelper.
5. Write Automated Test:
   - GetTestData method reads data from the CSV file.
   - Login_WithValidCredentials_ShouldSucceed method uses Selenium WebDriver to automate the login process.

Running the Tests

1. Build the solution to ensure there are no errors.
2. Run the tests using the Test Explorer in Visual Studio.

Exercise 4: Create an Automated Test for a Login Page

1. Set Up Your Environment

Ensure you have the following installed:

- Visual Studio
- .NET Core SDK
- Selenium WebDriver
- CsvHelper library

2. Create the Login Page (HTML)

- Create a simple HTML login page (login.html) with fields for username and password, and a login button.

```
<!DOCTYPE html>
<html>
<head>
    <title>Login Page</title>
</head>
<body>
    <form id="loginForm">
        <label for="username">Username:</label>
        <input type="text" id="username" name="username"><br><br>
        <label for="password">Password:</label>
        <input type="password" id="password" name="password"><br><br>
        <button type="submit" id="loginButton">Login</button>
    </form>
</body>
</html>
```

- **Note: If you already have a login page created with PHP and MySQL, you can run your page with XAMPP on a browser. Once the page is running, copy the URL from the browser's address bar to use in your Selenium WebDriver script (**`driver.Navigate(YourURL).GoToUrl)`

3. Create the CSV File

Create a CSV file named LoginTestData.csv with the following content:

```
username,password
user1,password1
```

```
user2,password2
user3,password3
```

4. Create the Automated Test Project

1. Open Visual Studio.
2. Create a new solution and add a Class Library (.NET Core) project named LoginAutomation.
3. Add the necessary NuGet packages:
    - Selenium.WebDriver
    - Selenium.WebDriver.ChromeDriver
    - CsvHelper

5. Write the Automated Test

LoginTests.cs

This file will contain the automated test using Selenium WebDriver:

```
using OpenQA.Selenium;
using OpenQA.Selenium.Chrome;
using CsvHelper;
using System.Collections.Generic;
using System.Globalization;
using System.IO;
using Xunit;

namespace LoginAutomation
{
    public class LoginTests
    {
        public static IEnumerable<object[]> GetTestData()
        {
            using (var reader = new StreamReader("LoginTestData.csv"))
            using (var csv = new CsvReader(reader, CultureInfo.InvariantCulture))
            {
                foreach (var record in csv.GetRecords<TestData>())
                {
                    yield return new object[] { record.Username, record.Password };
                }
            }
        }

        [Theory]
        [MemberData(nameof(GetTestData))]
        public void Login_WithValidCredentials_ShouldSucceed(string username, string
password)
        {
            // Initialize the ChromeDriver
            using (IWebDriver driver = new ChromeDriver())
            {
                // Navigate to the login page
                driver.Navigate().GoToUrl("file:///path/to/your/login.html");

                // Find the username and password fields and fill them
                driver.FindElement(By.Id("username")).SendKeys(username);
                driver.FindElement(By.Id("password")).SendKeys(password);

                // Click the login button
                driver.FindElement(By.Id("loginButton")).Click();

                // Add assertions as needed, e.g., check for a successful login message
                // For this example, we'll just wait for a few seconds to simulate the
login process
                System.Threading.Thread.Sleep(2000);
            }
        }

        public class TestData
        {
            public string Username { get; set; }
            public string Password { get; set; }
        }
    }
}
```

6. Running the Tests

1. Build the solution to ensure there are no errors.
2. Run the tests using the Test Explorer in Visual Studio.

7. Creating a Test Report

To create a test report after running your automation tests, you can use Allure reporting:

1. Add Allure NuGet Packages

- Open your test project in your development environment (e.g., Visual Studio).
- Add the following NuGet packages to your project via the NuGet Package Manager or the Package Manager Console:

        - Allure.Commons: Provides core Allure functionality.
        - Allure.NUnit: Integrates Allure with NUnit test framework.

2. Configure Allure in Your Test Project

- After installing the packages, configure Allure within your project. This typically involves:
    - Ensuring your app.config or appsettings.json file includes the necessary Allure settings.
    - Setting up directories where Allure will save results (commonly allure-results folder).

