

---

# **Welcome to the Step-By-Step Getting Started Guide!**

**Feb 08, 2022**



<b>1</b>	<b>Install Visual Studio extension</b>	<b>1</b>
<b>2</b>	<b>Create calculator project</b>	<b>5</b>
<b>3</b>	<b>Create SpecFlow project</b>	<b>11</b>
<b>4</b>	<b>Create SpecFlow project - Continue</b>	<b>17</b>
<b>5</b>	<b>Bind the first step</b>	<b>21</b>
<b>6</b>	<b>Bind remaining steps</b>	<b>27</b>
<b>7</b>	<b>Fix implementation</b>	<b>33</b>
<b>8</b>	<b>Add Living Documentation</b>	<b>37</b>
<b>9</b>	<b>Final</b>	<b>43</b>
<b>10</b>	<b>Install JetBrains Rider Plugin</b>	<b>45</b>
<b>11</b>	<b>Create calculator project</b>	<b>49</b>
<b>12</b>	<b>Create SpecFlow project</b>	<b>55</b>
<b>13</b>	<b>Create SpecFlow project - Continue</b>	<b>59</b>
<b>14</b>	<b>Bind the first step</b>	<b>63</b>
<b>15</b>	<b>Bind remaining steps</b>	<b>69</b>
<b>16</b>	<b>Fix implementation</b>	<b>75</b>
<b>17</b>	<b>Add Living Documentation</b>	<b>79</b>
<b>18</b>	<b>Final</b>	<b>83</b>
<b>19</b>	<b>Exercise</b>	<b>85</b>
<b>20</b>	<b>Exercise-solution</b>	<b>87</b>



# CHAPTER 1

---

## Install Visual Studio extension

---

10 minutes

In this step you'll learn how to install the Visual Studio extension for SpecFlow.

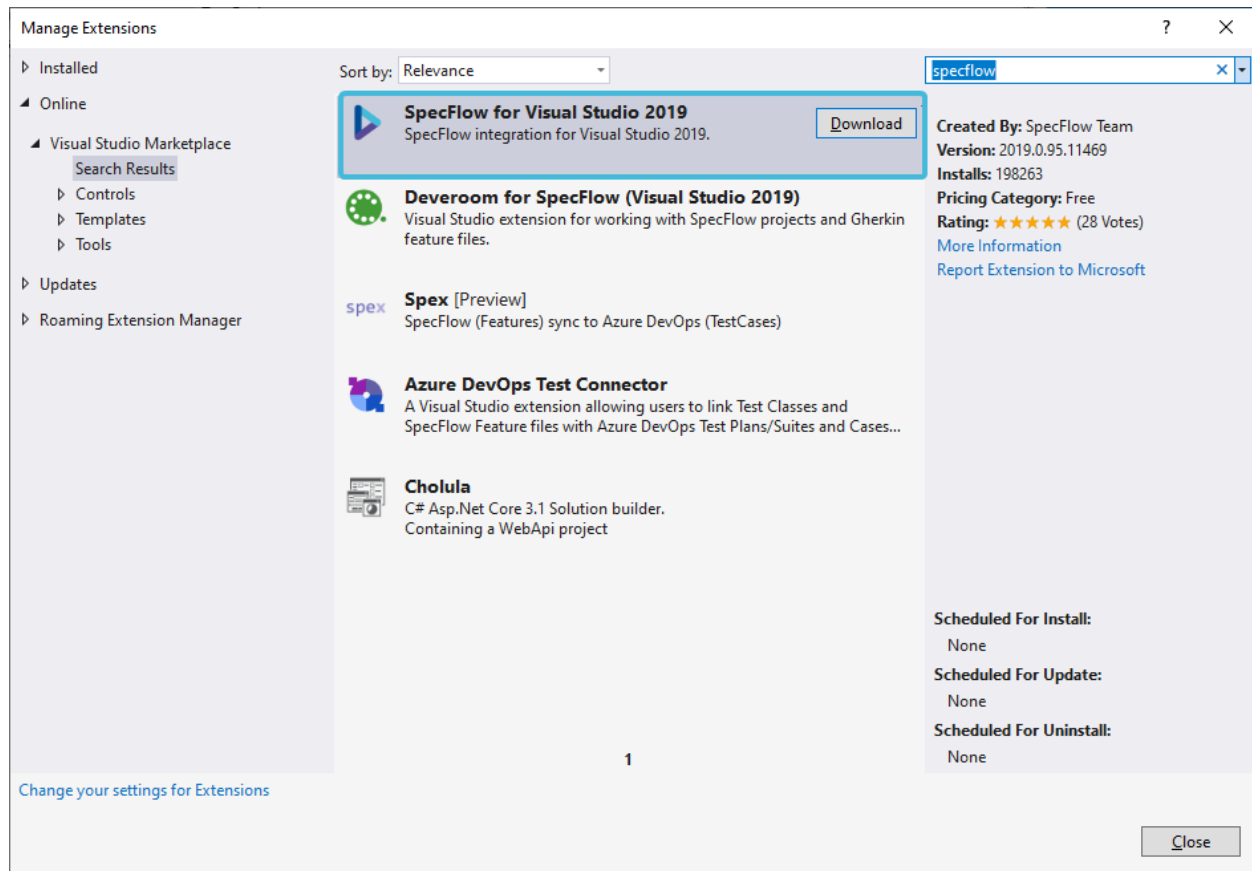
SpecFlow's Visual Studio extension not only enables the functionalities needed for testing automation, but is also bundled with several helpful features to make the journey more intuitive.

SpecFlow's Visual Studio extension works on Visual Studio 2017 & 2019. If you are using an older version of Visual Studio, please upgrade to the latest version.

Installation of the extension is simple:

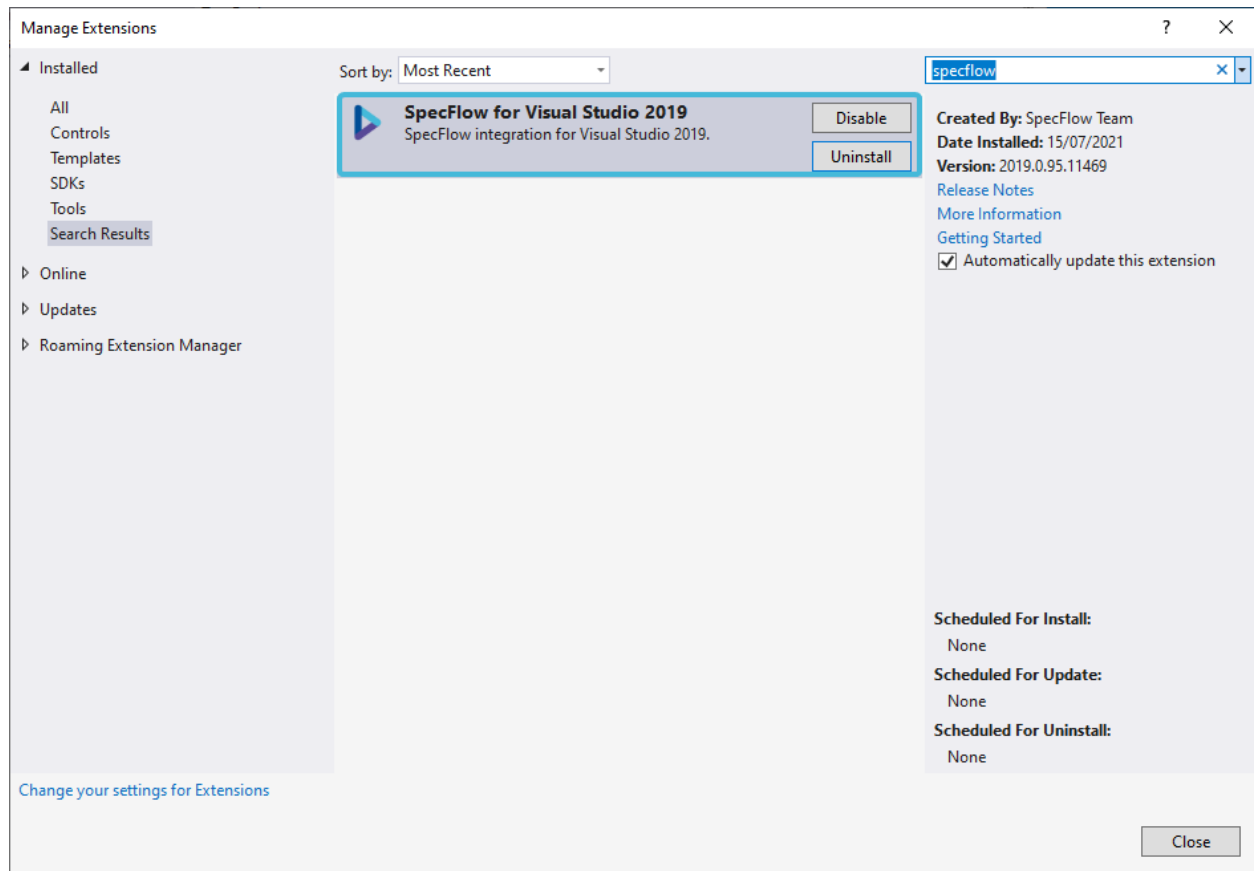
- 1- Open Visual Studio. *\*We use Visual Studio 2019 in this guide*
- 2- Navigate to “Extensions Manage Extensions Online “ and search for “SpecFlow” in the search bar.
- 3- Hit **Download** to begin the installation. You will need to restart Visual Studio for the installation to complete:

## Welcome to the Step-By-Step Getting Started Guide!



Once the extension is successfully installed, you can see it in the list of “Installed” extensions in the “Extensions Manage Extensions” dialog of Visual Studio.

## Welcome to the Step-By-Step Getting Started Guide!



In the next steps you'll create a simple application that will be used throughout this guide.





## CHAPTER 2

---

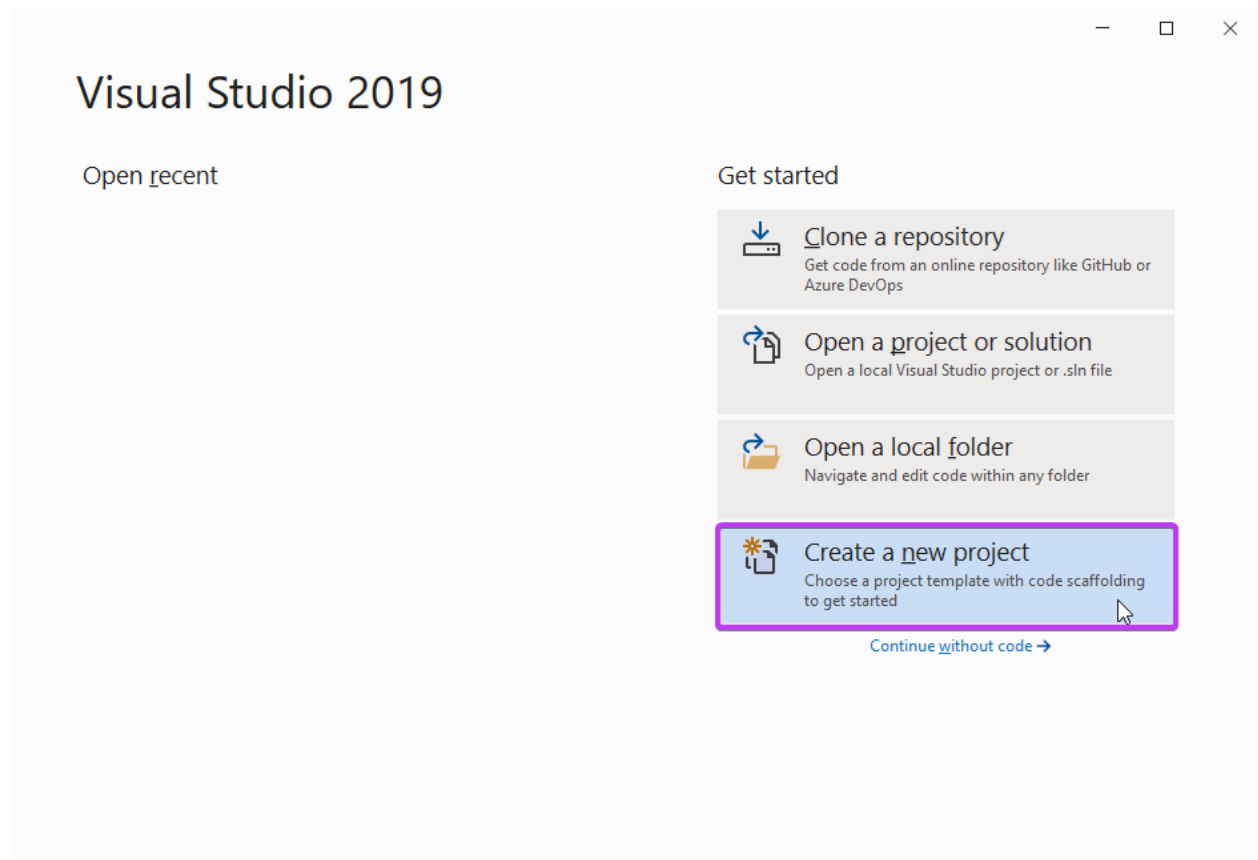
### Create calculator project

---

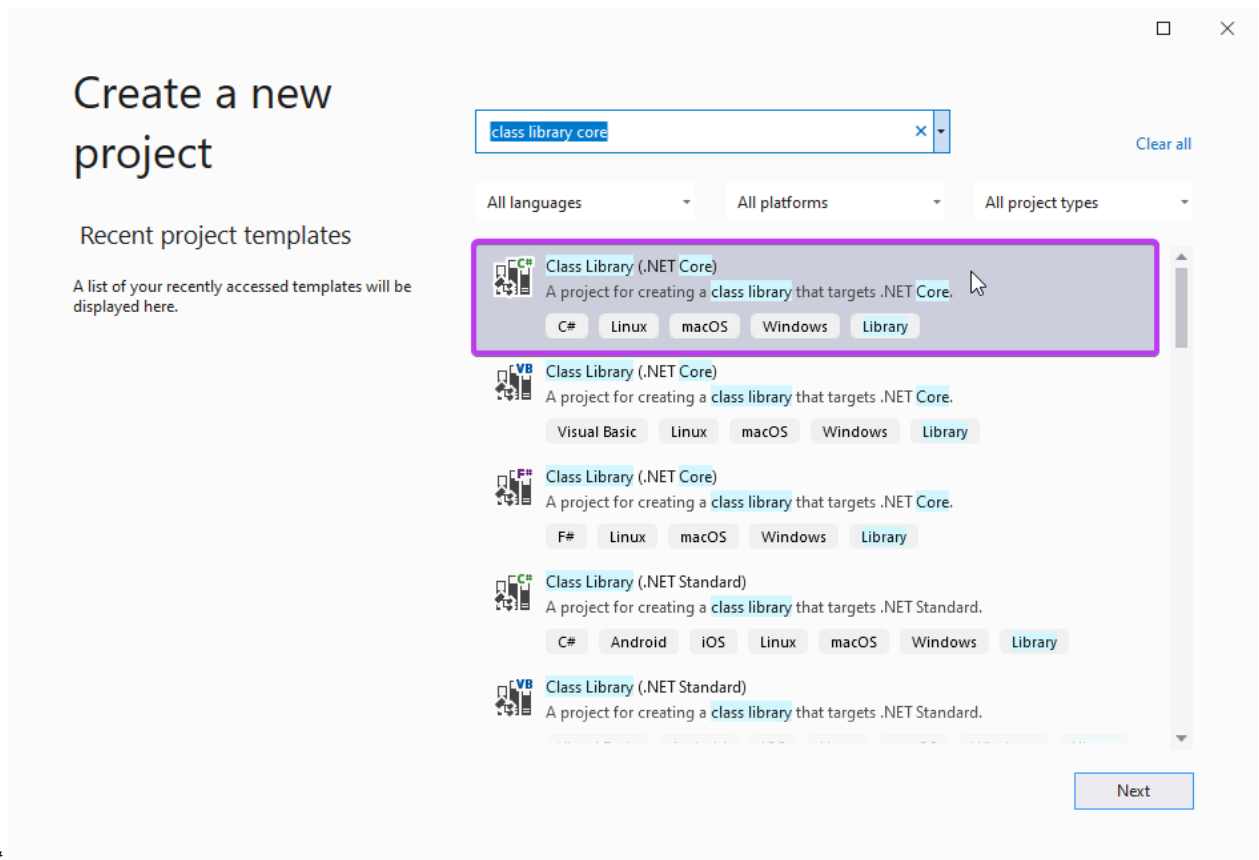
10 minutes

In this step you'll create the application that will be tested, also called System Under Test (SUT). The application will be a simple calculator in a *C#* class library.

**1-** Open Visual Studio and create a new *C#* class library by selecting “Create a new project” from the Visual Studio startup dialog:



2- Search for “Class library core” and select the “C# Class Library (.NET Core)” project template and click



*Next.*

**3-** Enter the project name as “SpecFlowCalculator”, choose a location to save the project and hit **Create**. In this scenario the solution will be saved to C:\work.

□

×

## Configure your new project

Class Library (.NET Core) C# Linux macOS Windows Library

Project name

SpecFlowCalculator

Location

C:\work

Solution name ⓘ

SpecFlowCalculator

☐ Place solution and project in the same directory

Back

Create

> **Note:** Do **NOT** use any special characters in your project name e.g. (parenthesis). This will result in build errors from the code generated by SpecFlow.

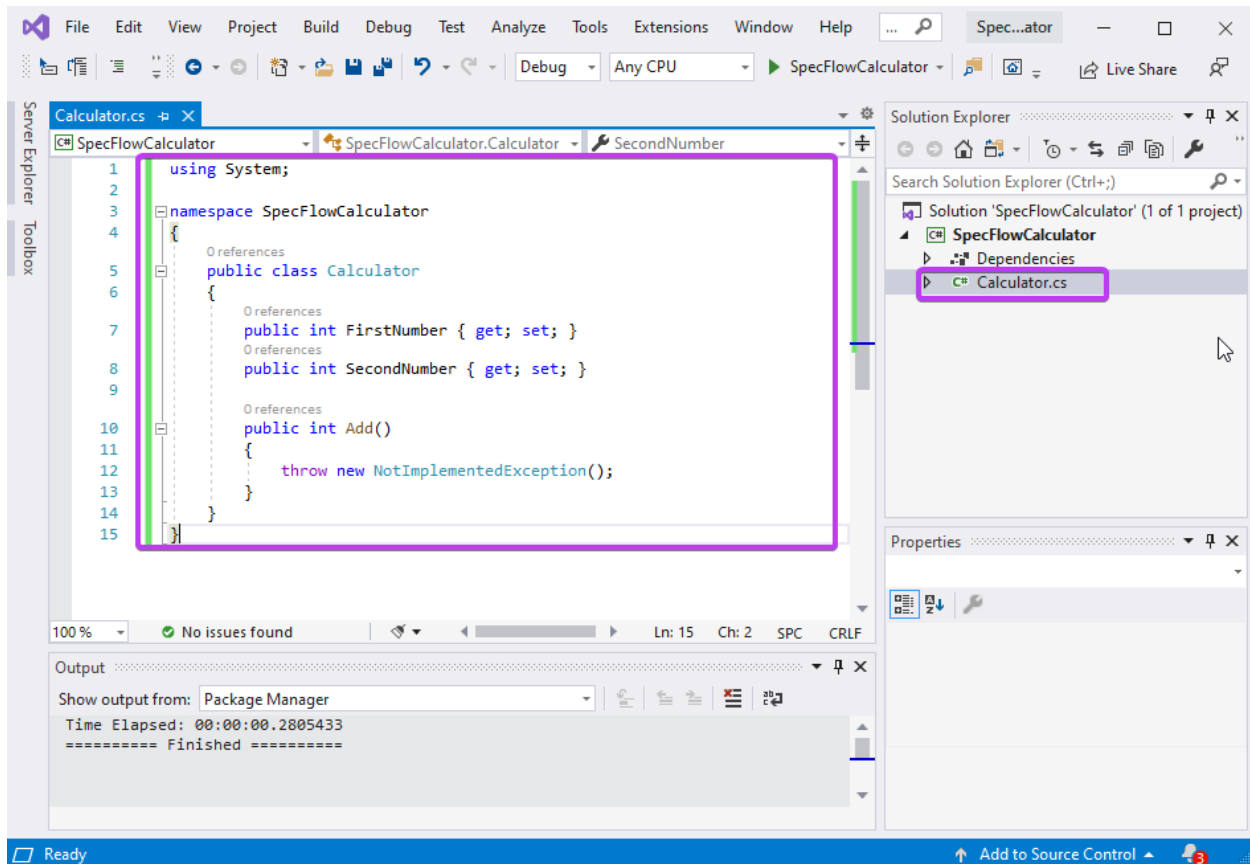
\*Solution name automatically updates to project name, leave it as is.

4- Rename `Class1.cs` to `Calculator.cs` and overwrite the content with the following code :

```
using System;

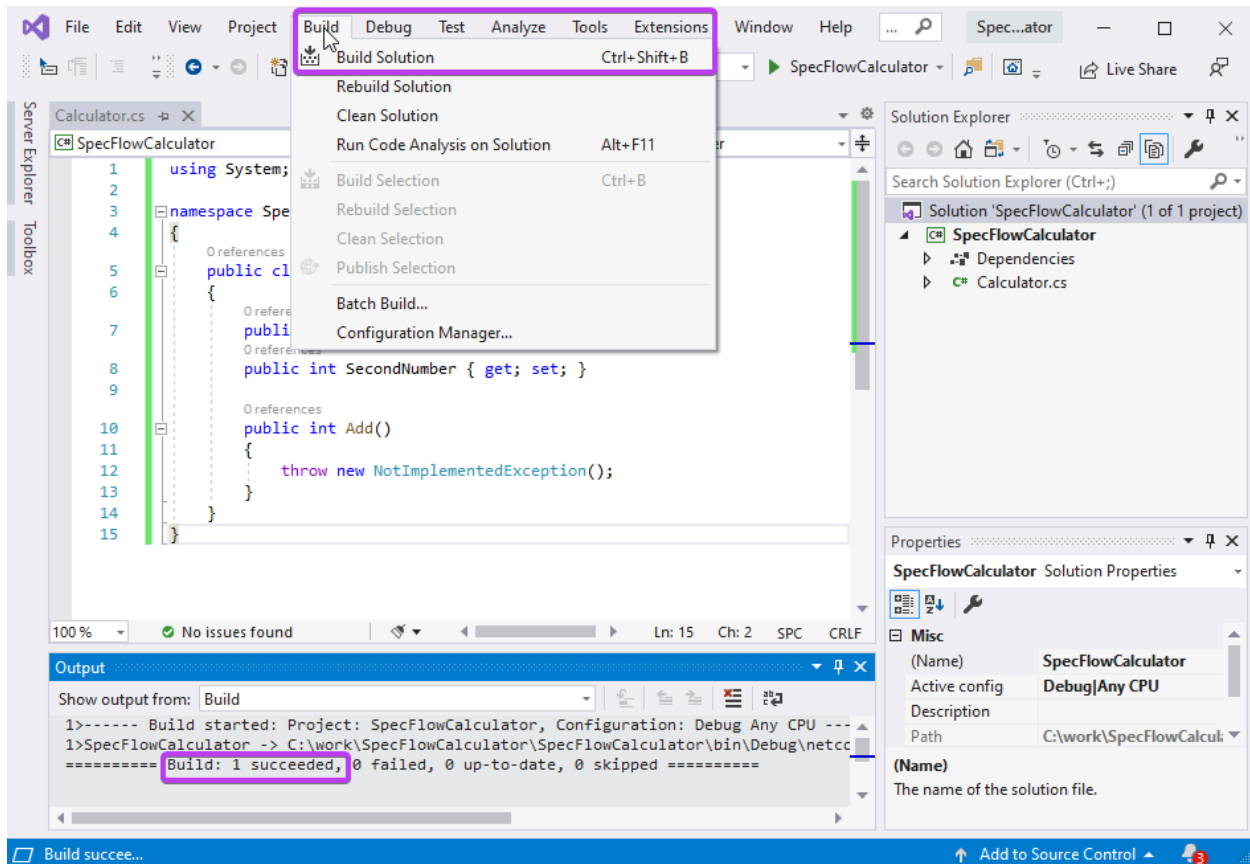
namespace SpecFlowCalculator
{
    public class Calculator
    {
        public int FirstNumber { get; set; }
        public int SecondNumber { get; set; }

        public int Add()
        {
            throw new NotImplementedException();
        }
    }
}
```



5- Now build the solution by navigating to “Build Build Solution” You will see a “Build : 1 Succeeded” message in the output window:

## Welcome to the Step-By-Step Getting Started Guide!



The calculator application is now built. In the next step you'll learn how to create a SpecFlow project.

## CHAPTER 3

---

### Create SpecFlow project

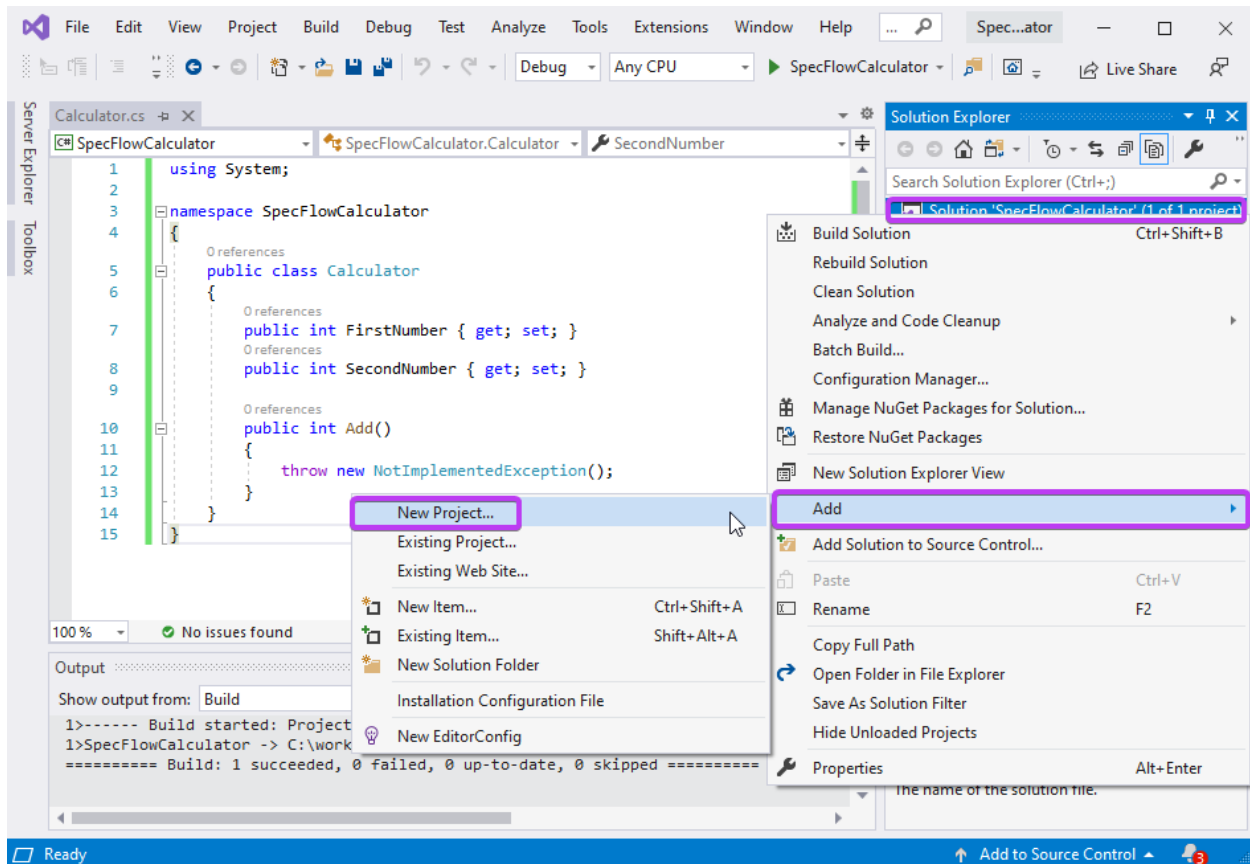
---

5 minutes

In this step you'll create a SpecFlow project and add it to the existing calculator solution:

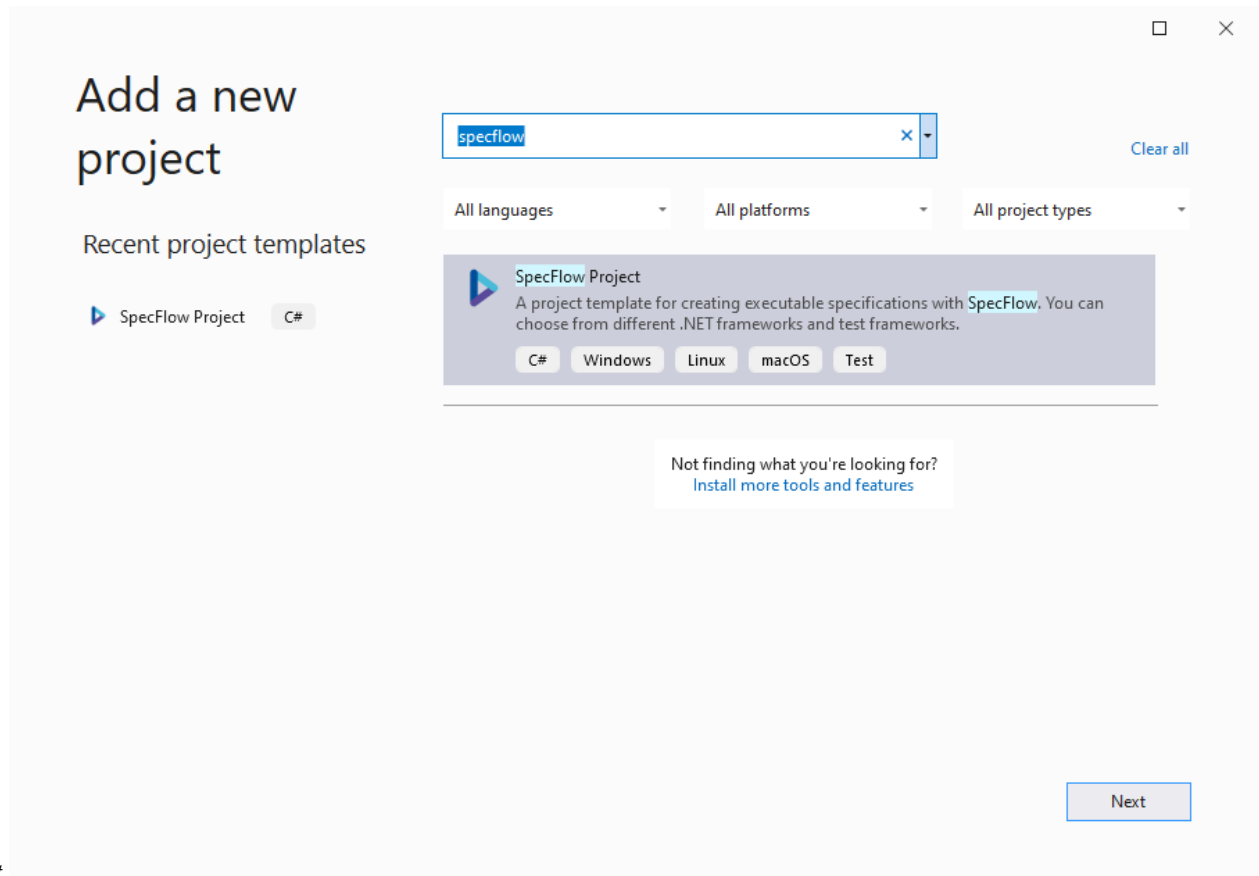
**1- Right-click** the solution item “Solution ‘SpecFlowCalculator’ (1 of 1 project)” under the Solution Explorer and select the “Add New Project...” menu item.

## Welcome to the Step-By-Step Getting Started Guide!



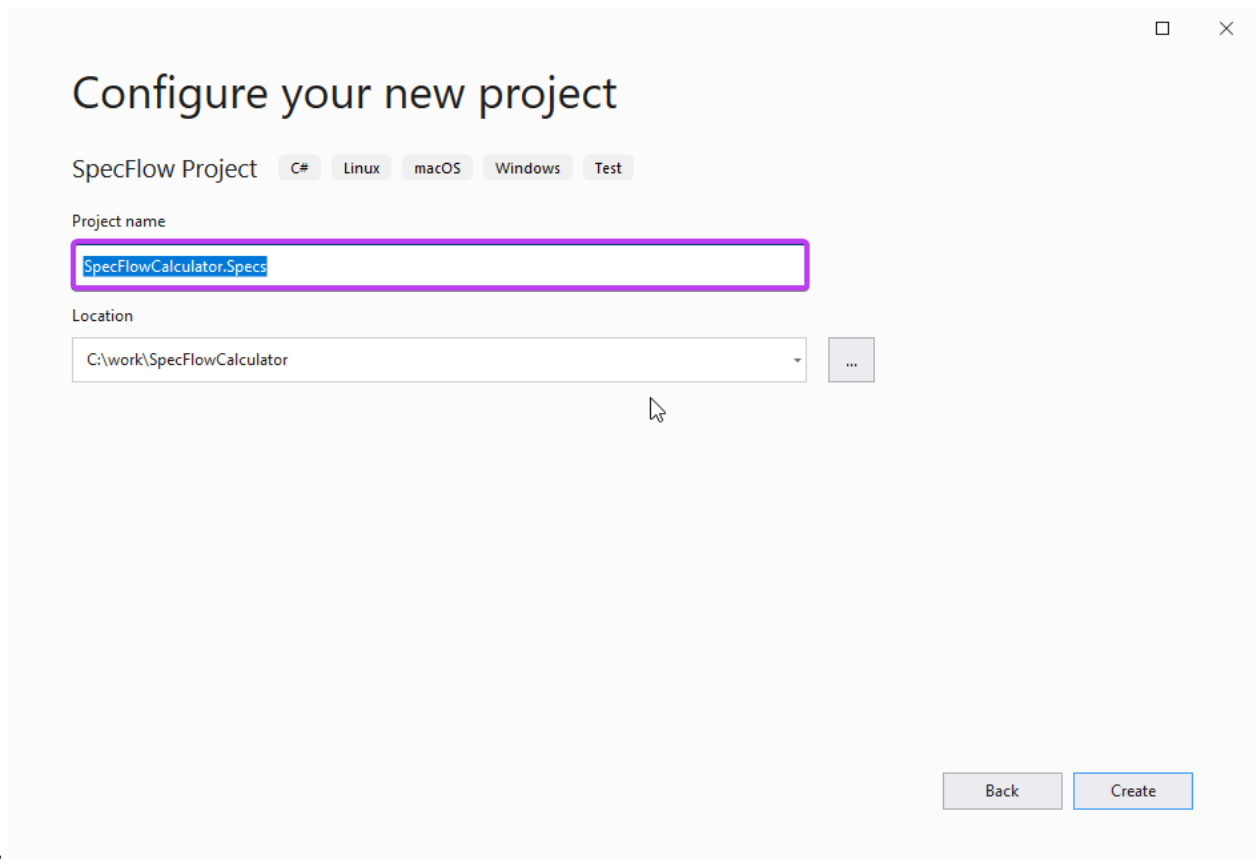
2- Search for “SpecFlow”, select the “SpecFlow Project” template and click





*Next.*

**3-** Enter the project name "SpecFlowCalculator.Specs". Keep the suggested location (the solution folder) and click



Configure your new project

SpecFlow Project C# Linux macOS Windows Test

Project name

SpecFlowCalculator.Specs

Location

C:\work\SpecFlowCalculator

Back Create

**Create.**

> **Note:** Do **NOT** use any special characters in your project name e.g. (parenthesis). This will result in build errors from the code generated by SpecFlow.

4- On this next screen you can configure the Test Framework (Runner) you want to use. We are using xUnit in this tutorial, but you may choose a different unit test runner if you have a particular preference. Hit **Create** once you have made your selection.

Create a new SpecFlow project

Framework

.NET 6.0

Test Framework

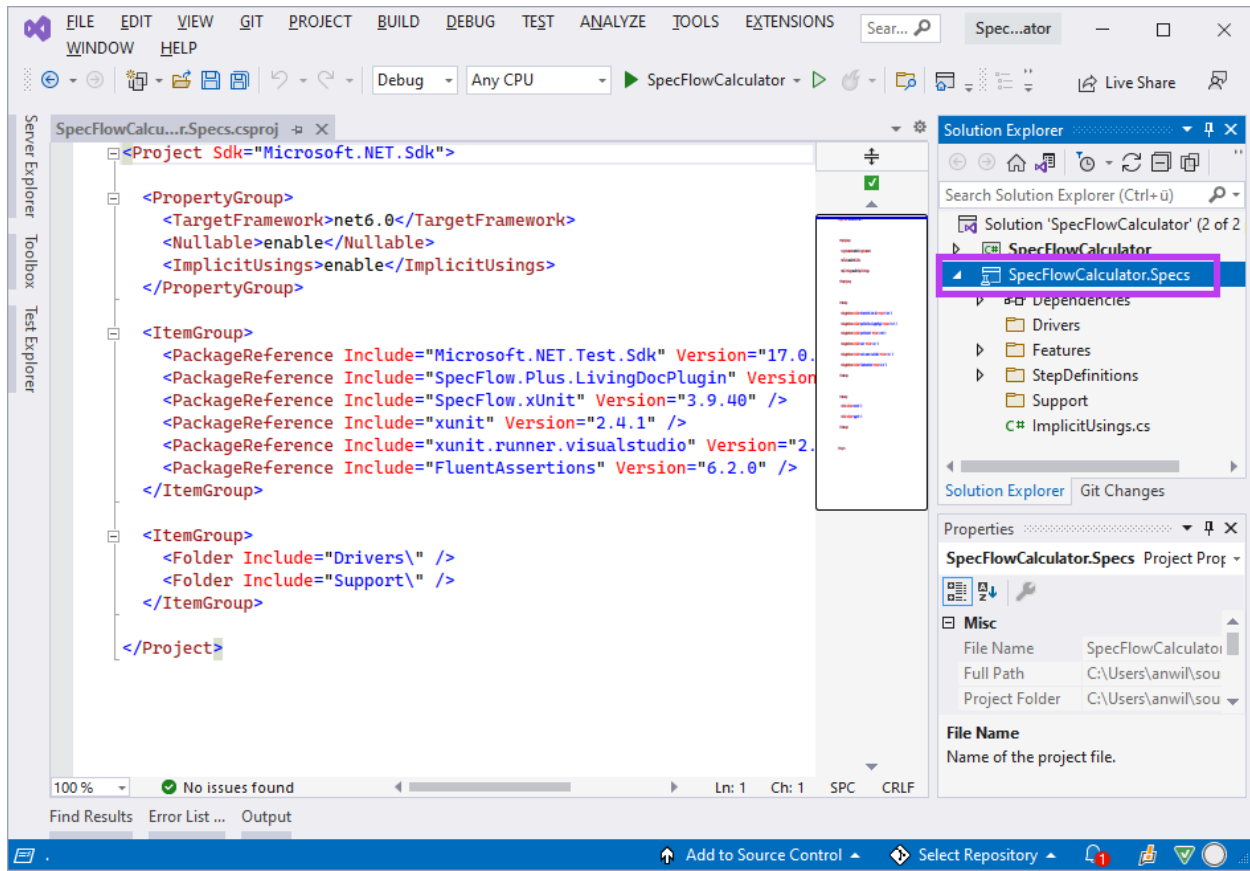
xUnit

☒ Add FluentAssertions library

Back Create

**5-** Visual Studio will now create the new SpecFlow project and resolve the NuGet packages in the background. You should see the new SpecFlow project in the Solution Explorer as per below:

## Welcome to the Step-By-Step Getting Started Guide!



In the next step you will learn how to add a project reference and how to use the test explorer.

---

### Create SpecFlow project - Continue

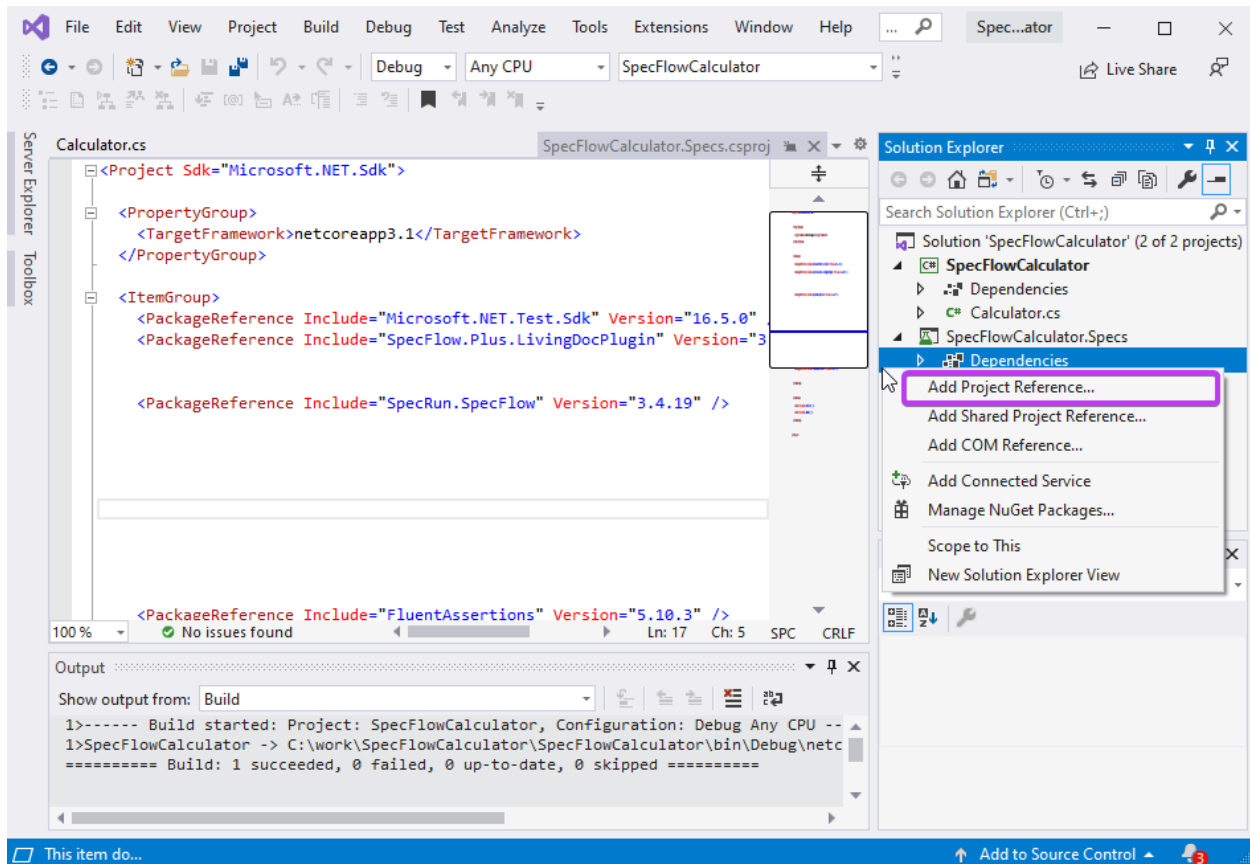
---

5 minutes

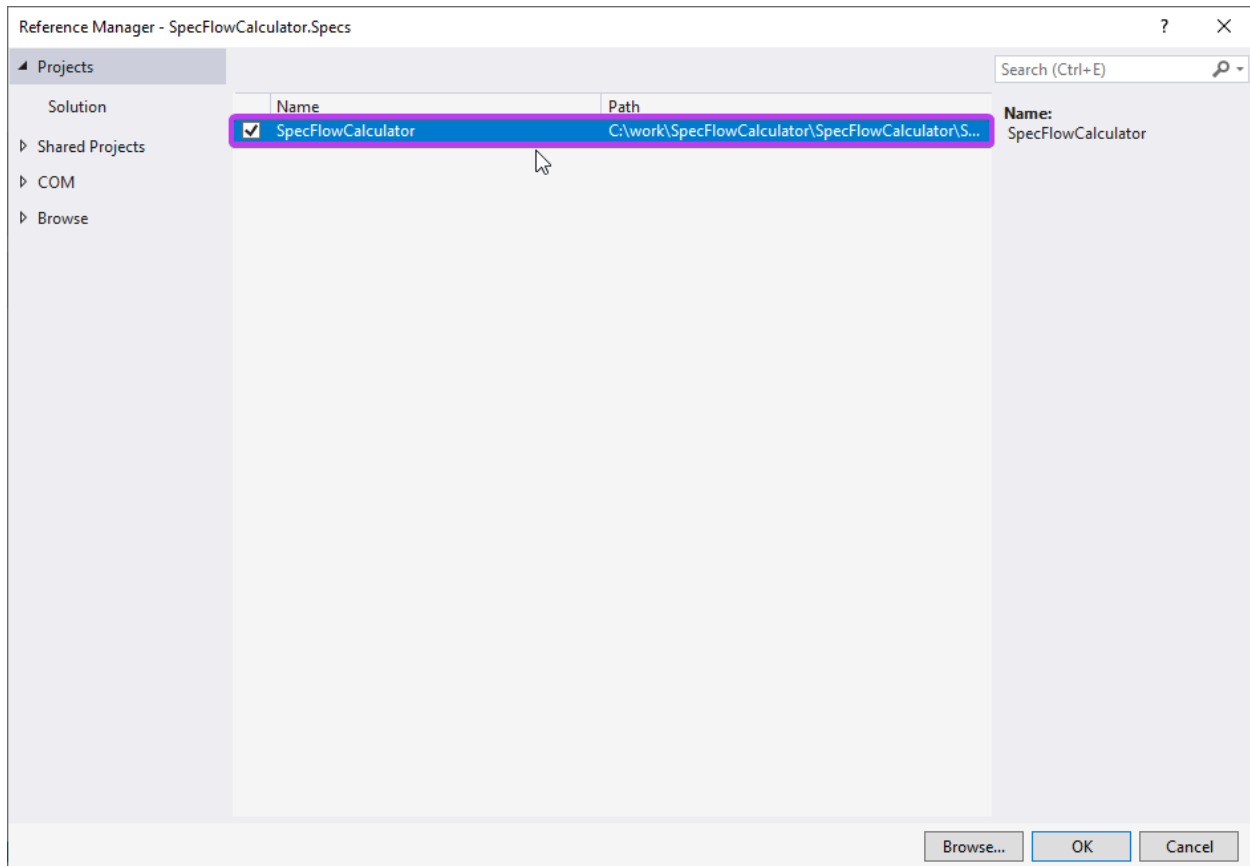
You will now add a project reference to the “SpecFlowCalculator” class library in the newly created SpecFlow project. This is necessary because we want to test the “Calculator” class implemented in the class library in the “SpecFlow-Calculator.Specs” project. To do this, follow the below steps:

**1-** Expand the project node “SpecFlowCalculator.Specs” in the Solution Explorer, right-click the “Dependencies” node and select the “Add Project Reference...” menu item.

## Welcome to the Step-By-Step Getting Started Guide!



2- In the “Reference Manager” dialog check the “SpecFlowCalculator” class library and click **OK**.

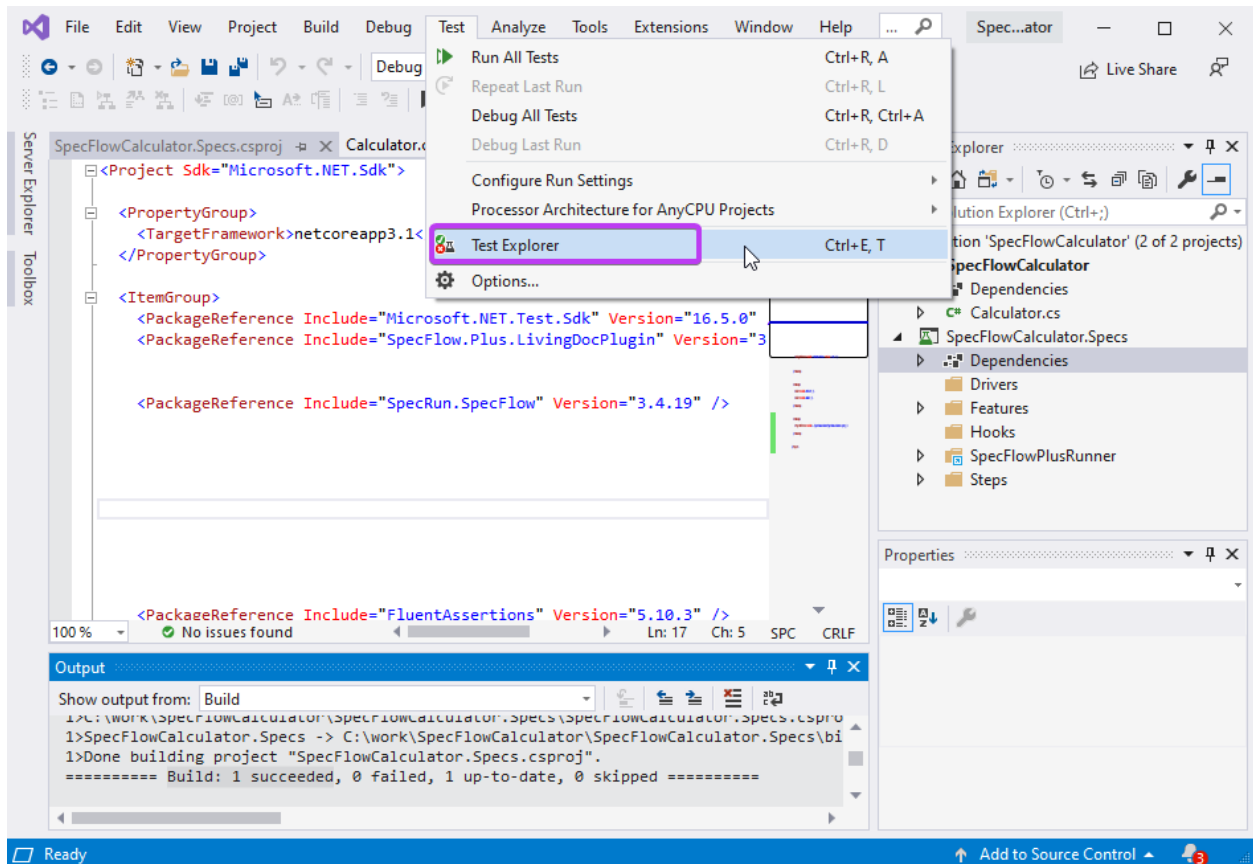


Now the solution is set up with a class library containing the implementation of the calculator and a SpecFlow project that contains the specification and tests of the calculator.