3. Annotate Test Methods with Allure Attributes

- Decorate your test classes and methods with Allure attributes to provide additional metadata:
    - [AllureSuite("Suite Name")]: Groups related test methods into a suite.
    - [AllureSubSuite("Sub-Suite Name")]: Further categorizes tests.
    - [AllureSeverity(AllureSeverityLevel.Critical)]: Indicates the priority or importance of the test.
    - [AllureFeature("Feature Name")]: Links the test to a particular feature.
    - [AllureStory("Story Name")]: Connects the test to a user story or requirement.
    - [AllureTag("Tag Name")]: Tags the test with keywords for easier filtering.
    - You can also attach additional context like screenshots, logs, or parameters using Allure methods.

4. Run Tests and Generate the Allure Report

- Run your test suite as you normally would using NUnit.
- After running the tests, a folder (e.g., allure-results) will be populated with test execution data in JSON format.
- Use Allure's command-line interface (CLI) to generate and view the report:
    1. Install the Allure CLI by following the instructions from Allure's official documentation.
    2. Open your terminal, navigate to the directory containing the allure-results folder, and execute:
    3. allure serve

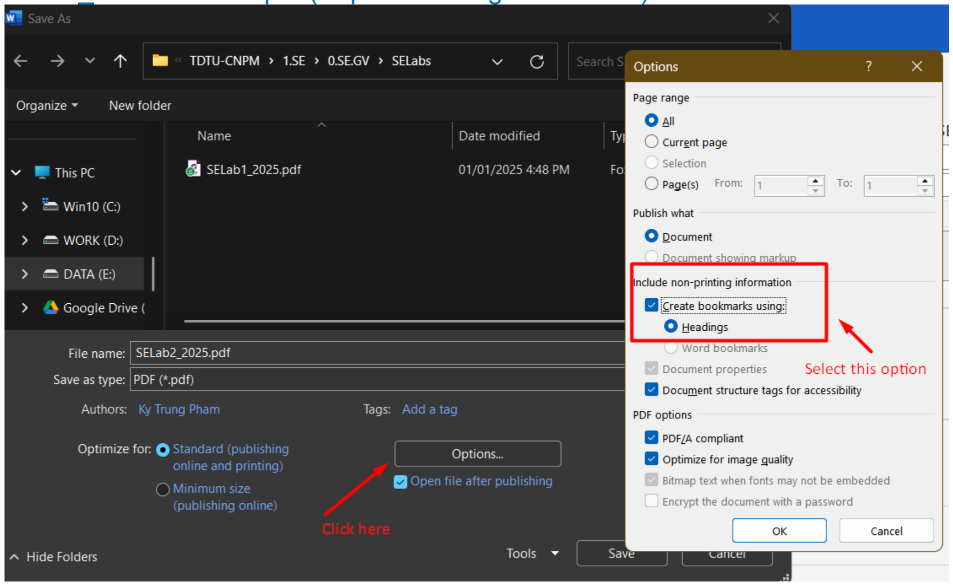
This will generate the report and launch it in your default web browser for viewing. Allure reports provide a detailed overview of test results, including passed/failed/skipped tests, trends, and test case metadata. It's an excellent tool for visualizing and analyzing test outcomes.

Submit on eLearning: Summary Report

- **Introduction**: Briefly describe the purpose of the exercises.
- **Exercises**: List each exercise with a brief description of what you did and learned.
- **Screenshots**: Include the screenshots of every step for each exercise and steps (both Visual Studio and Github) (Index each step with sequential number) and the output when press F5.
- **Conclusion**: Summarize your overall learning experience and any challenges you faced.

Submit 2 files on eLearning:

- StudentID_StudentName.zip (contain Windows Web Project & .sql file) for both Class Activity and Homework.
- StudentID_StudentName.pdf (require creating bookmarks) includes GitHub link as Appendix



SELab1_2025.pdf 01/01/2025 4:48 PM

File name: SELab2_2025.pdf
Save as type: PDF (*.pdf)
Authors: Ky Trung Pham
Tags: Add a tag
Optimize for: [✓ Standard (publishing online and printing)] Minimum size (publishing online)
[✓ Open file after publishing]

Options
Page range: [✓ All] Current page Selection Page(s) From: 1 To: 1
Publish what: [✓ Document] Document showing markup
Include non-printing information
[✓ Create bookmarks using:] [✓ Headings] Word bookmarks
[✓ Document properties]
[✓ Document structure tags for accessibility]
PDF options
[✓ PDF/A compliant]
[✓ Optimize for image quality]
[✓ Bitmap text when fonts may not be embedded]
Encrypt the document with a password

Notes: There may be a typo/mistake during creating this document. Please correct it by yourself.

References:

[Selenium with C# : How to start running Automated Tests | BrowserStack]