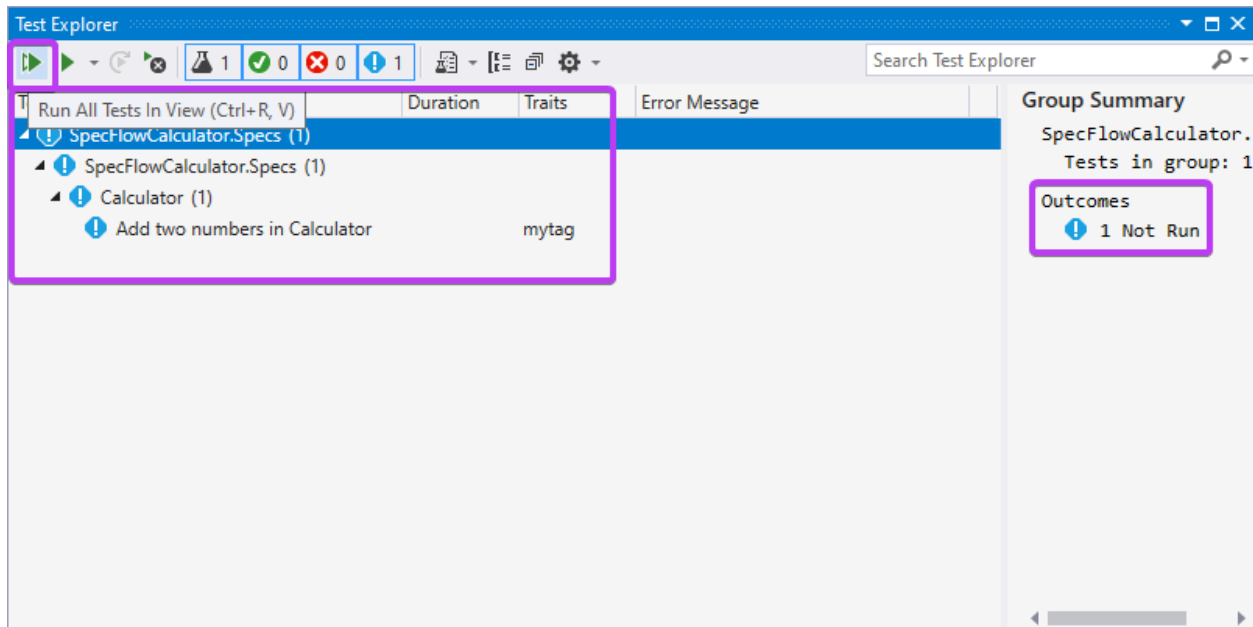
**3-** Now build the solution. You should see the “Build : 1 succeeded” message in the output window. *\*Refer to page 2 step 6 of this guide if you cannot recall how to build the solution.*

**4-** Open the test explorer dialog from the menu “Tests Test Explorer”.

## Welcome to the Step-By-Step Getting Started Guide!



5- You should see a test already added to the SpecFlow project by the project template. Run the test using the “Run All Tests in View” icon. Note that the outcome/status of the test remains “Not Run” as the test has not executed yet.



In the next step you will learn how to automate your first scenario.



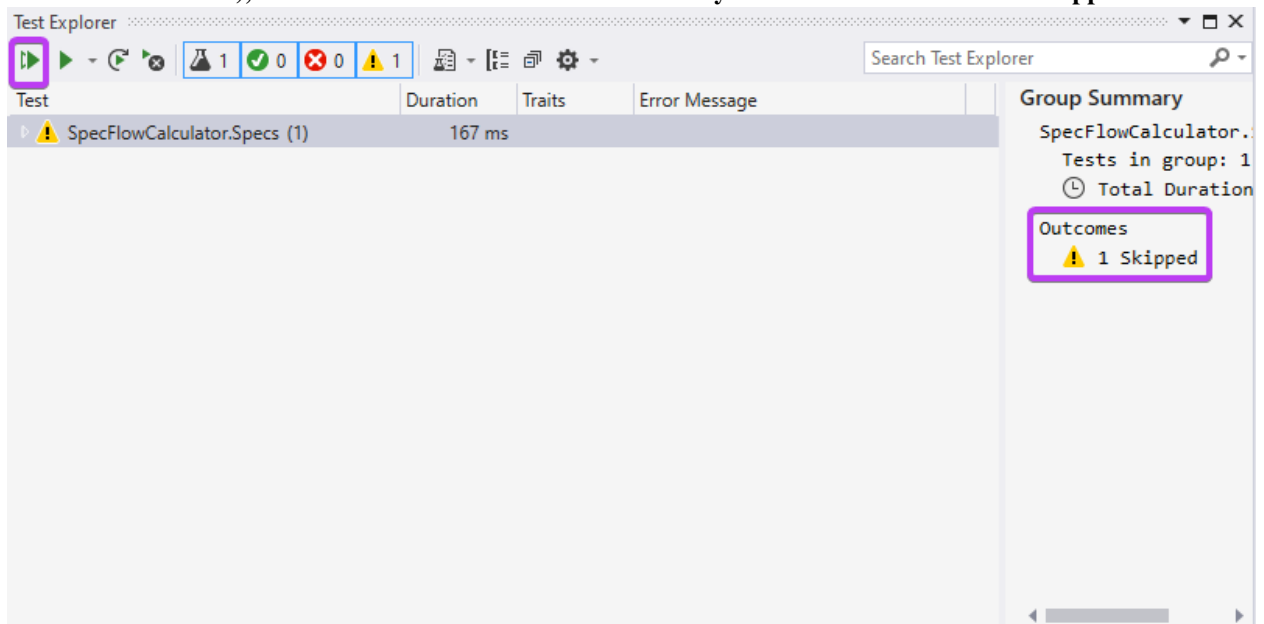
## CHAPTER 5

### Bind the first step

10 minutes

In this step you'll bind your first step (automate your first scenario step with SpecFlow).

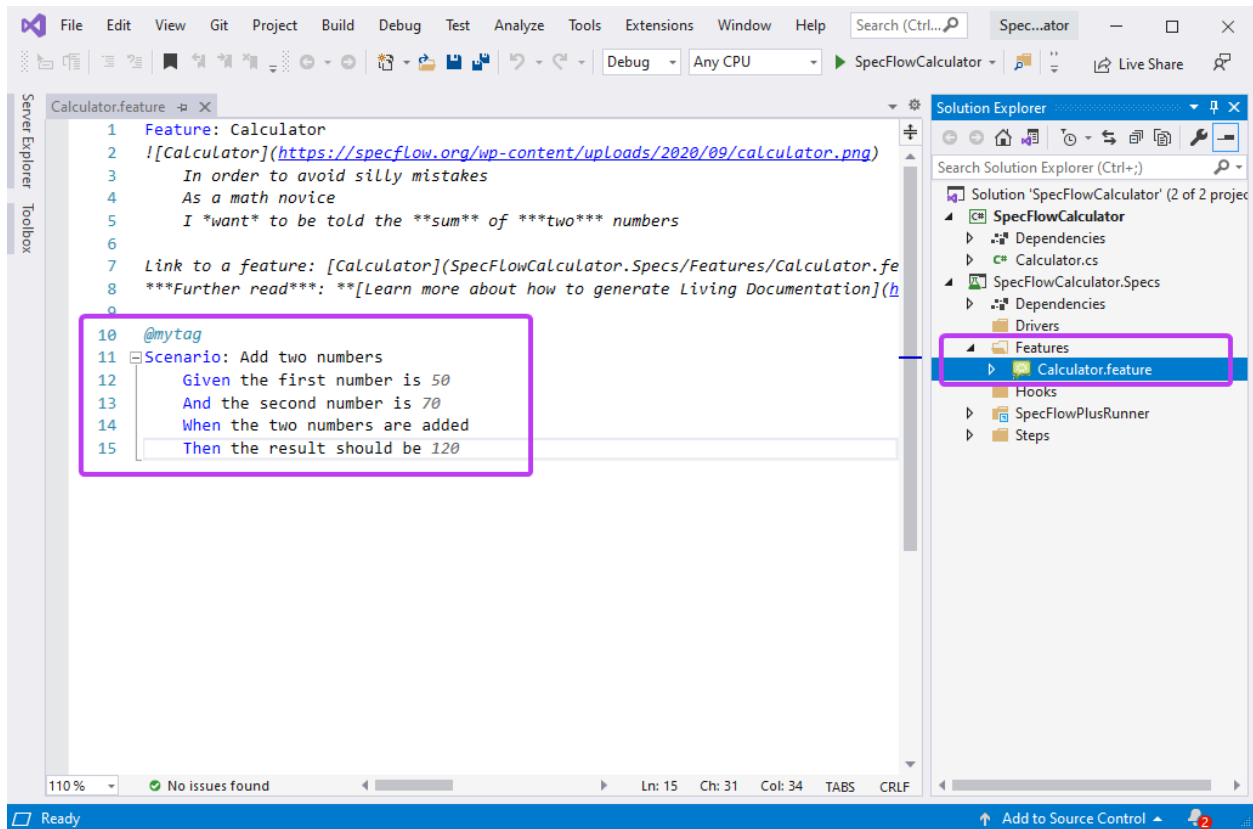
**\*If you skipped the previous page make sure you execute the tests with your preferred runner. The test explorer would look like below (see the duration in milliseconds), but it does not do much yet and shows the “Skipped” status.**



tus.

1- Open the Calculator.feature file by double-clicking it in the Solution Explorer (SpecFlowCalculator.Specs Features Calculator.feature)

## Welcome to the Step-By-Step Getting Started Guide!



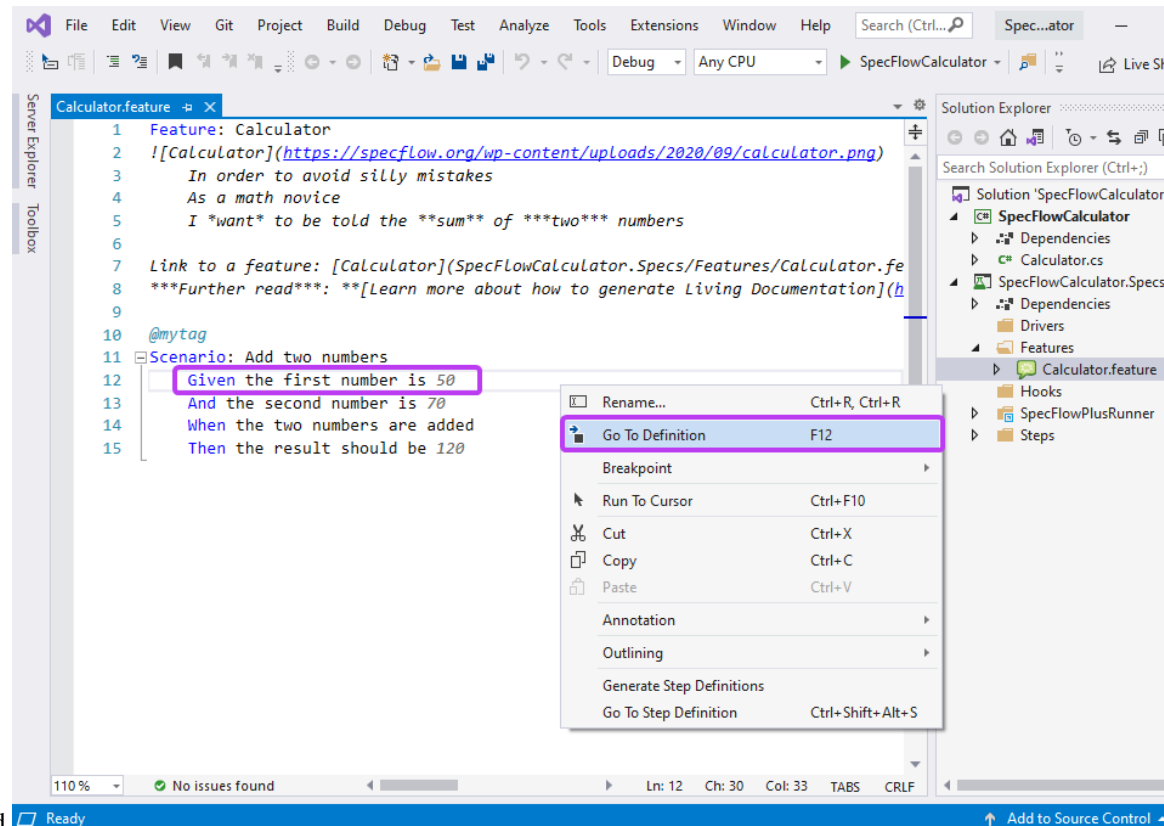
The purpose of this feature file is to document the expected behavior of the calculator in a way that it is both human-readable and suitable for test automation. SpecFlow uses the Gherkin language where you can phrase the scenarios using *Given/When/Then* steps. Currently there is a single scenario (automatically added by the SpecFlow project template) that describes how adding two numbers should work with the calculator.


Here is a closer look at the Gherkin scenario used in this template:

```
Scenario: Add two numbers
  Given the first number is 50
  And the second number is 70
  When the two numbers are added
  Then the result should be 120
```

Based on the scenario text, SpecFlow generates an automated test that executes the scenario. However, it is **not yet defined** what the steps of the scenario should actually “do”.

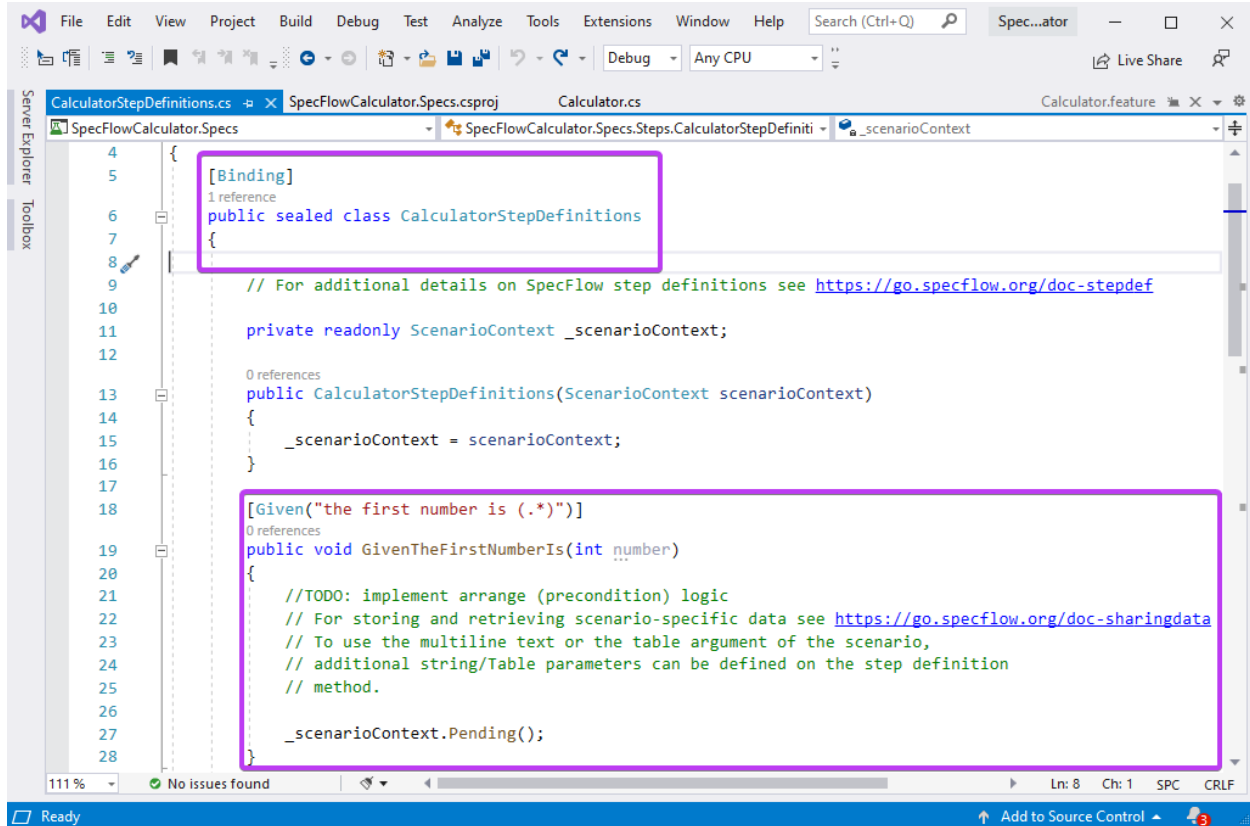
2- Right-click the first *Given* step “Given the first number is 50” and select either the “Go To Definition” or the “Go To



Step Definition” command.  Ready

Visual Studio locates the step definition (binding) that belongs to this step. In this example, it opens the `CalculatorStepDefinitions` class and jumps to the `GivenTheFirstNumberIs` method.

## Welcome to the Step-By-Step Getting Started Guide!



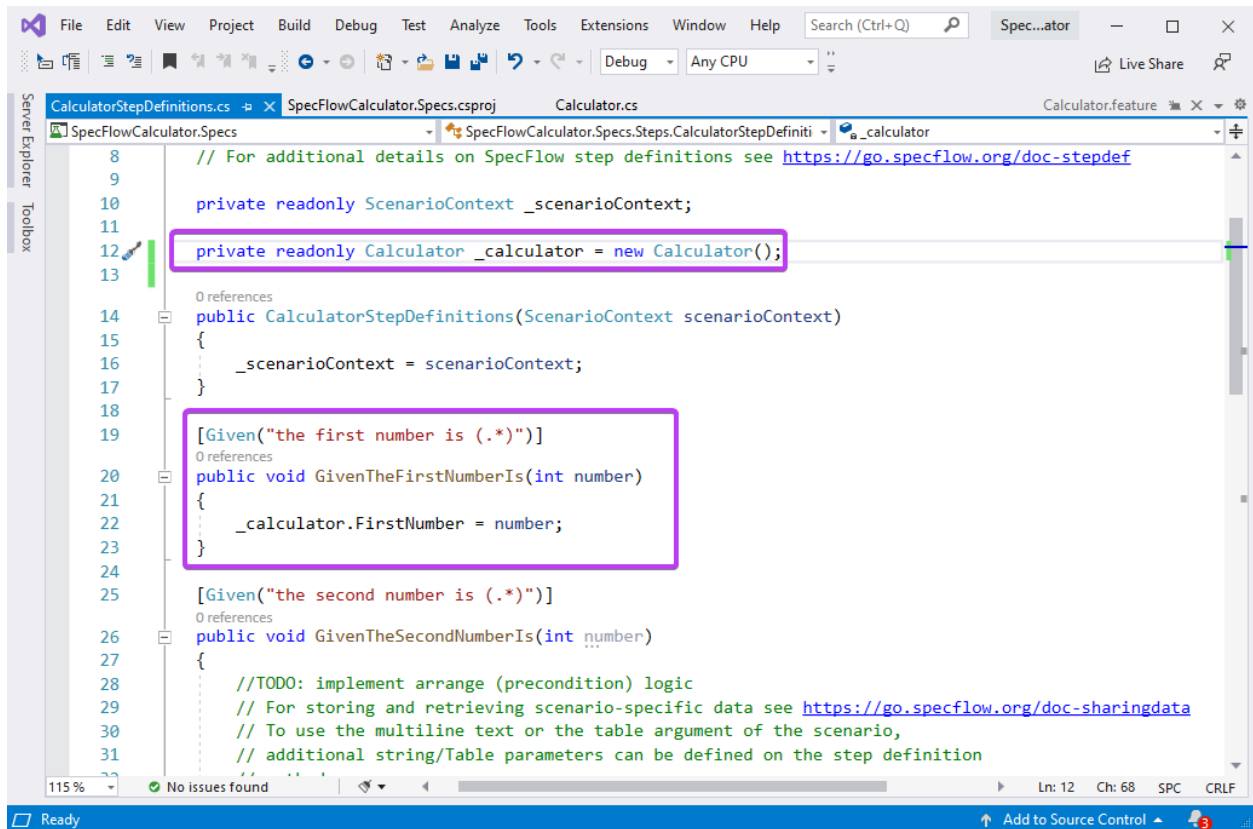
\*The step definition is located based on the `[Binding]` attribute on the class and the `[Given]` attribute on the method. The regular expression of the `Given` attribute matches the text of the scenario step.

3- Add the below field to the class to instantiate the calculator that we want to test and created in [Step 2](#) of this guide (SUT).

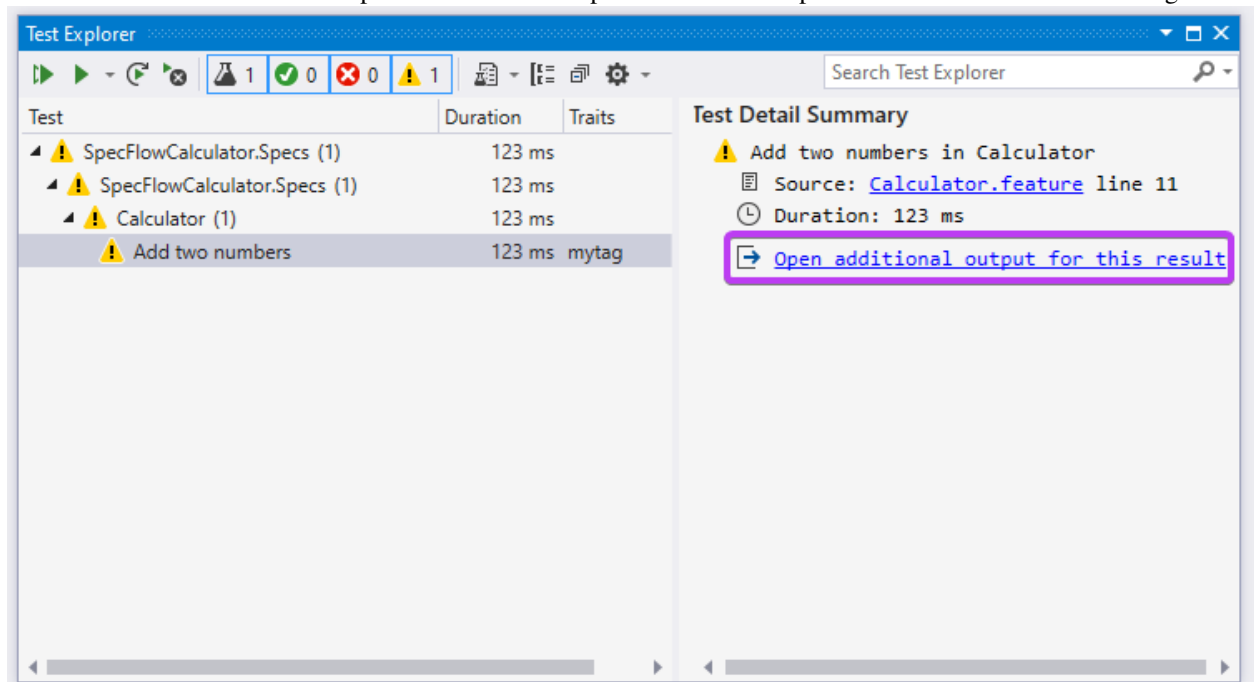
```
private readonly Calculator _calculator = new Calculator();
```

4- Replace the implementation of the first step definition method with the below code which sets the first number of the calculator.

```
[Given("the first number is (.*)")]
public void GivenTheFirstNumberIs(int number)
{
    _calculator.FirstNumber = number;
}
```



5- Execute the test in the Test Explorer and click “Open additional output for this result” from the right



pane.

In the detailed output you can see that the first step “Given the first number is 50” has been matched to the step definition method “CalculatorStepDefinitions.GivenTheFirstNumberIs” as expected, and it has been called with the argument 50. The done status means that the step executed successfully with no errors:

```
Given the first number is 50  
-> done: CalculatorStepDefinitions.GivenTheFirstNumbers(50) (0.0s)
```

The next step in the scenario is pending, as we have not yet implemented it:

```
And the second number is 70  
-> pending: CalculatorStepDefinitions.GivenTheSecondNumbers(70)
```

In the next step you will bind the rest of the scenario steps.

---

### Bind remaining steps

---

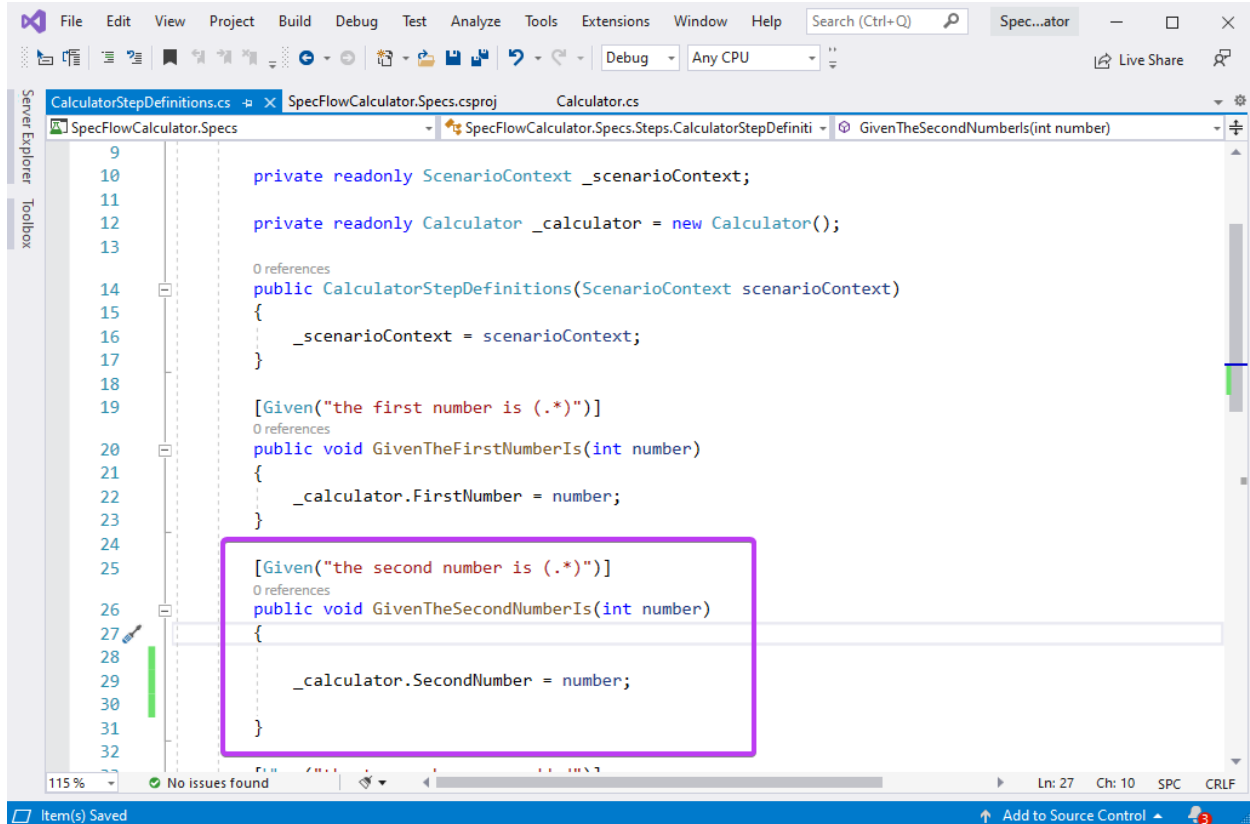
10 minutes

In this step you'll bind the remaining steps of the scenario.

**1-** Similar to the previous page, navigate to the bindings from the feature file by right-clicking the second Given step "And the second number is 70" and select either the "Go To Definition" or the "Go To Step Definition" command. Alternatively you can open the `CalculatorStepDefinitions.cs` directly.

**2-** Implement the binding of the second step "And the second number is 70" by replacing the code of the `GivenTheSecondNumberIs` method with the below:

```
[Given("the second number is (.*)")]
public void GivenTheSecondNumberIs(int number)
{
    _calculator.SecondNumber = number;
}
```



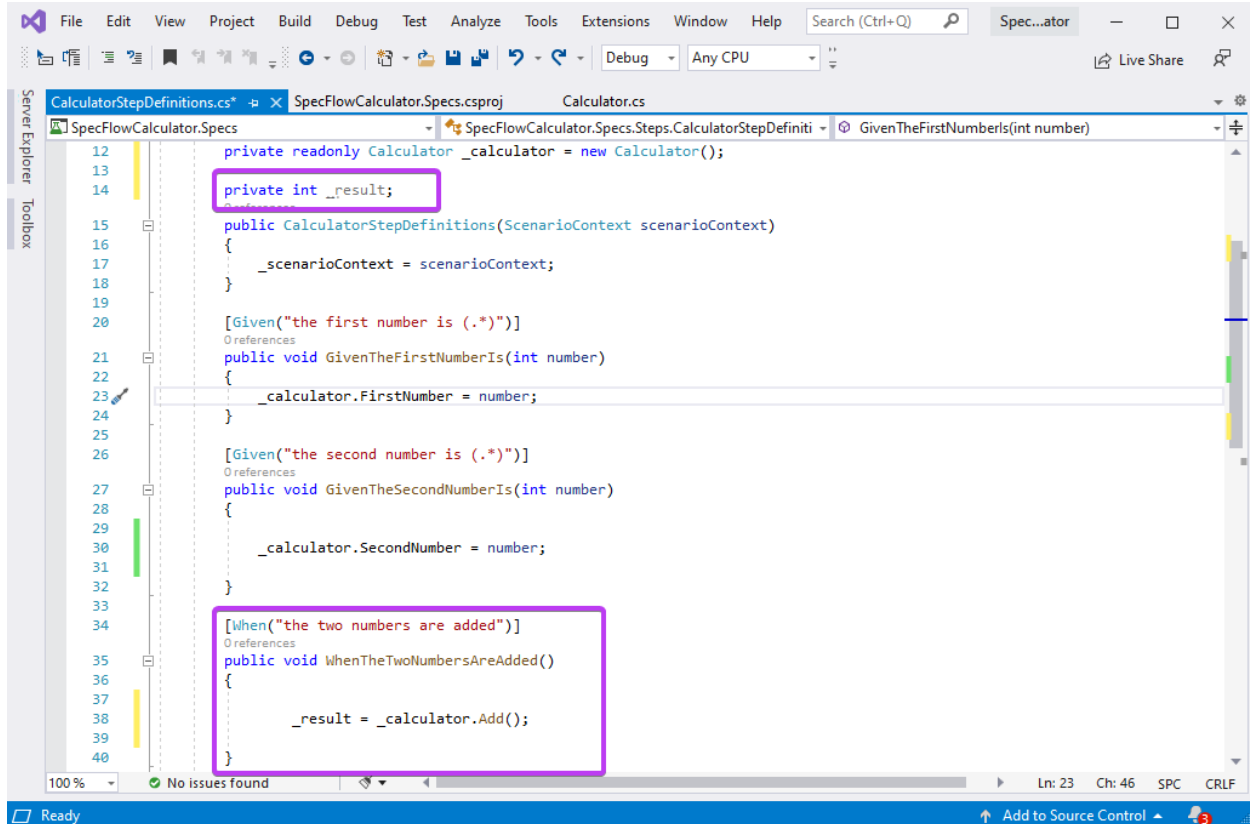
\*We use the “And” keyword in the Gherkin scenario for better readability. The “And” keyword will be interpreted as “Given”, “When” or “Then” depending on the previous step(s) in the scenario. In this example the “And the second number is 70” is interpreted as a “Given” step because the previous step is a “Given” step.

3- Next, implement the binding of the third step, “When the two numbers are added”, by replacing the code of the `WhenTheTwoNumbersAreAdded` method with the below. The method must have a `When` attribute, as it belongs to the “When” step in the scenario.

```
private int _result;

[When("the two numbers are added")]
public void WhenTheTwoNumbersAreAdded()
{
    _result = _calculator.Add();
}
```





This implementation calls the Add method of the calculator. Note that the result of the addition is not stored by the calculator in a property/field but it is returned to the caller. It's a good idea to store the returned value in a field so that we can work with the result afterwards.

4- Implement the binding of the last step, “Then the result should be 120”, by replacing the code of the ThenTheResultShouldBe method. The method must have a Then attribute, as it belongs to a “Then” step in the scenario.

Add a namespace using for FluentAssertions at the top of the file:

```
using FluentAssertions;
```

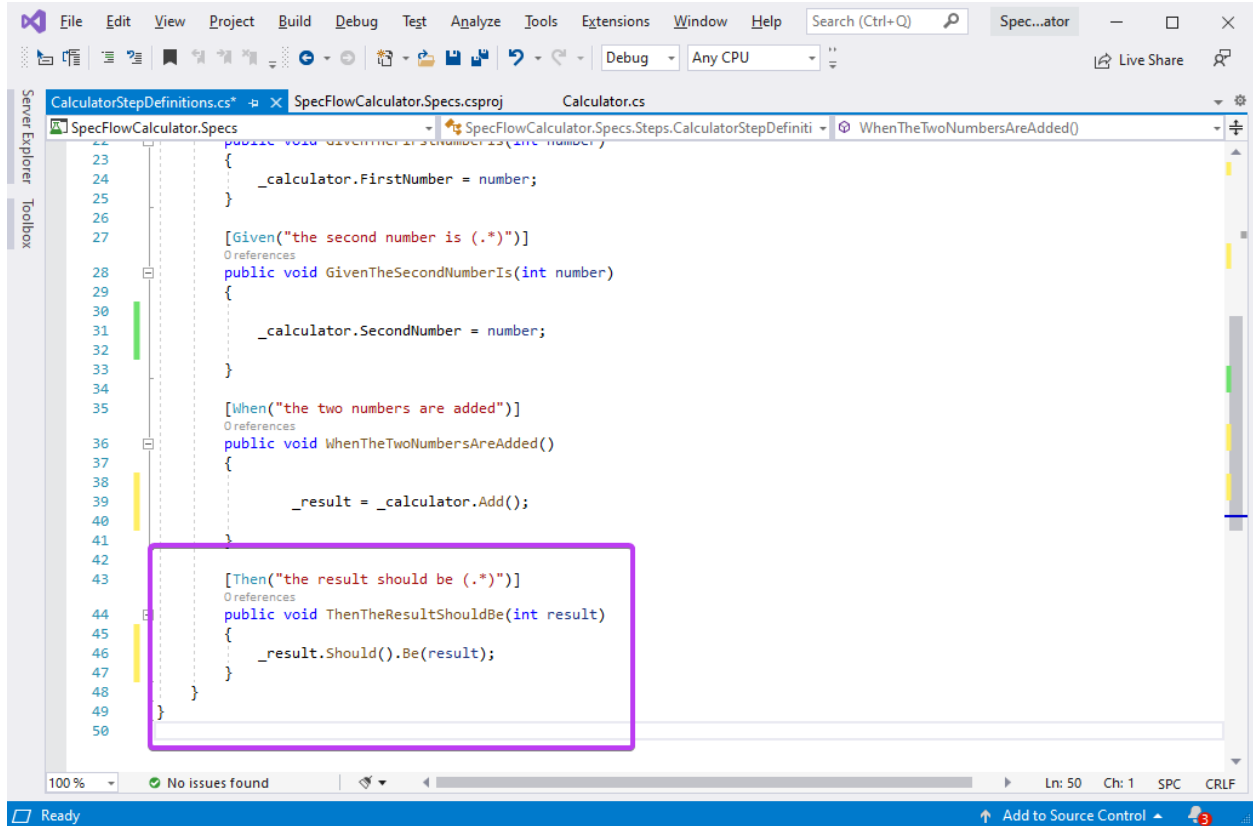
Use the below code for implementation of the “Then” step which validates if the result of the addition matches the expected value (using the FluentAssertions library).

```

[Then("the result should be (.*)")]
public void ThenTheResultShouldBe(int result)
{
    _result.Should().Be(result);
}

```

## Welcome to the Step-By-Step Getting Started Guide!



After implementing all step definitions and cleaning up the file you should have the following code:

```
using FluentAssertions;
using TechTalk.SpecFlow;

namespace SpecFlowCalculator.Specs.Steps
{
    [Binding]
    public sealed class CalculatorStepDefinitions
    {
        // For additional details on SpecFlow step definitions see https://go.
        // specflow.org/doc-stepdef

        private readonly ScenarioContext _scenarioContext;

        private readonly Calculator _calculator = new Calculator();
        private int _result;

        public CalculatorStepDefinitions(ScenarioContext scenarioContext)
        {
            _scenarioContext = scenarioContext;
        }

        [Given("the first number is (.*)")]
        public void GivenTheFirstNumberIs(int number)
        {
            _calculator.FirstNumber = number;
        }
    }
}
```

(continues on next page)

(continued from previous page)

```
[Given("the second number is (.*)")]
public void GivenTheSecondNumberIs(int number)
{
    _calculator.SecondNumber = number;
}

[When("the two numbers are added")]
public void WhenTheTwoNumbersAreAdded()
{
    _result = _calculator.Add();
}

[Then("the result should be (.*)")]
public void ThenTheResultShouldBe(int result)
{
    _result.Should().Be(result);
}
}
```

5- Build the solution. The build should succeed.


6- Run the test again.

The test should execute and fail, this is expected. In the Test Detail Summary pane of Test Explorer you can see that a `NotImplementedException` has been thrown in the `Add` method of the

calculator.

7- Click on the “Open additional output for this result” below the stack trace to see a more detailed log of the

Test Name: Add two numbers

Test Outcome:  Failed

Message: The method or operation is not implemented.

Standard Output

```
-> -> Loading plugin C:\work\SpecFlowCalculator\SpecFlowCalculator.Specs\bin\Debug\netcoreapp3.1\LivingDoc.SpecFlowPlugin.dll
-> -> Loading plugin C:\work\SpecFlowCalculator\SpecFlowCalculator.Specs\bin\Debug\netcoreapp3.1\SpecRun.Runtime.SpecFlowPlugin.dll
-> -> Using default config

Given the first number is 50
-> done: CalculatorStepDefinitions.GivenTheFirstNumberIs(50) (0,1s)

And the second number is 70
-> done: CalculatorStepDefinitions.GivenTheSecondNumberIs(70) (0,0s)

When the two numbers are added
-> error: The method or operation is not implemented.

Then the result should be 120
-> skipped because of previous errors
```

Standard Error

```
The method or operation is not implemented.System.NotImplementedException: The method or operation is not implemented.
at SpecFlowCalculator.Calculator.Add() in C:\work\SpecFlowCalculator\SpecFlowCalculator\Calculator.cs:line 12
at SpecFlowCalculator.Specs.Steps.CalculatorStepDefinitions.WhenTheTwoNumbersAreAdded() in C:\work\SpecFlowCalculator
\SpecFlowCalculator.Specs\Steps\CalculatorStepDefinitions.cs:line 39
at TechTalk.SpecFlow.Bindings.BindingInvoker.InvokeBinding(IBinding binding, IContextManager contextManager, Object[] arguments,
ITestTracer testTracer, TimeSpan& duration) in D:\a\1\s\TechTalk.SpecFlow\Bindings\BindingInvoker.cs:line 69
at TechTalk.SpecFlow.Infrastructure.TestExecutionEngine.ExecuteStepMatch(BindingMatch match, Object[] arguments) in D:\a\1\s
\TechTalk.SpecFlow\Infrastructure\TestExecutionEngine.cs:line 514
at TechTalk.SpecRun.SpecFlowPlugin.Runtime.RunnerTestExecutionEngine.ExecuteStepMatch(BindingMatch match, Object[] arguments)
at TechTalk.SpecFlow.Infrastructure.TestExecutionEngine.ExecuteStep(IContextManager contextManager, StepInstance stepInstance) in D:\a\1
\s\TechTalk.SpecFlow\Infrastructure\TestExecutionEngine.cs:line 435
at TechTalk.SpecFlow.Infrastructure.TestExecutionEngine.OnAfterLastStep() in D:\a\1\s\TechTalk.SpecFlow\Infrastructure
\TestExecutionEngine.cs:line 260
at TechTalk.SpecRun.SpecFlowPlugin.Runtime.RunnerTestExecutionEngine.OnAfterLastStep()
at TechTalk.SpecFlow.TestRunner.CollectScenarioErrors() in D:\a\1\s\TechTalk.SpecFlow\TestRunner.cs:line 60
at SpecFlowCalculator.Specs.Features.CalculatorFeature.ScenarioCleanup()
at SpecFlowCalculator.Specs.Features.CalculatorFeature.AddTwoNumbers() in C:\work\SpecFlowCalculator\SpecFlowCalculator.Specs
\Features\CalculatorFeature.cs:line 15
```

scenario.

You can see that the first two “Given” steps executed successfully and the “When the two numbers are added” step failed with an error. This is because the addition method of the calculator is not implemented yet.

In the next step you’ll fix the implementation of the calculator to fix this error.

---

### Fix implementation

---

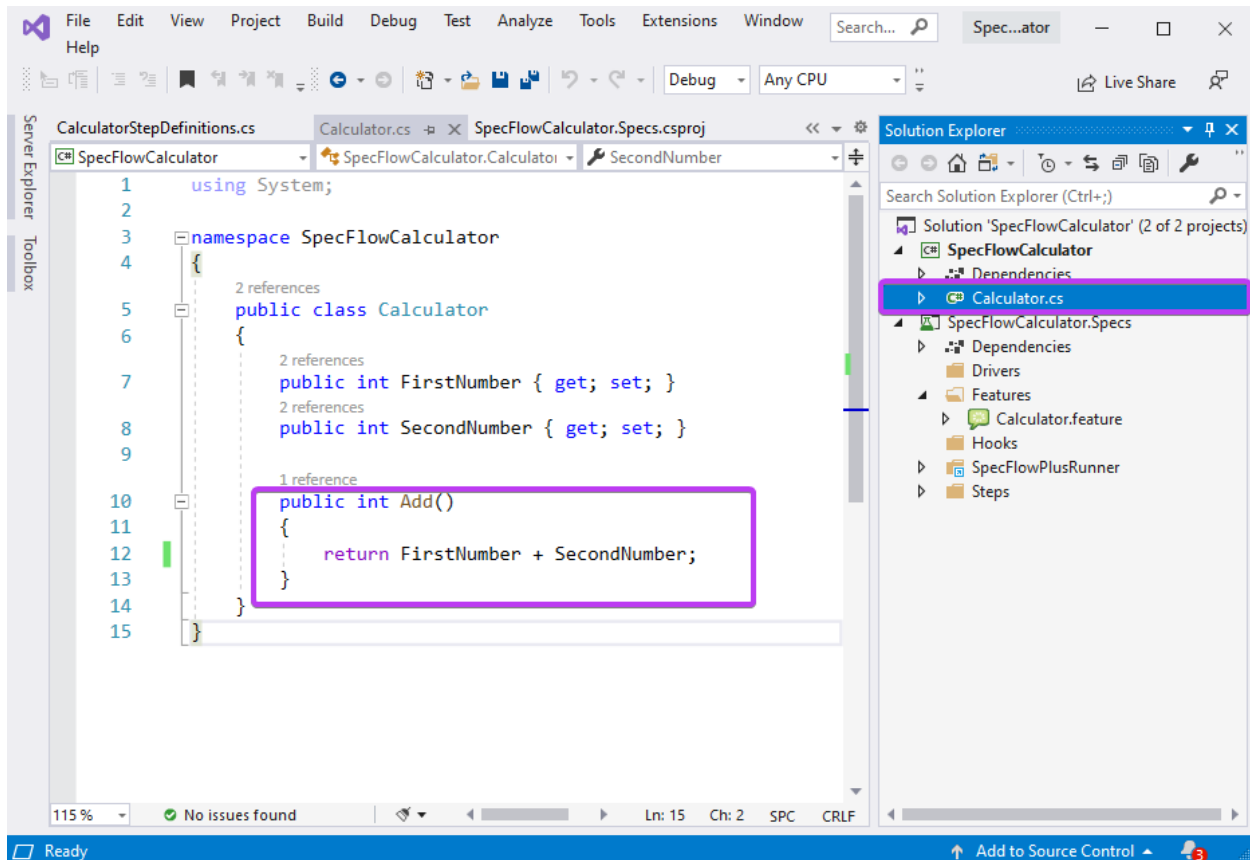
3 minutes

In this step you'll fix the implementation error of the calculator in the previous page.

**1-** Open `Calculator.cs` in the `SpecFlowCalculator` class library and replace the implementation of the `Add` method with the below code:

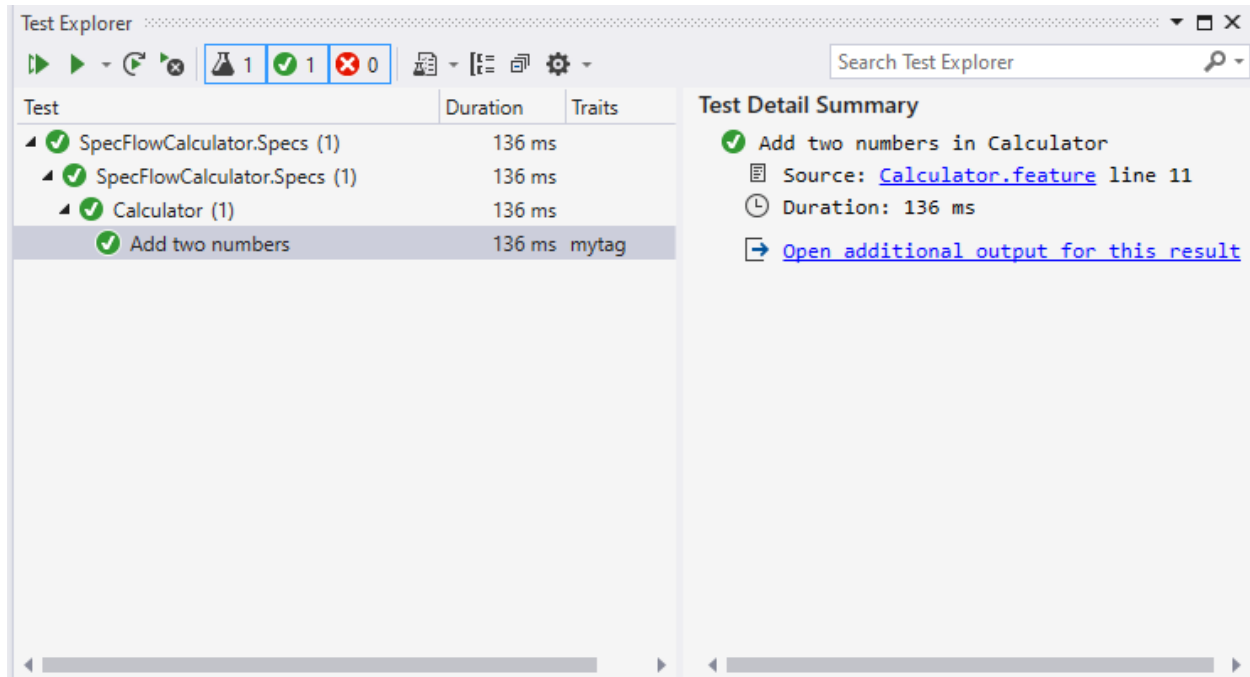
```
public int Add()
{
    return FirstNumber + SecondNumber;
}
```

## Welcome to the Step-By-Step Getting Started Guide!

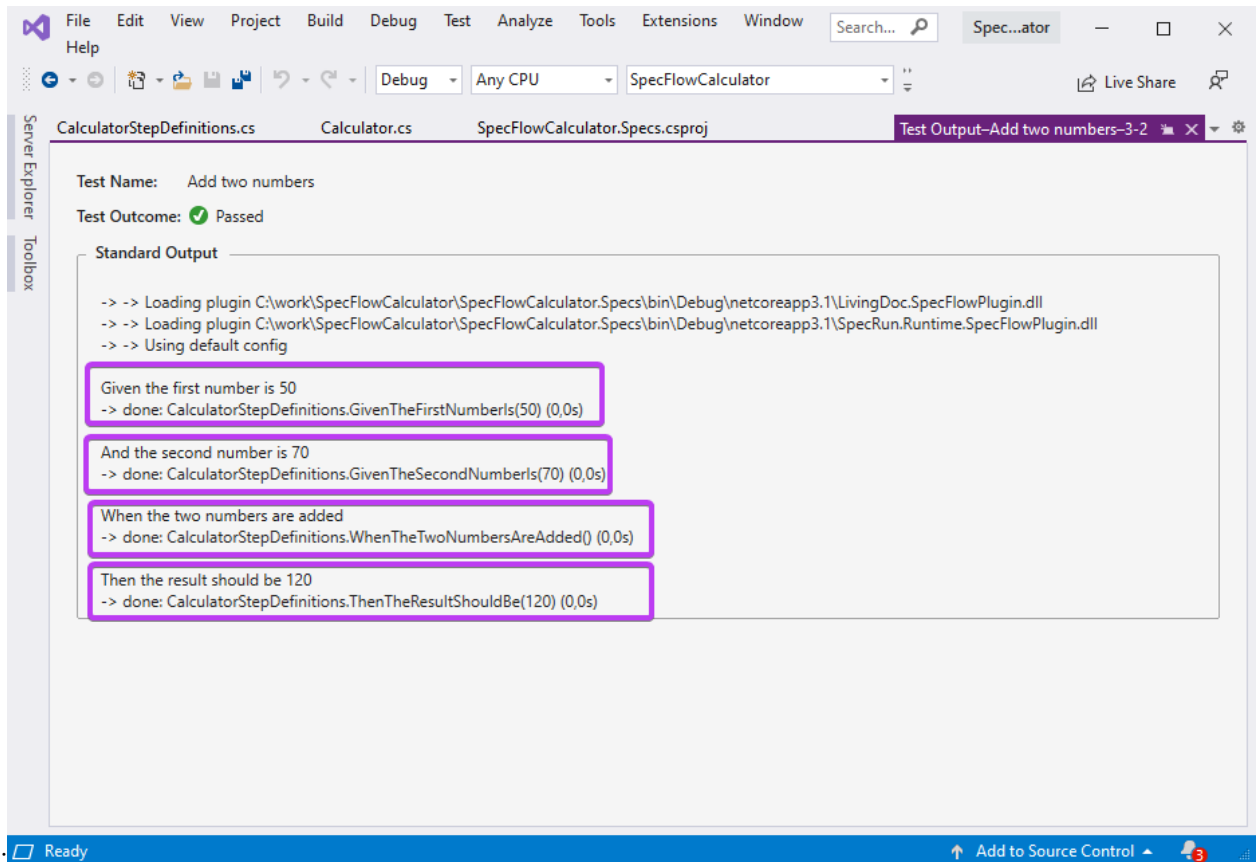


2- Build the solution. The build should succeed.

3- Run the test. The test should now execute and succeed with the green tick marks indicating no errors:



4- Click on the “Open additional output for this result” to see a more detailed log of the



scenario:

You can see that each step executed successfully and the test is passed.

In the next step you'll learn how to generate living documentation.





---

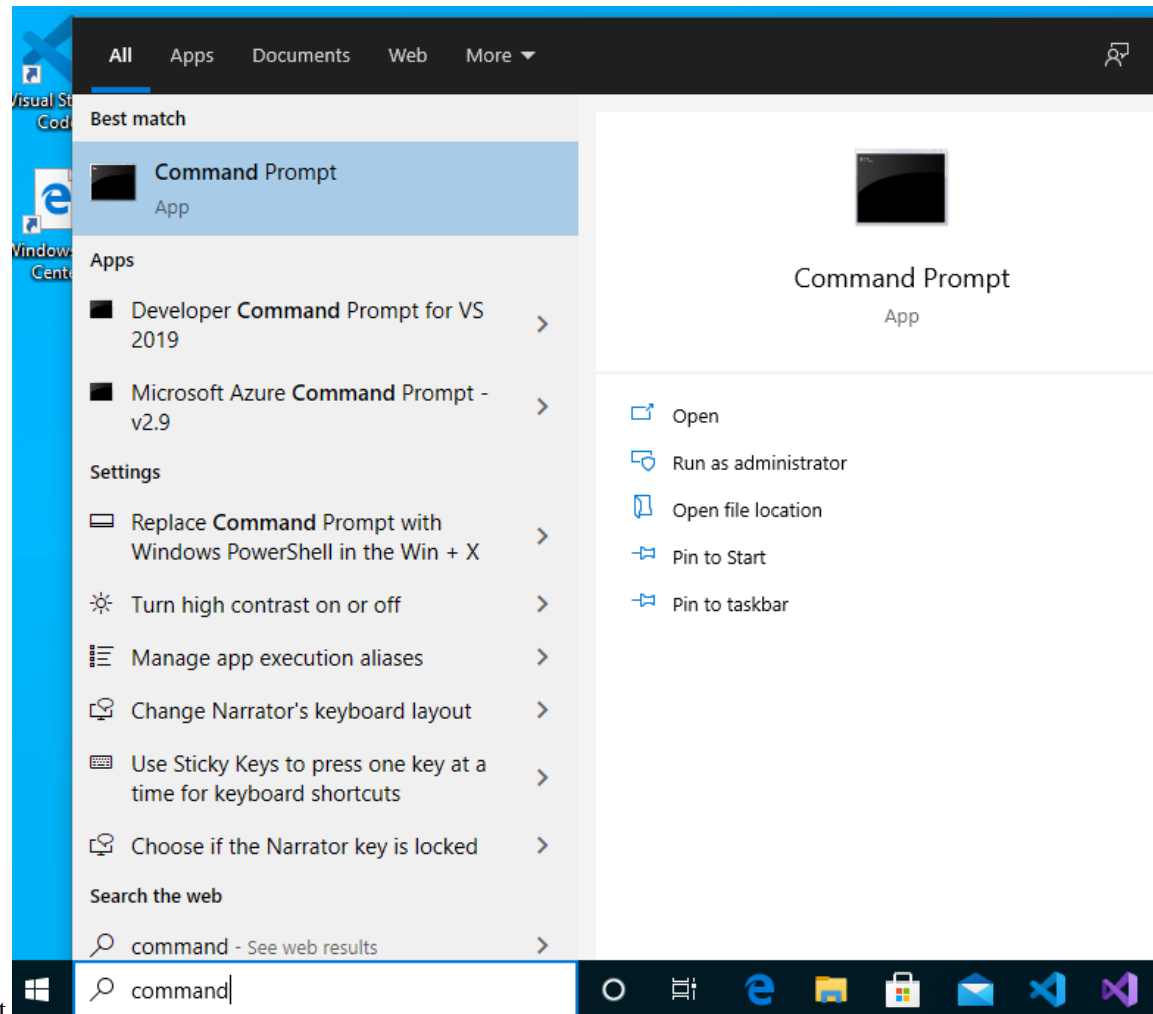
### Add Living Documentation

---

5 minutes

In this step you'll learn how to generate a living documentation from your test execution results so you can easily share them with your team.

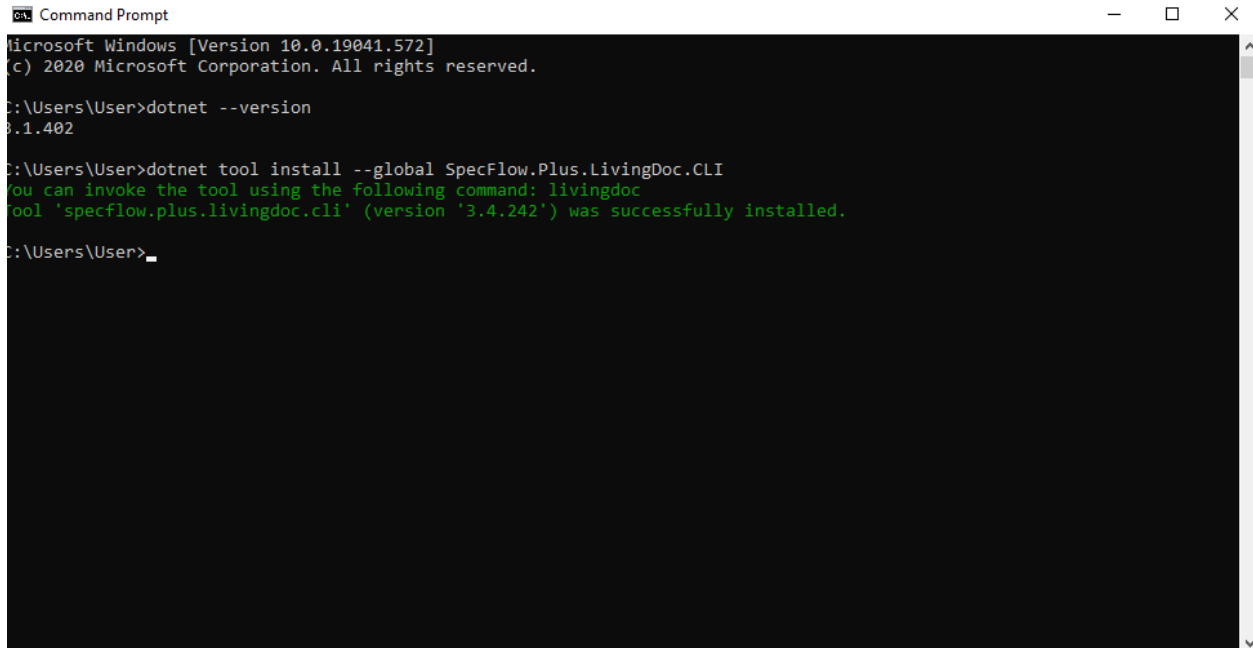
> **Note:** *If you have skipped the previous steps make sure your project tests have been executed before continuing with this step.*



1- Open a command prompt.

2- Install the LivingDoc CLI as a global dotnet tool.

```
dotnet tool install --global SpecFlow.Plus.LivingDoc.CLI
```



```
Command Prompt
Microsoft Windows [Version 10.0.19041.572]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\User>dotnet --version
3.1.402

C:\Users\User>dotnet tool install --global SpecFlow.Plus.LivingDoc.CLI
You can invoke the tool using the following command: livingdoc
Tool 'specflow.plus.livingdoc.cli' (version '3.4.242') was successfully installed.

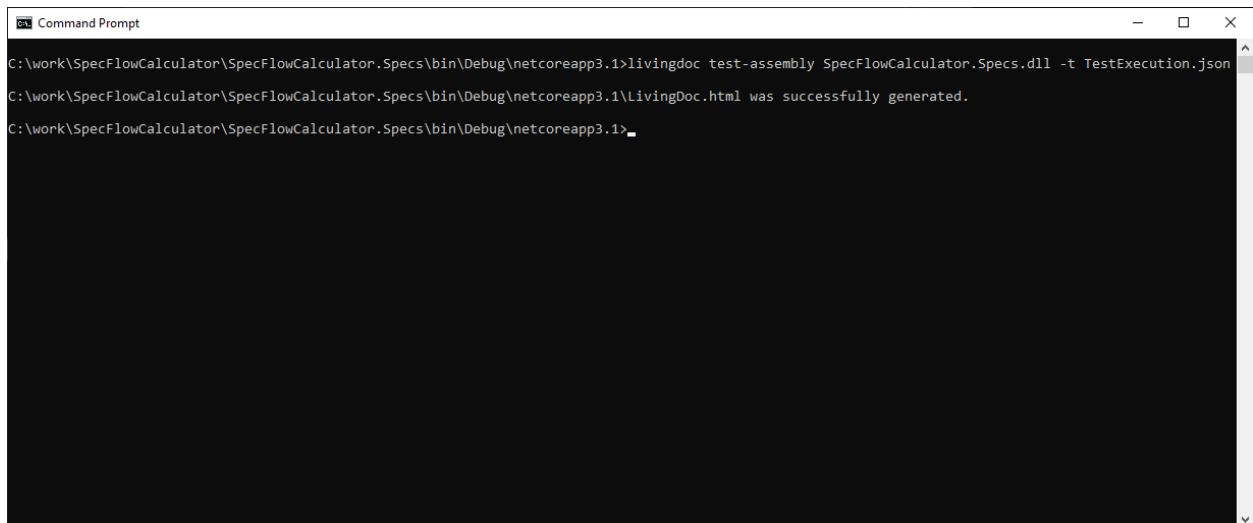
C:\Users\User>
```

3- Navigate to the **output directory of the SpecFlow project**. In this example the solution was setup in the `C:\work` folder.

```
cd C:\work\SpecFlowCalculator\SpecFlowCalculator.Specs\bin\Debug\netcoreapp3.1
```

4- Run the LivingDoc CLI by using the below command to generate the HTML report.

```
livingdoc test-assembly SpecFlowCalculator.Specs.dll -t TestExecution.json
```



```
Command Prompt
C:\work\SpecFlowCalculator\SpecFlowCalculator.Specs\bin\Debug\netcoreapp3.1>livingdoc test-assembly SpecFlowCalculator.Specs.dll -t TestExecution.json
C:\work\SpecFlowCalculator\SpecFlowCalculator.Specs\bin\Debug\netcoreapp3.1\LivingDoc.html was successfully generated.
C:\work\SpecFlowCalculator\SpecFlowCalculator.Specs\bin\Debug\netcoreapp3.1>
```

5- Open the generated HTML with your favorite browser. The HTML file is stored in the same folder as the **output directory of the SpecFlow project**.

```
C:\work\SpecFlowCalculator\SpecFlowCalculator.Specs\bin\Debug\netcoreapp3.1\LivingDoc.
↩html
```

*\*Note: if you run into issues here, e.g your JSON file name is FeatureData.JSON instead of TestExecution.JSON, this indicates you have an older version of the CLI tool. Please check our [migration guide here](#) to upgrade to the latest*

## Welcome to the Step-By-Step Getting Started Guide!

version.

Review the living documentation of the calculator features that you have implemented. Select the “Calculator” feature in the tree. On the right pane check the detailed description of the feature and the scenarios. You can also see the “green” test execution result of the scenarios and steps.

The screenshot shows a web browser window displaying the SpecFlowCalculator.Specs Living Documentation. The browser address bar shows the file path: C:/work/SpecFlowCalculator/SpecFlowCalculator.Specs/bin/Debug/netcoreapp3.1/LivingDoc.html#/document/Standalone/fea... The page title is "SpecFlowCalculator.Specs" and it was generated on Dec 1, 2020, 11:34 AM GMT+1. The interface has two tabs: "Living Documentation" (active) and "Analytics".

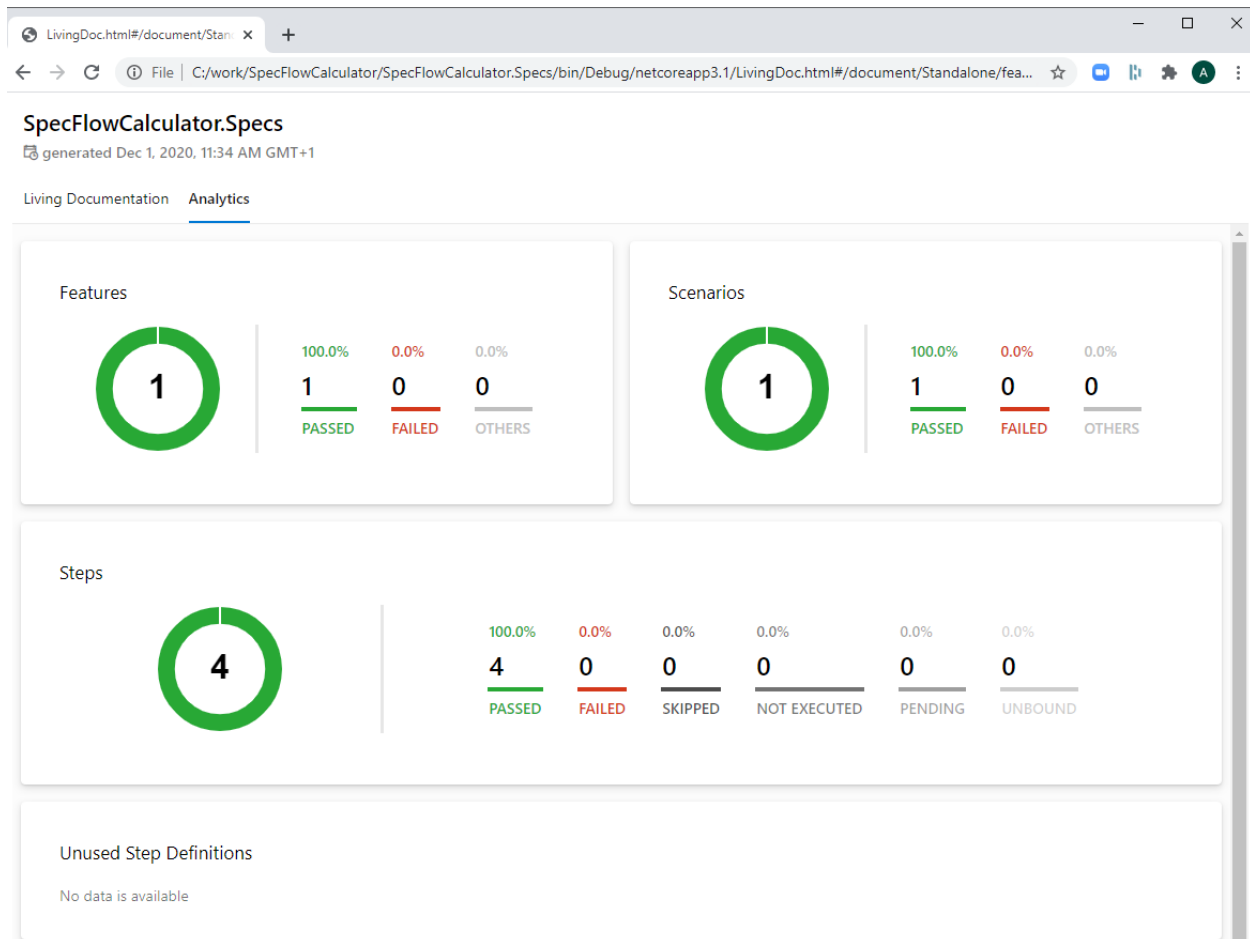
On the left side, there is a tree view under "SpecFlowCalculator.Specs". It shows a folder "Features" with a sub-item "Calculator" (marked with a green checkmark). Under "Calculator", there is a scenario "Add two numbers" (also marked with a green checkmark). Above the tree, there is a "Filter by Keyword" search bar and a "Test results" toggle switch.

On the right side, the details for the "Feature: Calculator" are shown. It includes an image of a calculator. Below the image, there is a text description: "In order to avoid silly mistakes  
As a math idiot  
I want to be told the **sum** of **two** numbers". There is a link to the feature: "Link to a feature: [Calculator](#)". Below this, there is a "Further read" link: "Further read: [Learn more about how to generate Living Documentation](#)".

Below the feature description, there is a section for the scenario "Add two numbers" (2s 122ms). It lists four steps, each with a green checkmark:

- ✓ **Given** the first number is 50
- ✓ **And** the second number is 70
- ✓ **When** the two numbers are added
- ✓ **Then** the result should be 120

Check the test result summary by clicking on the “Analytics” tab:



SpecFlow+LivingDoc is packed with great features that truly bring your documentation to life!

To read more about SpecFlow+LivingDoc and its features, please visit our dedicated [LivingDoc documentation page](#).



CONGRATULATIONS!



You have now successfully created and tested your first SpecFlow project.

We have put together a little exercise for you to test your newly acquired skills, [check it out here](#).

Check out our [examples](#) page if you are looking for additional sample projects. We have also put together a sample project using Selenium for UI automation, you can find it [here](#).

To keep up to date with the latest on SpecFlow [Join the SpecFlow Community](#).





## CHAPTER 10

---

### Install JetBrains Rider Plugin

---

10 minutes

In this step you'll learn how to install the SpecFlow for Rider plugin. SpecFlow's JetBrains Rider plugin not only enables the functionalities needed for testing automation, but is also bundled with several helpful features to make the journey more intuitive.

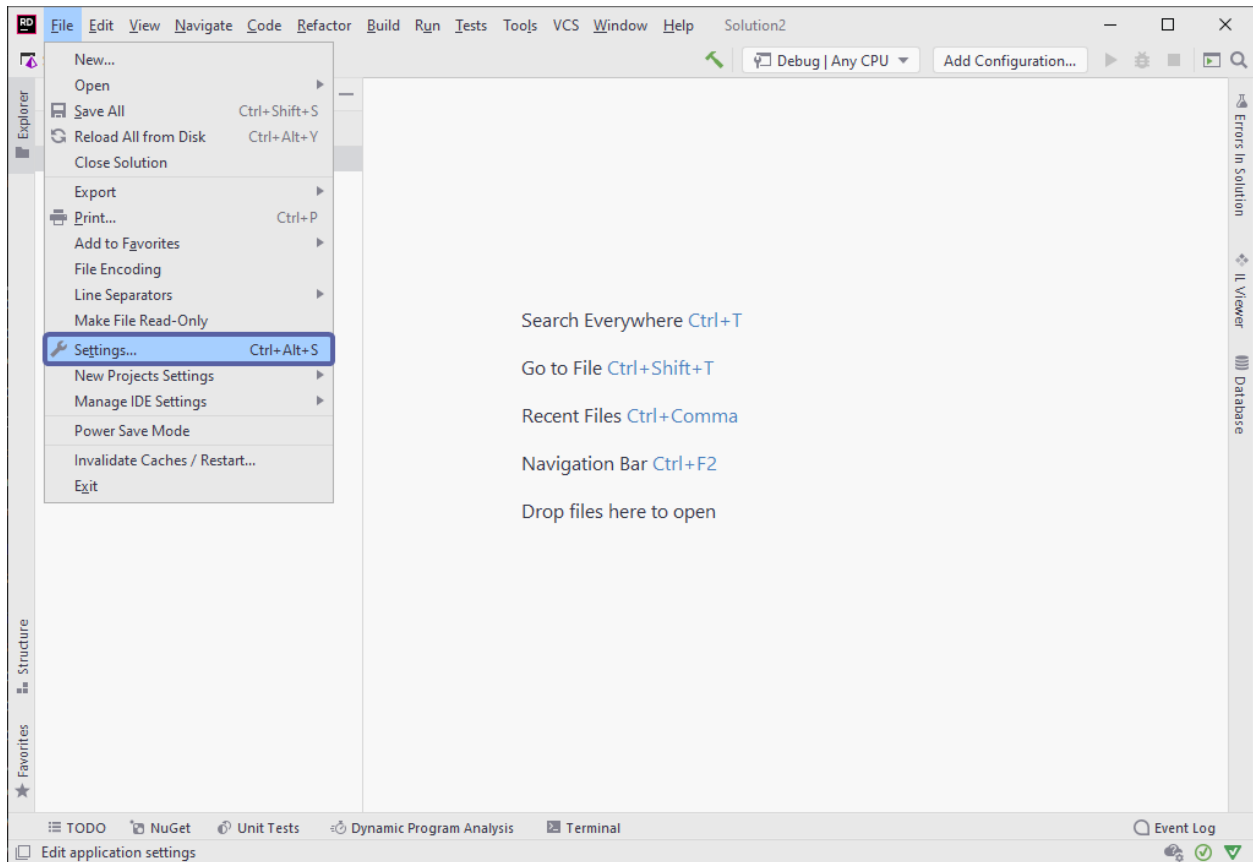
The plugin can be found either at the [JetBrains marketplace](#) or directly from within the Rider IDE.

To install the plugin directly from JetBrains Rider:

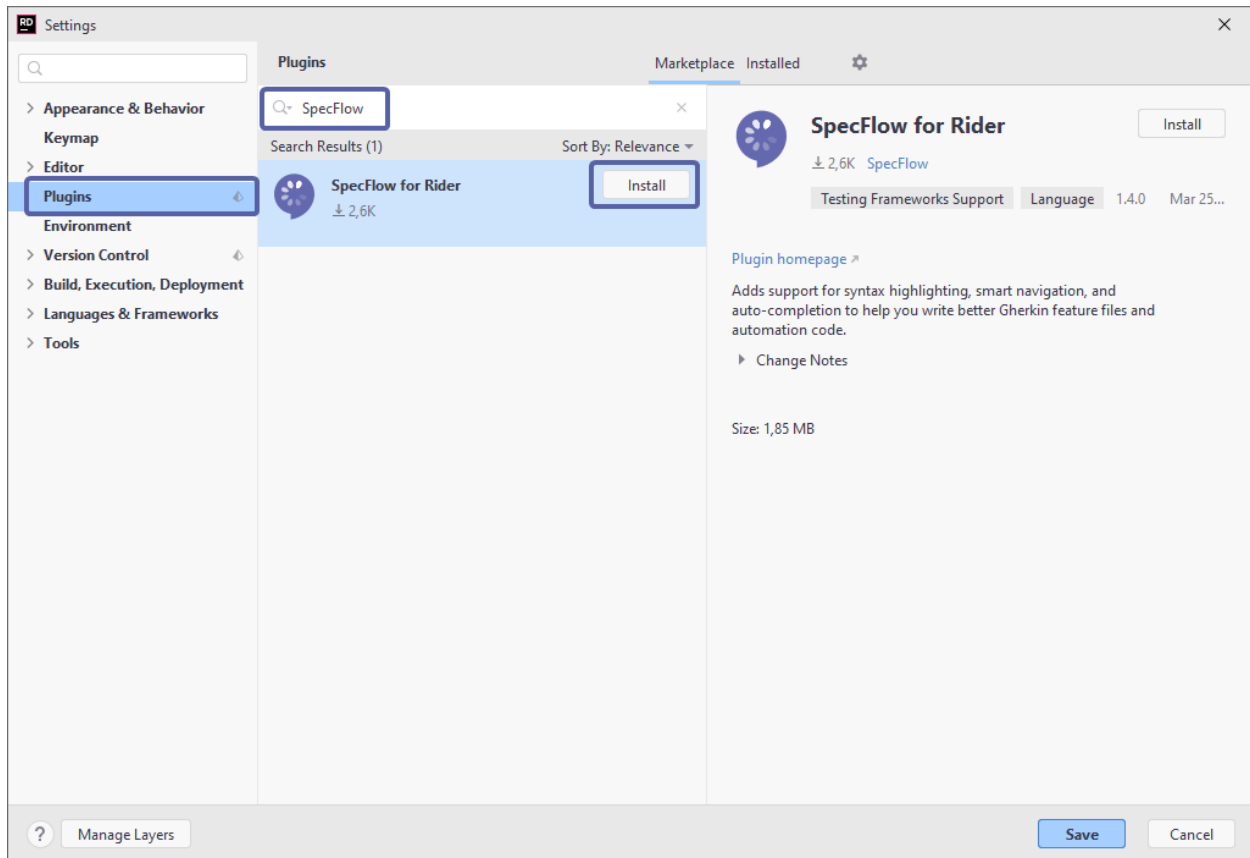
**1-** Open JetBrains Rider

**2-** Navigate to **File Settings Plugins** (**Ctrl+Alt+S**) and search for "SpecFlow" in the search bar:

## Welcome to the Step-By-Step Getting Started Guide!

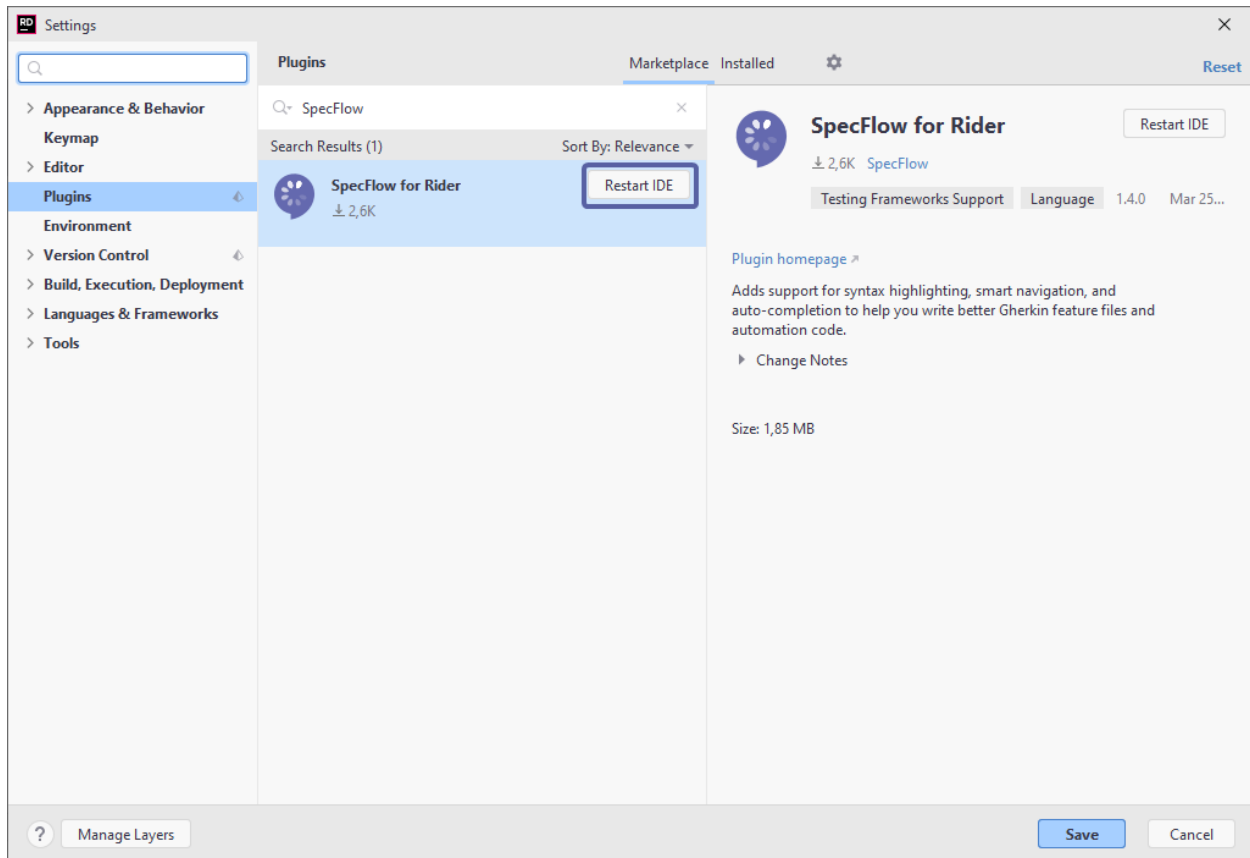


3- Hit **Install** and then **Accept** when prompted with the privacy note. You can find our privacy policy [here](#)



4- You are then required to restart the Rider IDE, hit **Restart**:

## Welcome to the Step-By-Step Getting Started Guide!



The installation is now finished. In the next step you'll create a simple application that will be used throughout this guide.

## CHAPTER 11

---

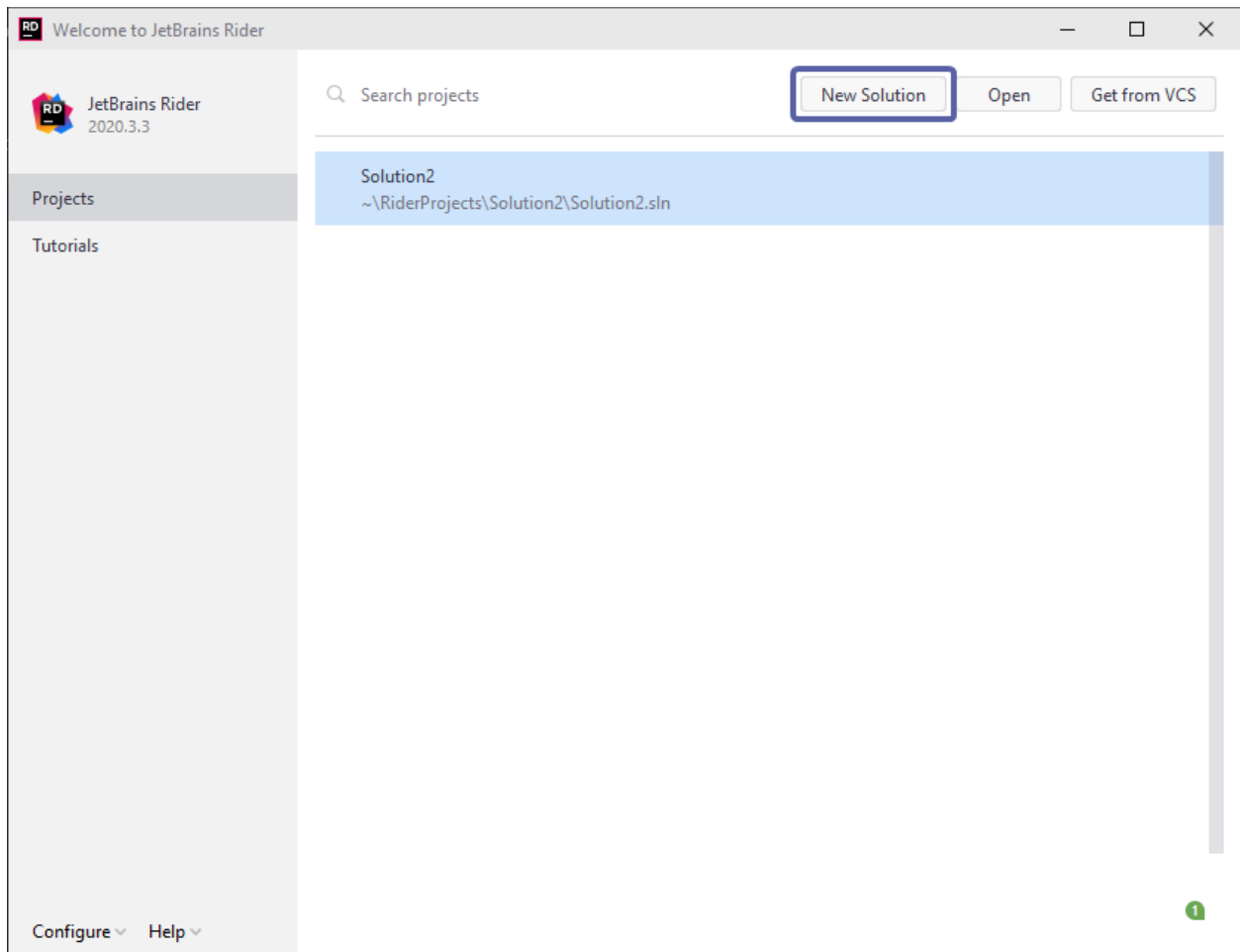
### Create calculator project

---

10 minutes

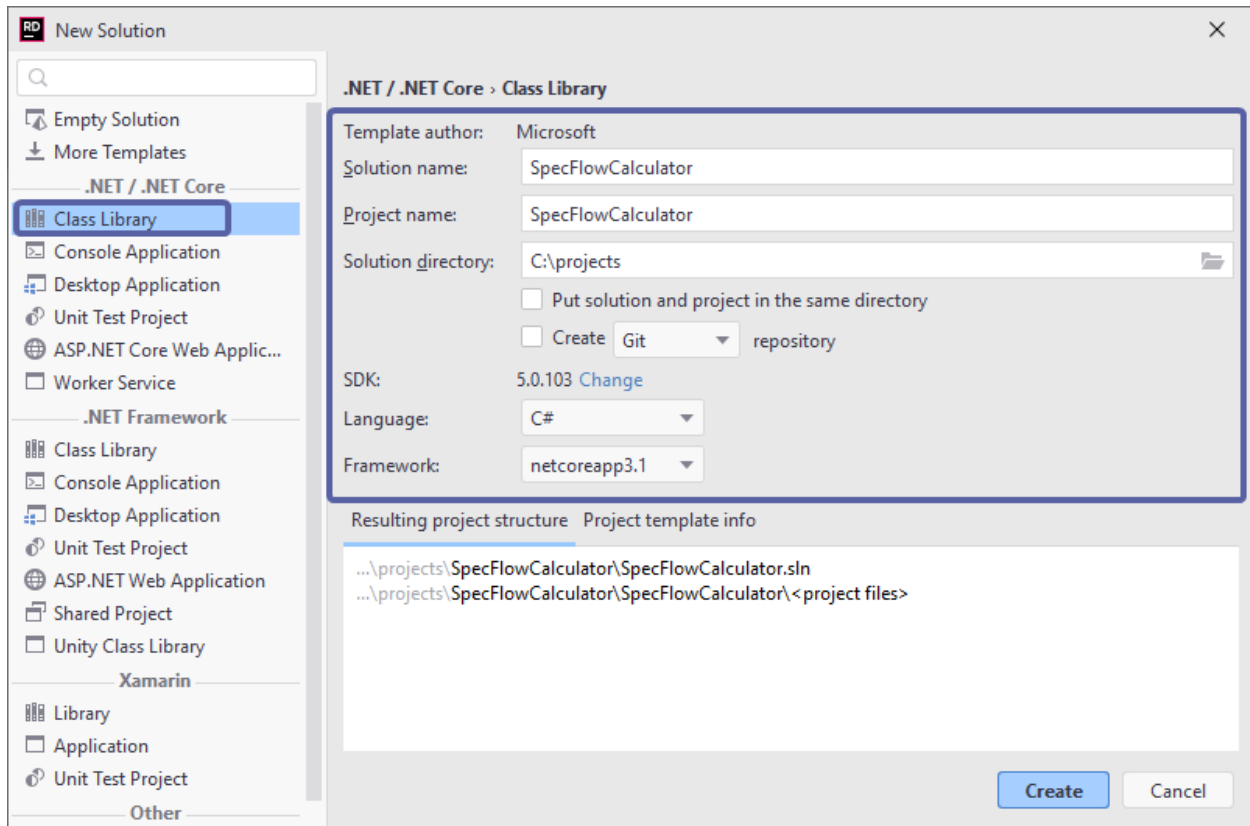
In this step you'll create the application that will be tested, also called System Under Test (SUT). The application will be a simple calculator in a *C#* class library.

**1-** Open JetBrains Rider and create a new *C#* class library by selecting “New Solution” from the startup dialog:



2- Select “Class library” and use the below configurations and click **Create**.

- Solution & Project name: SpecFlowCalculator
- Solution directory: *\*choose a location to save the project* - in this example the solution is saved to C:\projects
- Language: C#
- Framework: netcoreapp3.1



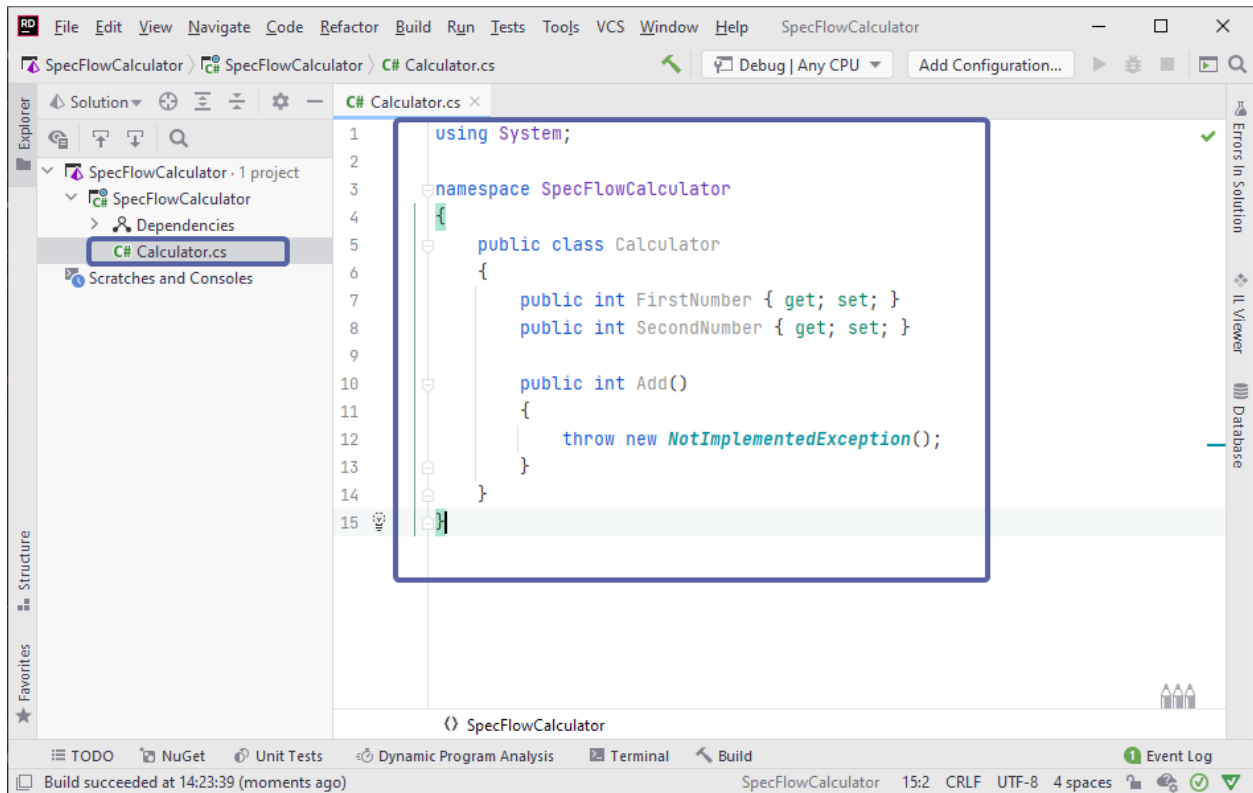
3- Rename `Class1.cs` to `Calculator.cs` and overwrite the content with the following code :

```
using System;

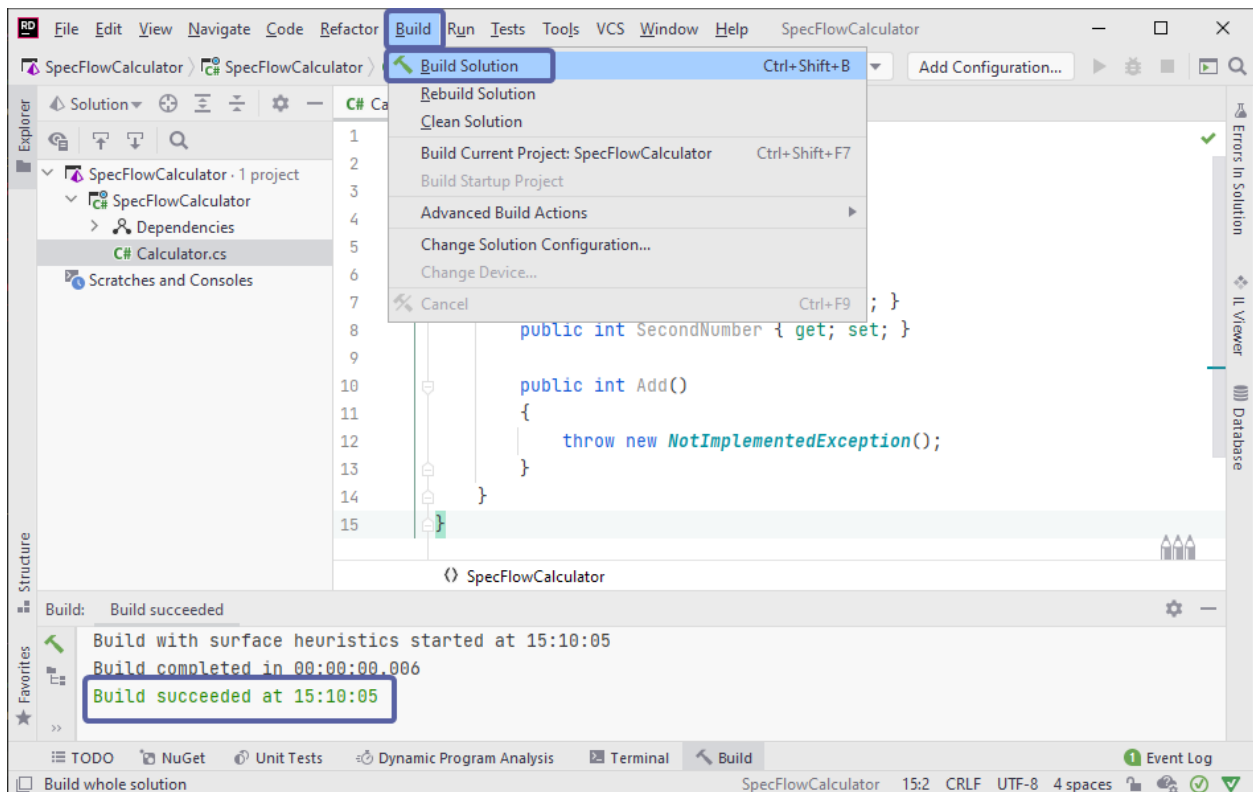
namespace SpecFlowCalculator
{
    public class Calculator
    {
        public int FirstNumber { get; set; }
        public int SecondNumber { get; set; }

        public int Add()
        {
            throw new NotImplementedException();
        }
    }
}
```

## Welcome to the Step-By-Step Getting Started Guide!



4- Now build the solution by navigating to “Build Build Solution”. You should see a “Build Succeeded” message in the output window:





The calculator application is now built. In the next step you'll learn how to create a SpecFlow project.



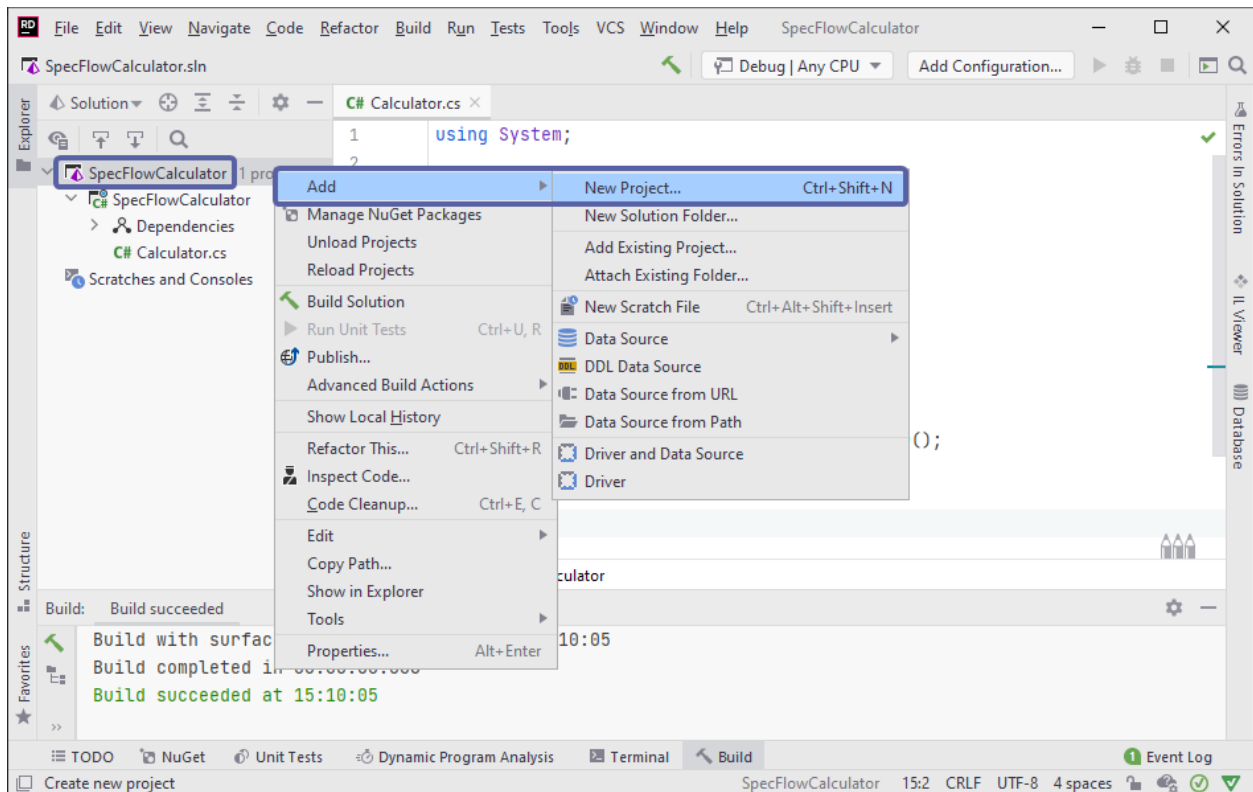
## CHAPTER 12

### Create SpecFlow project

5 minutes

In this step you'll create a SpecFlow project and add it to the existing calculator solution.

**1- Right-click** the solution item “*SpecFlowCalculator* (1 of 1 project)” under the Solution Explorer and select the “Add New Project” menu item.

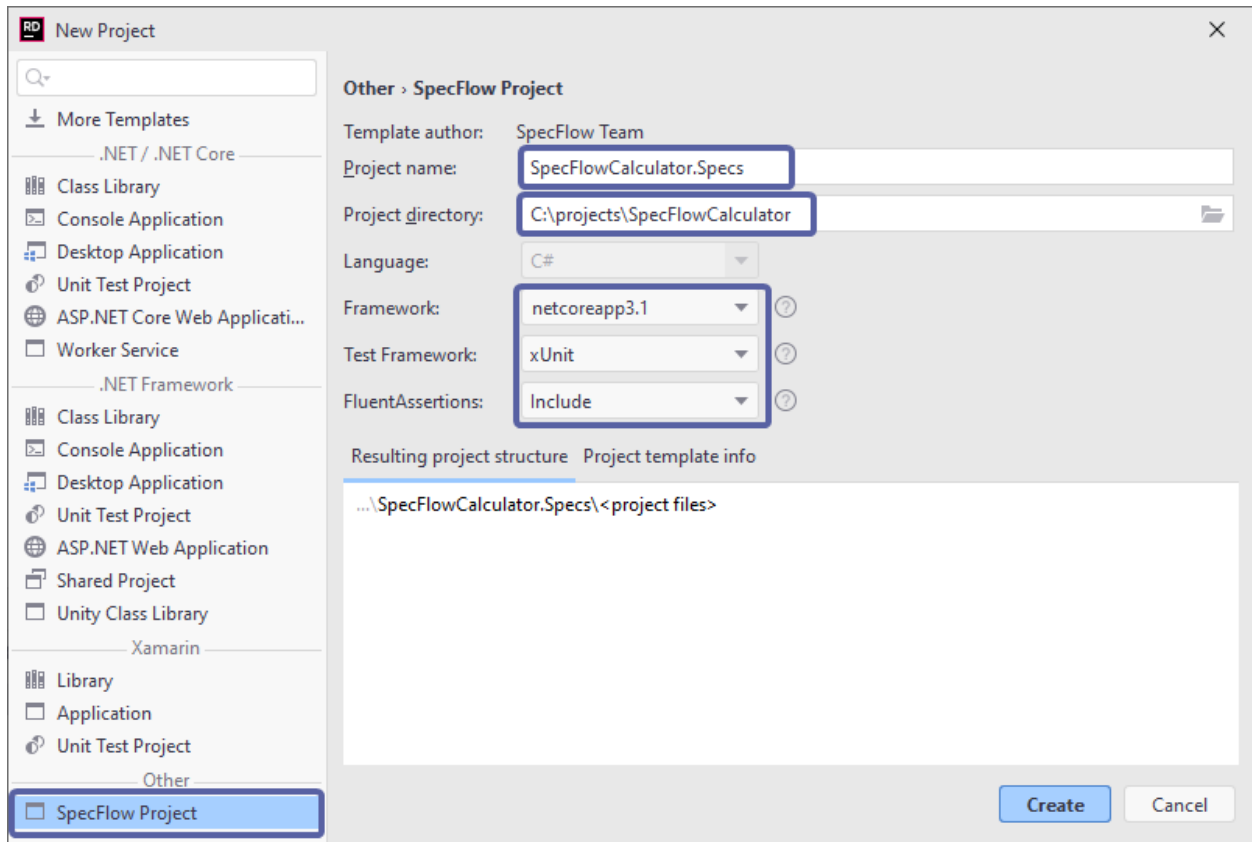


## Welcome to the Step-By-Step Getting Started Guide!

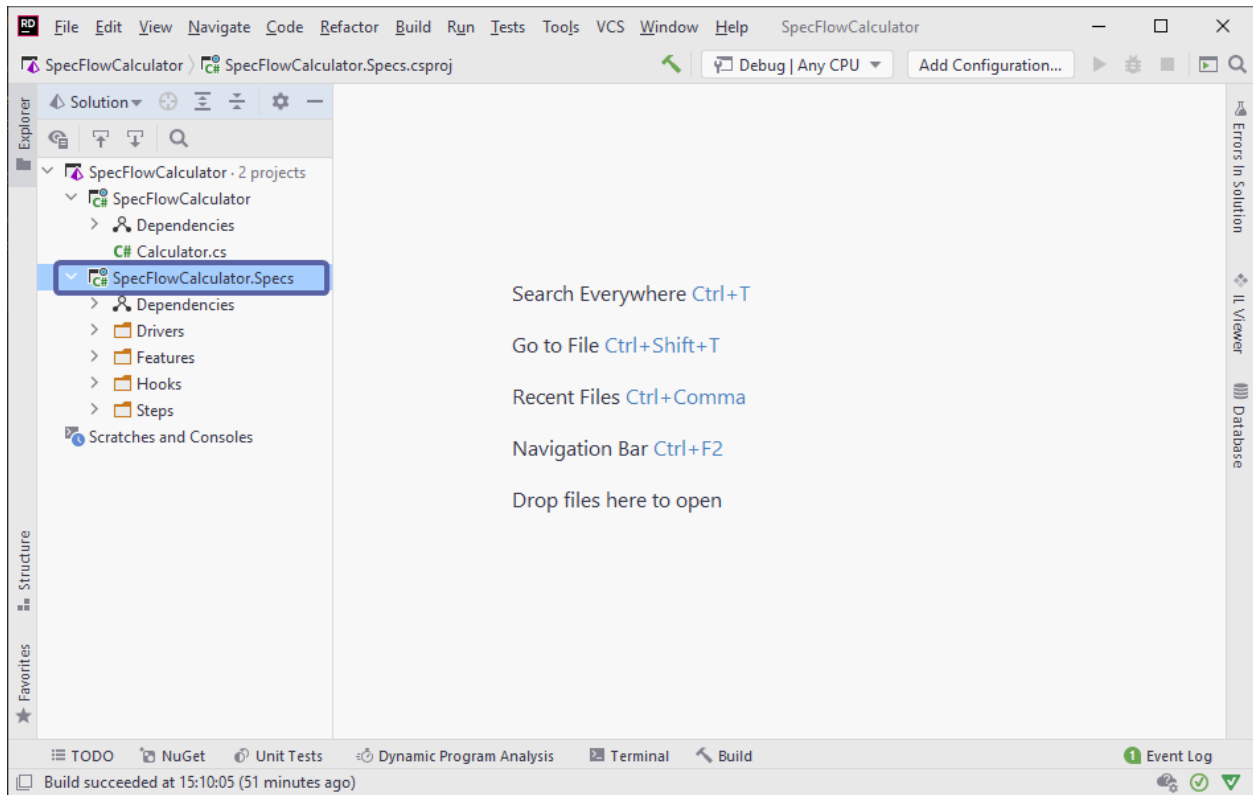
2- Click on *SpecFlow Project Template* under the *Other* category, enter the project name as “**SpecFlowCalculator.Specs**”, keep the suggested location (the solution folder), pick *xUnit* as the Test Framework and hit *Create*:

Note: If you cannot see SpecFlow Project Template, ensure you have SpecFlow for Rider Plugin 1.6.0 or higher installed. (Only compatible with Rider 2021.1 or higher)

Note: Currently running the tests from the feature files is only possible with **xUnit** and **NUnit**.\*



3- JetBrains Rider will now create the new project, you should see the new SpecFlow project in the Solution Explorer as per below:



In the next step you will learn how to add a project reference and how to use the test explorer.



---

### Create SpecFlow project - Continue

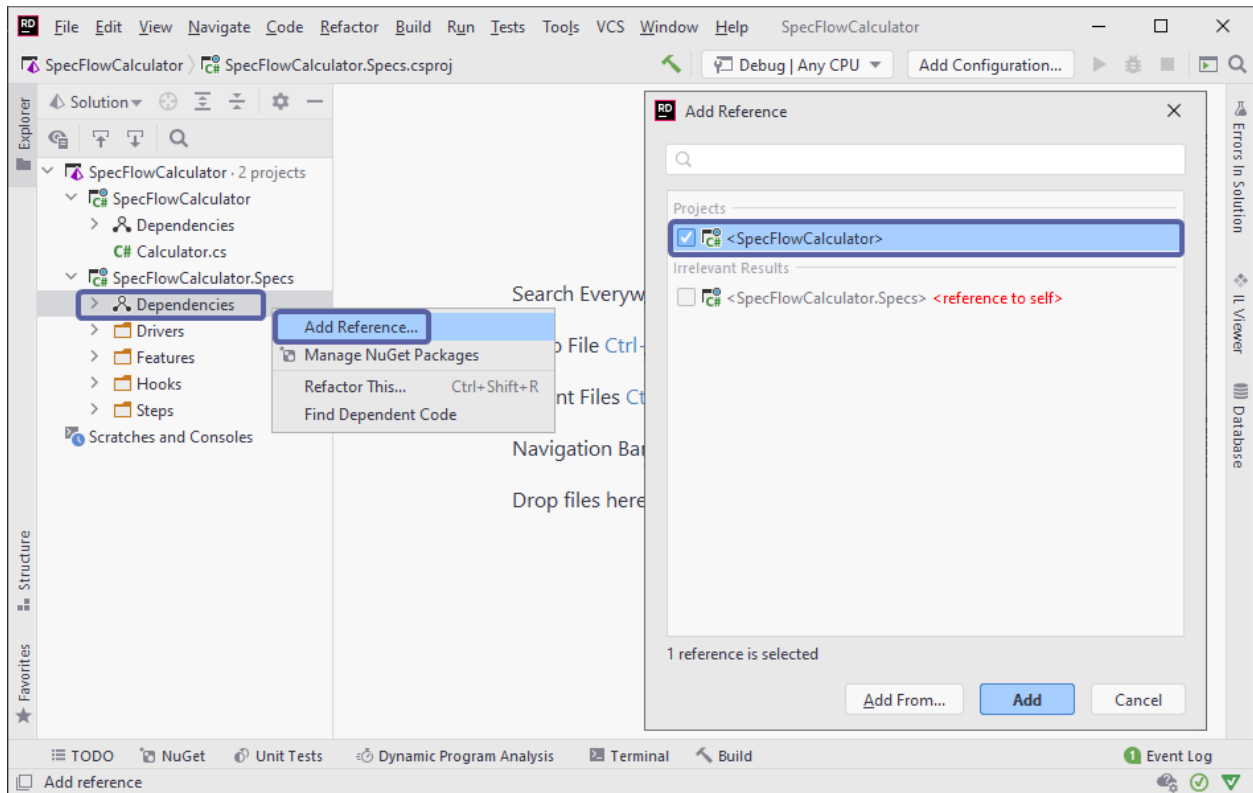
---

5 minutes

You will now add a project reference to the “SpecFlowCalculator” class library in the newly created SpecFlow project. This is necessary because we want to test the “Calculator” class implemented in the class library in the “SpecFlow-Calculator.Specs” project. To do this, follow the below steps:

- 1- Expand the project node “*SpecFlowCalculator.Specs*” in the Solution Explorer, right-click the “*Dependencies*” node and select the “*Add Reference...*” menu item.
- 2- In the “Add Reference” dialog check the “SpecFlowCalculator” class library and click **Add**.

## Welcome to the Step-By-Step Getting Started Guide!



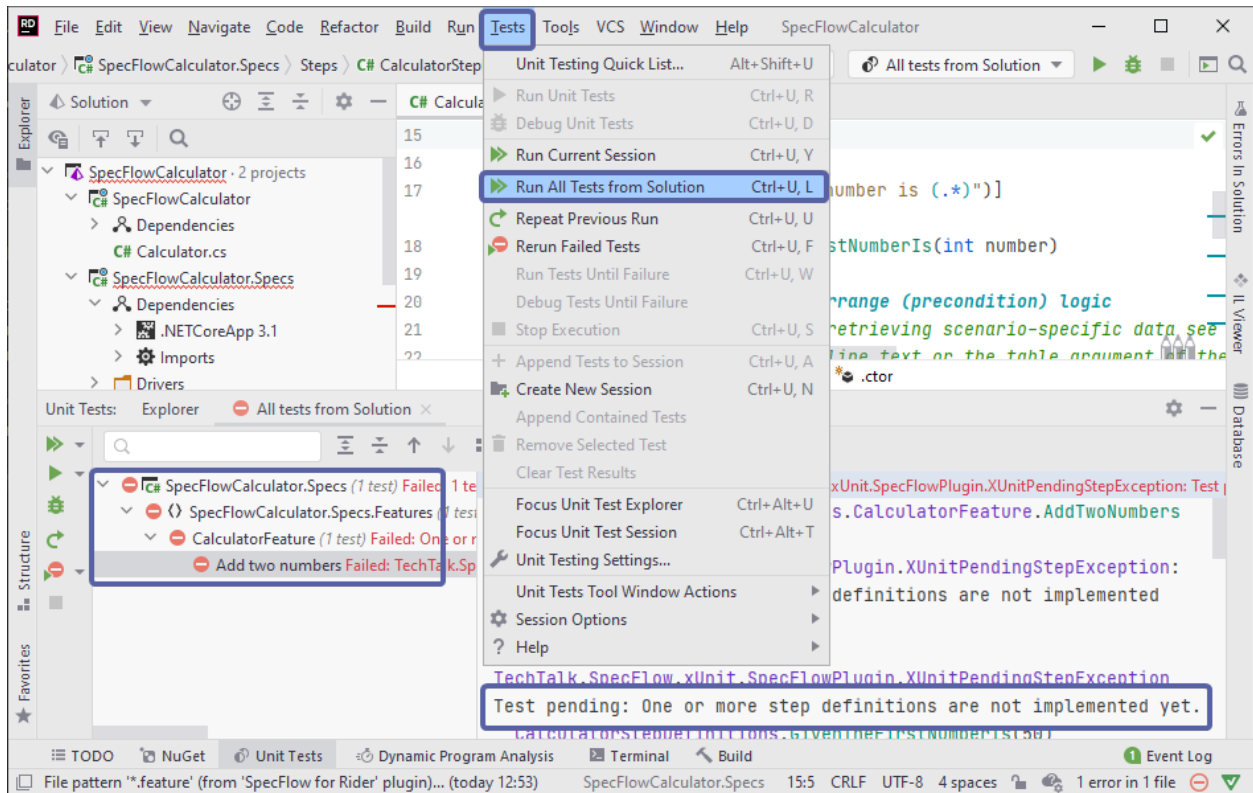
Now the solution is set up with a class library containing the implementation of the calculator and a SpecFlow project that contains the specification and tests of the calculator.

3- Now build the solution. You should see the “Build Succeeded” message in the output window.

4- Run all the tests by navigating to “Tests Run All Tests from Solution”:

> Note: The red underline applied to the project name and feature file in the explorer pane is a known xUnit bug in Rider and does **NOT** indicate an error.





The tests would fail as expected as our step definitions are not yet implemented.

In the next step you will learn how to automate your first scenario and implement the step definitions.



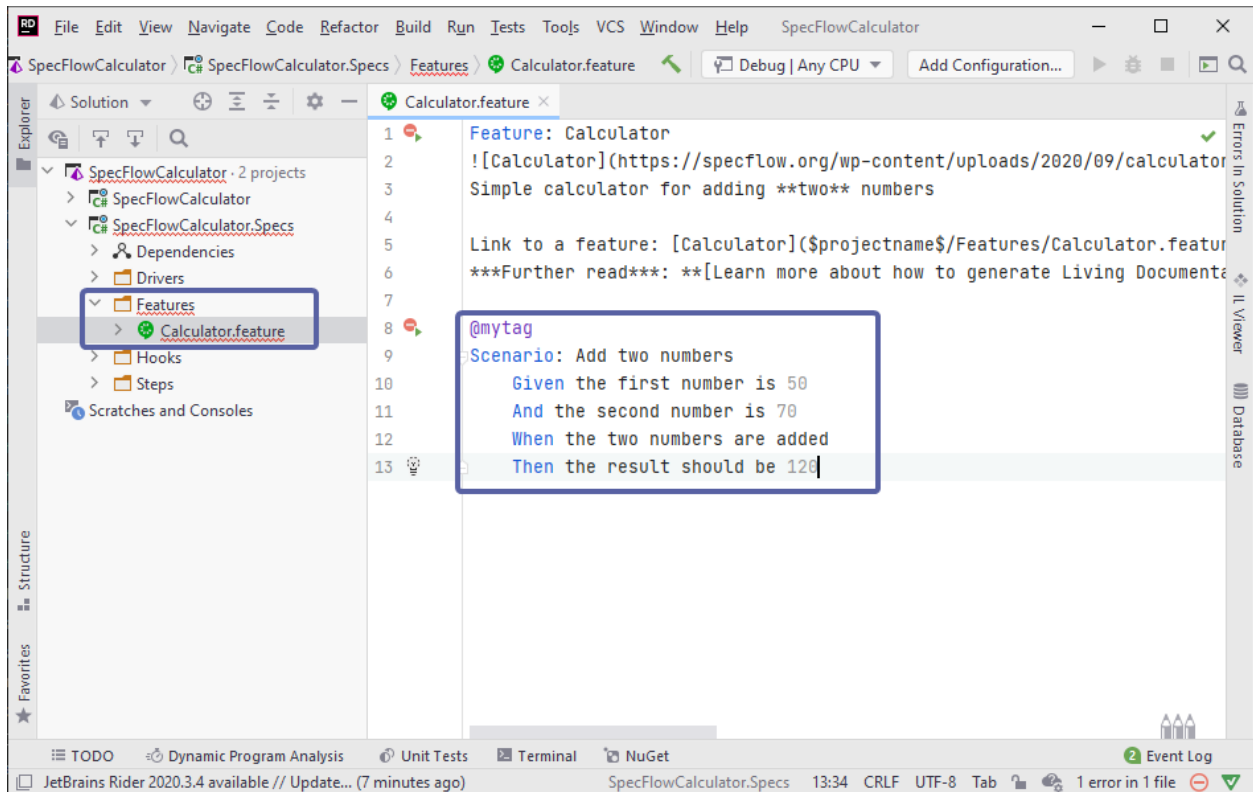
## CHAPTER 14

### Bind the first step

10 minutes

In this step you'll bind your first step (automate your first scenario step with SpecFlow).

1- Open the `Calculator.feature` file by double-clicking it in the Solution Explorer (SpecFlowCalculator.Specs Features Calculator.feature)



The purpose of this feature file is to document the expected behavior of the calculator in a way that it is both human-

## Welcome to the Step-By-Step Getting Started Guide!

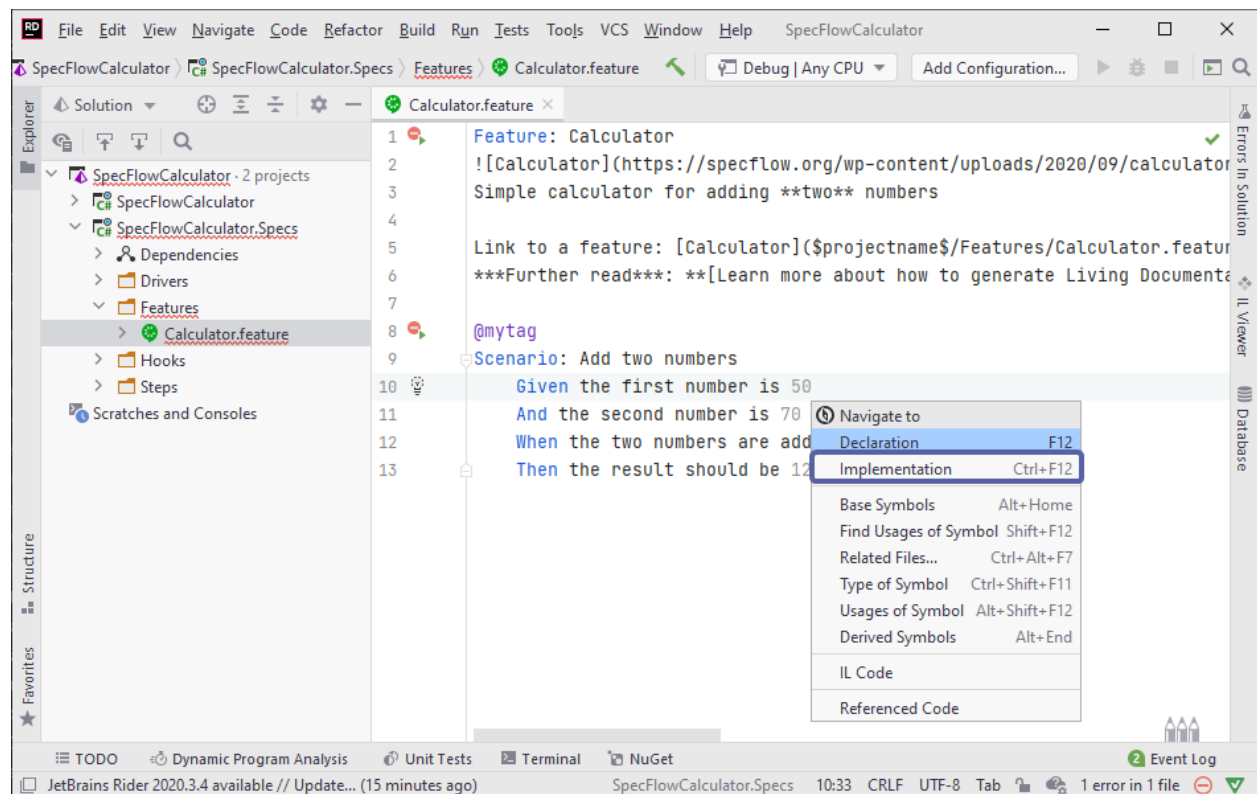
readable and suitable for test automation. SpecFlow uses the Gherkin language where you can phrase the scenarios using *Given/When/Then* steps. Currently there is a single scenario (automatically added by the SpecFlow project template) that describes how adding two numbers should work with the calculator.

Here is a closer look at the Gherkin scenario used in this template:

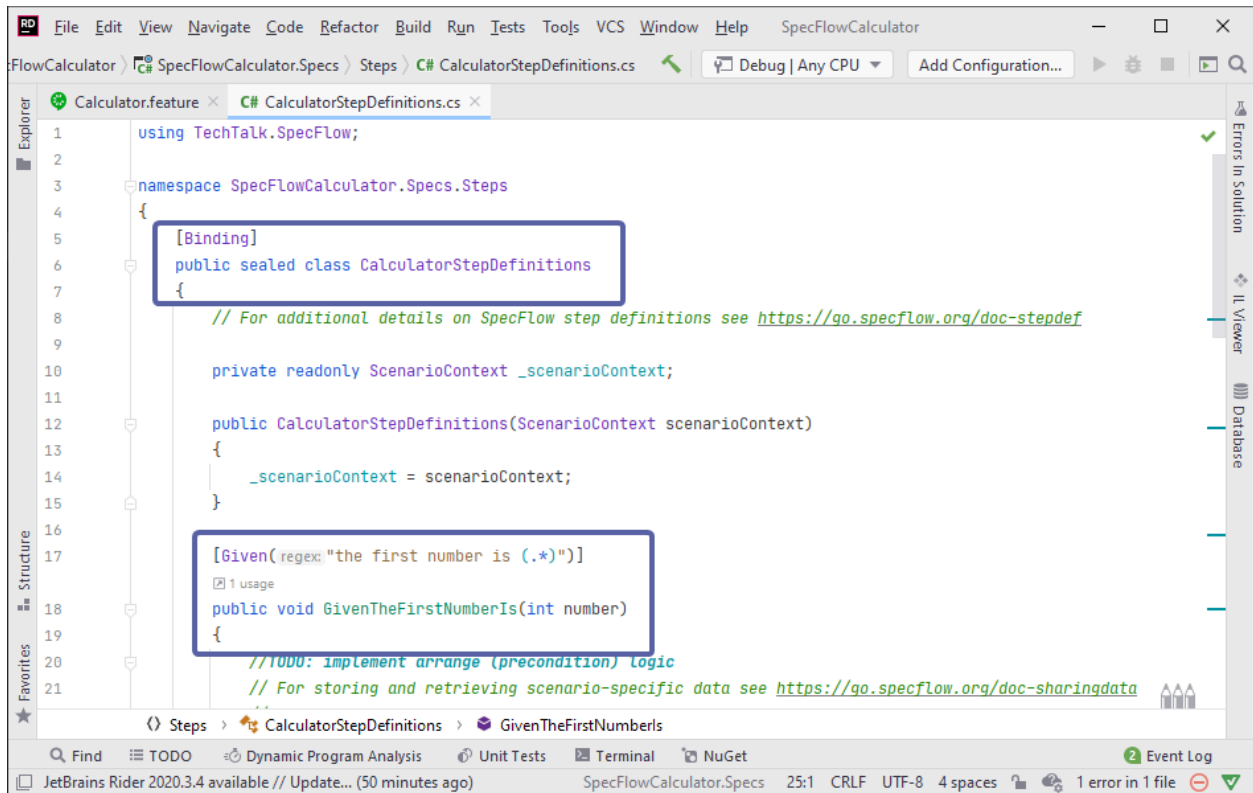
```
Scenario: Add two numbers
  Given the first number is 50
  And the second number is 70
  When the two numbers are added
  Then the result should be 120
```

Based on the scenario text, SpecFlow generates an automated test that executes the scenario. However, it is **not yet defined** what the steps of the scenario should actually “do”.

2- Right-click the first *Given* step “Given the first number is 50” and select “Go To → Implementation” or use the “Ctrl + F12” shortcut.



The SpecFlow plugin locates the step definition (binding) that belongs to this step. In this example, it opens the `CalculatorStepDefinitions` class and jumps to the `GivenTheFirstNumberIs` method.



*\*The step definition is located based on the [Binding] attribute on the class and the [Given] attribute on the method. The regular expression of the Given attribute matches the text of the scenario step.*

**3-** Add the below field to the class to instantiate the calculator that we want to test and created in [Step 2](#) of this guide (SUT).

```
private readonly Calculator _calculator = new Calculator();
```

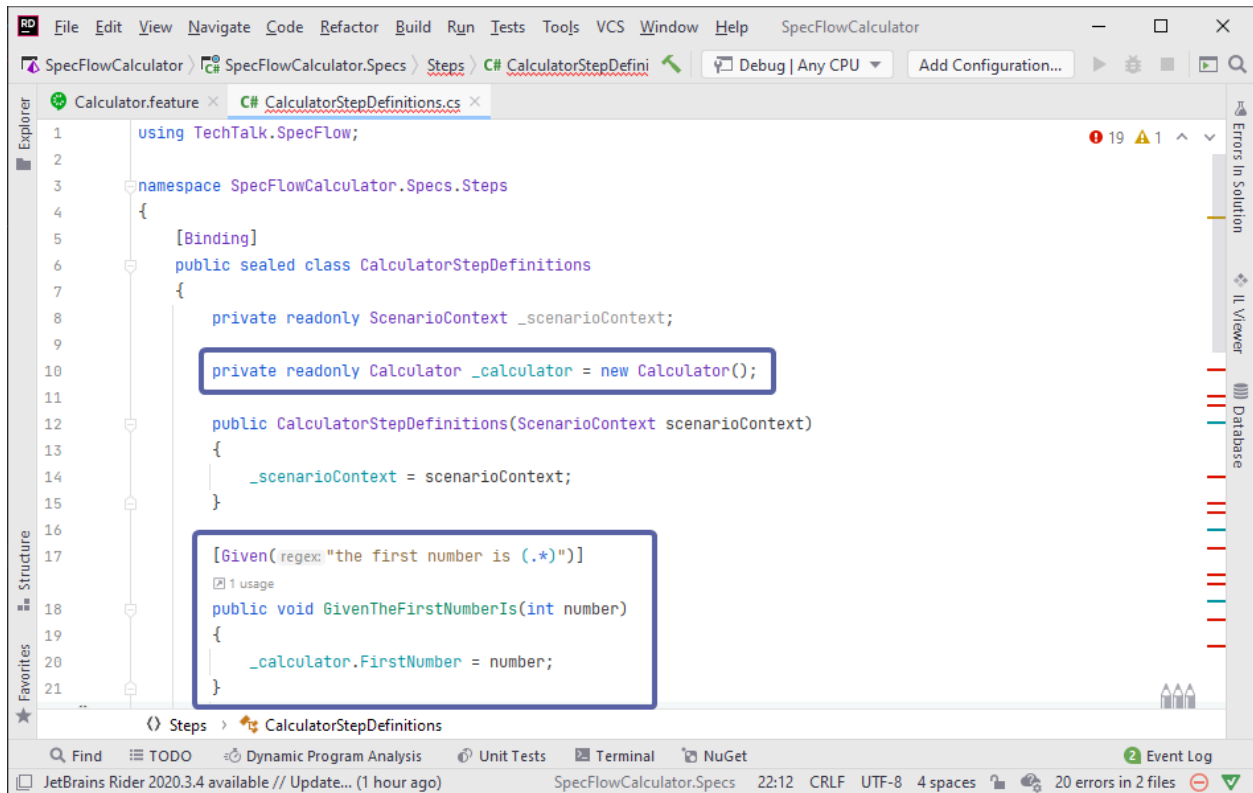
**4-** Replace the implementation of the first step definition method with the below code which sets the first number of the calculator.

```

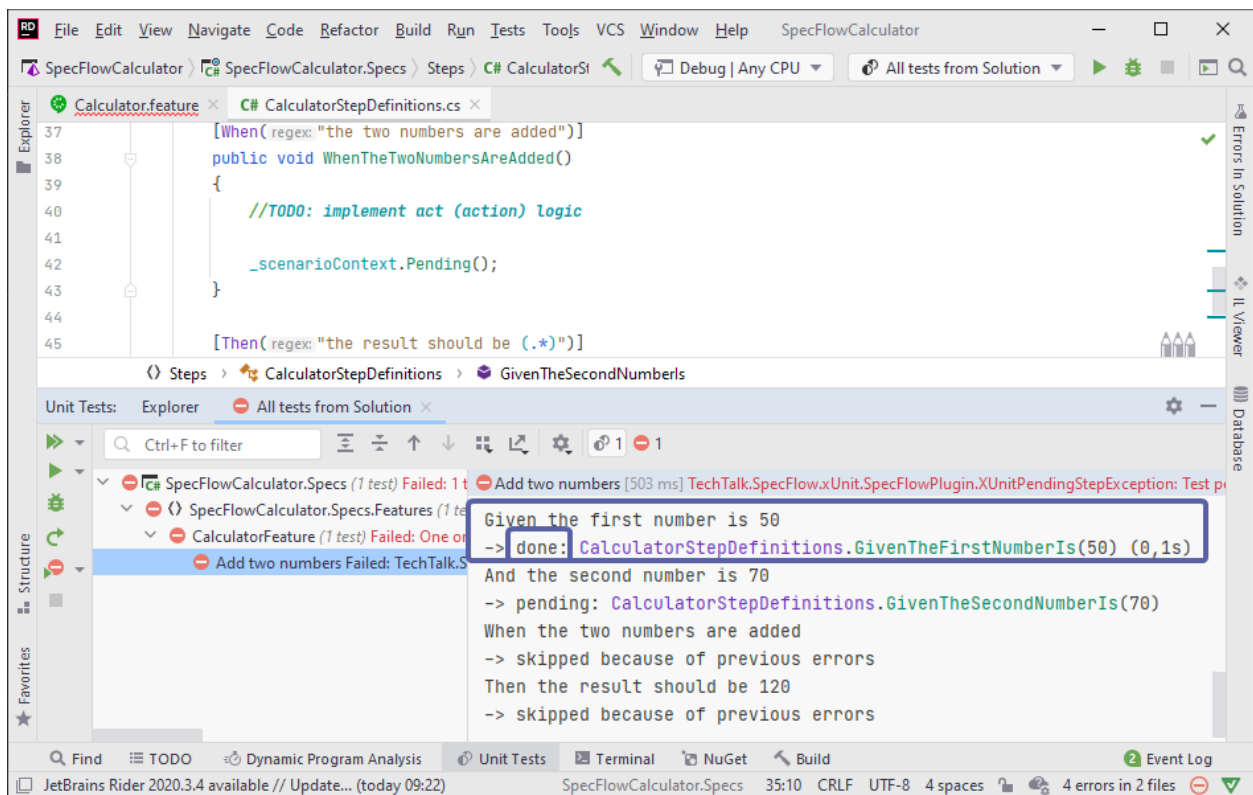
[Given("the first number is (.*)")]
public void GivenTheFirstNumberIs(int number)
{
    _calculator.FirstNumber = number;
}

```

## Welcome to the Step-By-Step Getting Started Guide!



5- Execute the test in the Test Explorer and open the text explorer output to see the details. You can see the “done” status here indicating the first step “Given the first number is 50” has been matched to the step definition method as per above binding. The remaining steps are yet to be implemented and are in “pending” and “skipped” status as expected.



In the next step you will bind the rest of the scenario steps.





## CHAPTER 15

---

### Bind remaining steps

---

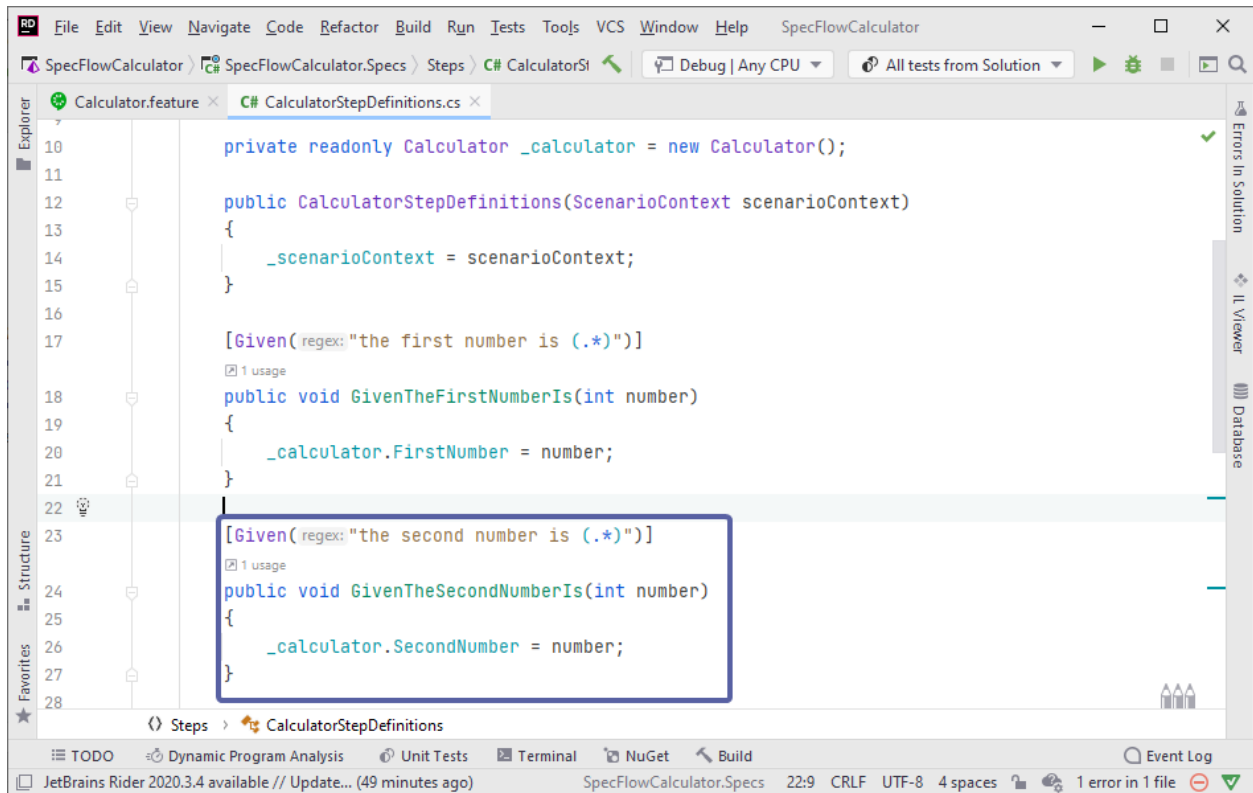
10 minutes

In this step you'll bind the remaining steps of the scenario.

- 1- Similar to the previous step, Right-click the second *Given* step “And the second number is 70” and select “Go To → Implementation” or use the “Ctrl + F12” shortcut.
- 2- Implement the binding of the second step “And the second number is 70” by replacing the code of the `GivenTheSecondNumberIs` method with the below:

```
[Given("the second number is (.*)")]
public void GivenTheSecondNumberIs(int number)
{
    _calculator.SecondNumber = number;
}
```

## Welcome to the Step-By-Step Getting Started Guide!

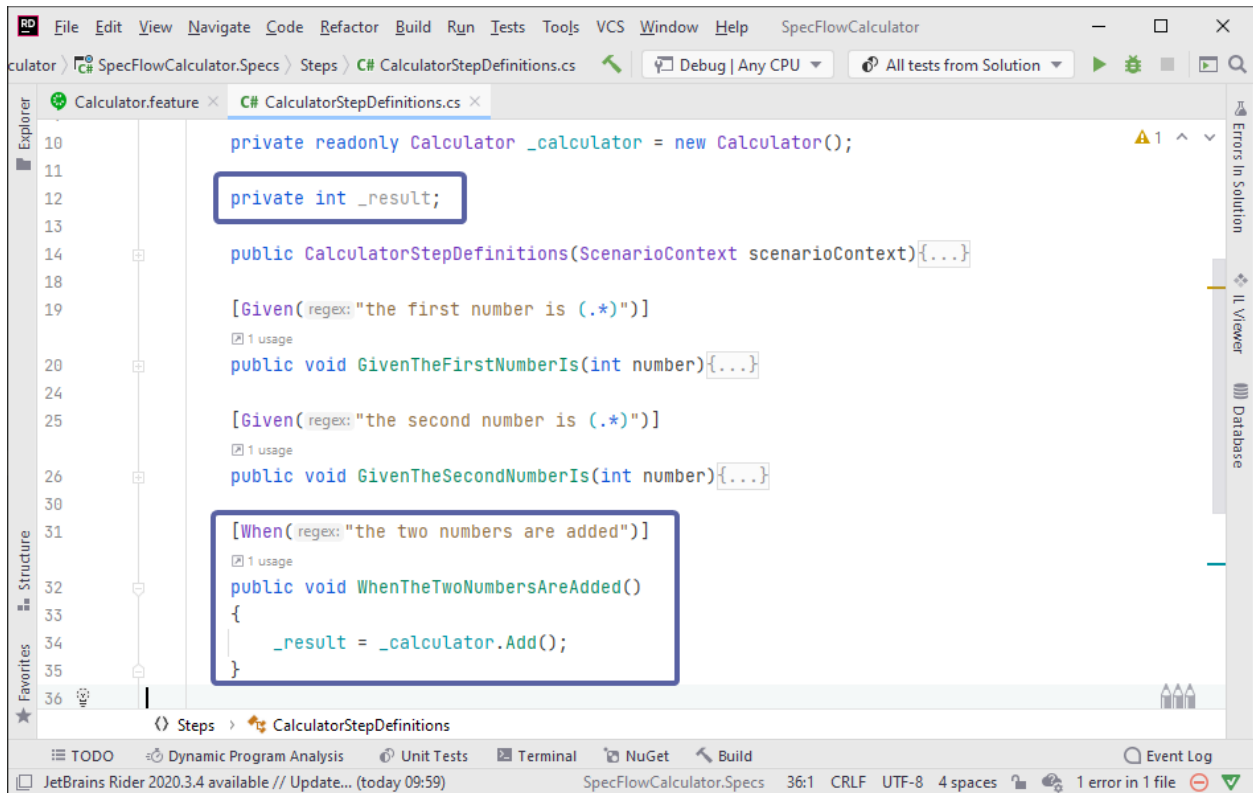


> Note: We use the “And” keyword in the Gherkin scenario for better readability. The “And” keyword will be interpreted as “Given”, “When” or “Then” depending on the previous step(s) in the scenario. In this example the “And the second number is 70” is interpreted as a “Given” step because the previous step is a “Given” step.

3- Next, implement the binding of the third step, “When the two numbers are added”, by replacing the code of the `WhenTheTwoNumbersAreAdded` method with the below. The method must have a `When` attribute, as it belongs to the “When” step in the scenario.

```
private int _result;
```

```
[When("the two numbers are added")]
public void WhenTheTwoNumbersAreAdded()
{
    _result = _calculator.Add();
}
```



This implementation calls the Add method of the calculator. Note that the result of the addition is not stored by the calculator in a property/field but it is returned to the caller. It's a good idea to store the returned value in a field so that we can work with the result afterwards.

4- Implement the binding of the last step, “Then the result should be 120”, by replacing the code of the ThenTheResultShouldBe method. The method must have a Then attribute, as it belongs to a “Then” step in the scenario.

Add a namespace using for xUnit at the top of the file:

```
using Xunit;
```

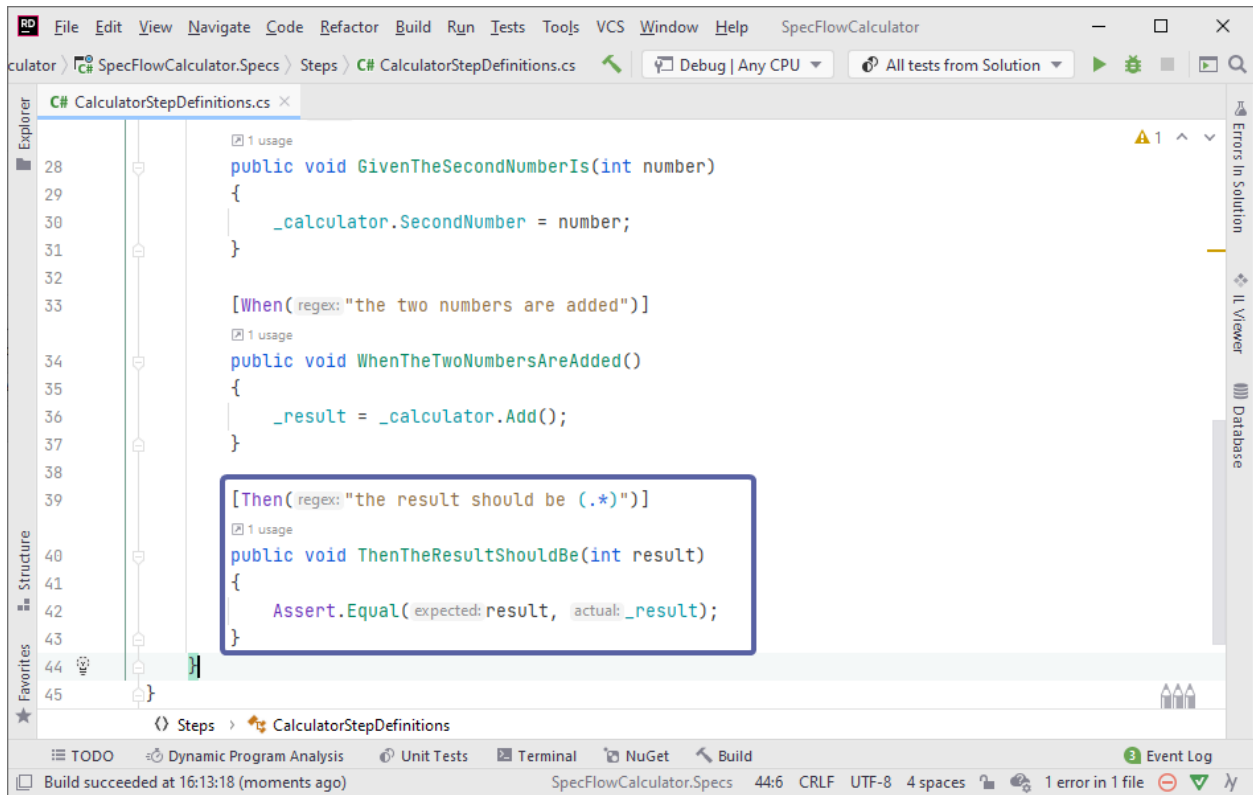
Use the below code for implementation of the “Then” step which validates if the result of the addition matches the expected value.

```

[Then("the result should be (.*)")]
public void ThenTheResultShouldBe(int result)
{
    Assert.Equal(result, _result);
}

```

## Welcome to the Step-By-Step Getting Started Guide!



After implementing all step definitions and cleaning up the file you should have the following code:

```
using TechTalk.SpecFlow;
using Xunit;

namespace SpecFlowCalculator.Specs.Steps
{
    [Binding]
    public sealed class CalculatorStepDefinitions
    {
        private readonly ScenarioContext _scenarioContext;

        private readonly Calculator _calculator = new Calculator();

        private int _result;

        public CalculatorStepDefinitions(ScenarioContext scenarioContext)
        {
            _scenarioContext = scenarioContext;
        }

        [Given("the first number is (.*)")]
        public void GivenTheFirstNumberIs(int number)
        {
            _calculator.FirstNumber = number;
        }

        [Given("the second number is (.*)")]
        public void GivenTheSecondNumberIs(int number)
```

(continues on next page)

(continued from previous page)

```

    {
        _calculator.SecondNumber = number;
    }

    [When("the two numbers are added")]
    public void WhenTheTwoNumbersAreAdded()
    {
        _result = _calculator.Add();
    }

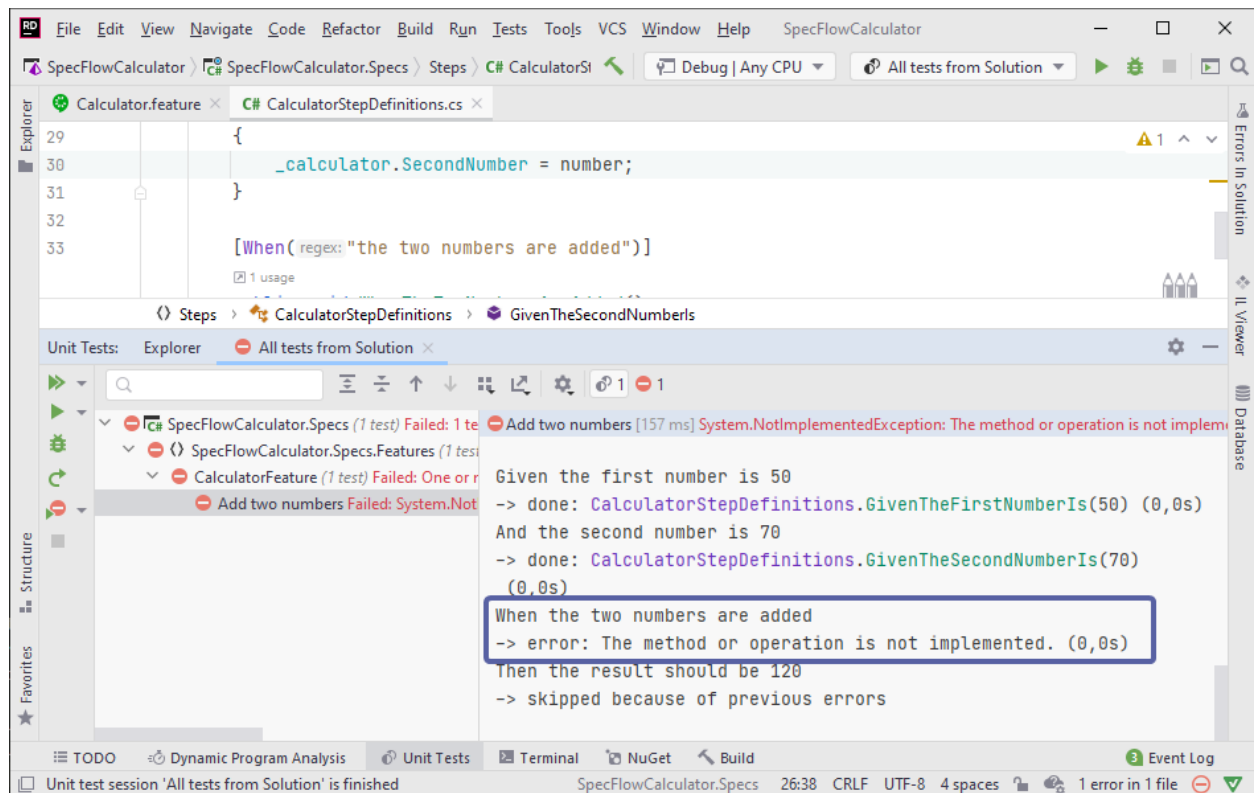
    [Then("the result should be (.*)")]
    public void ThenTheResultShouldBe(int result)
    {
        Assert.Equal(result, _result);
    }
}
}

```

5- Build the solution. The build should succeed.

6- Run the test again.

The test should execute and fail, this is expected. In the Test Detail Summary pane of Test Explorer you can see that the first two “Given” steps executed successfully and the “When the two numbers are added” step failed with an *error* : *The method or operation is not implemented*. This is because the addition method of the calculator is not implemented yet.



In the next step you'll fix the implementation of the calculator to fix this error.



## CHAPTER 16

---

### Fix implementation

---

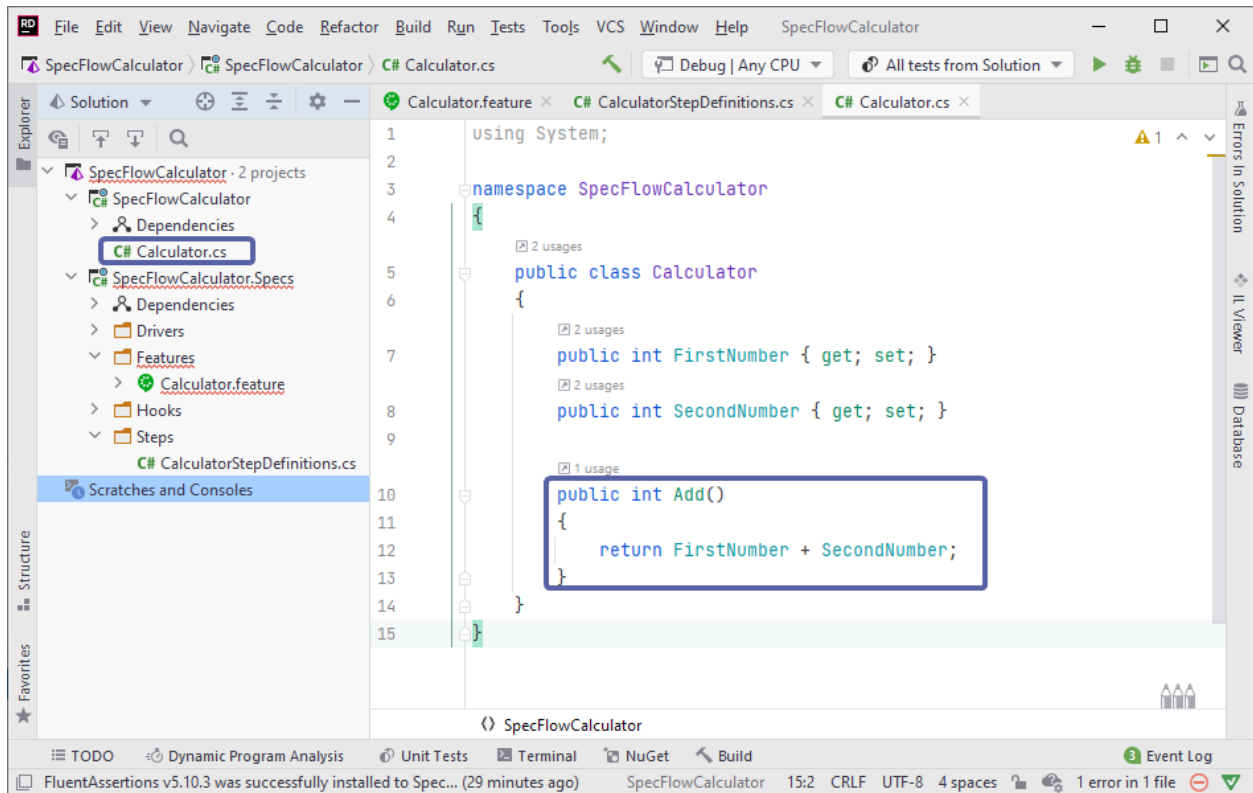
3 minutes

In this step you'll fix the implementation error of the calculator in the previous page.

**1-** Open `Calculator.cs` in the `SpecFlowCalculator` class library and replace the implementation of the `Add` method with the below code:

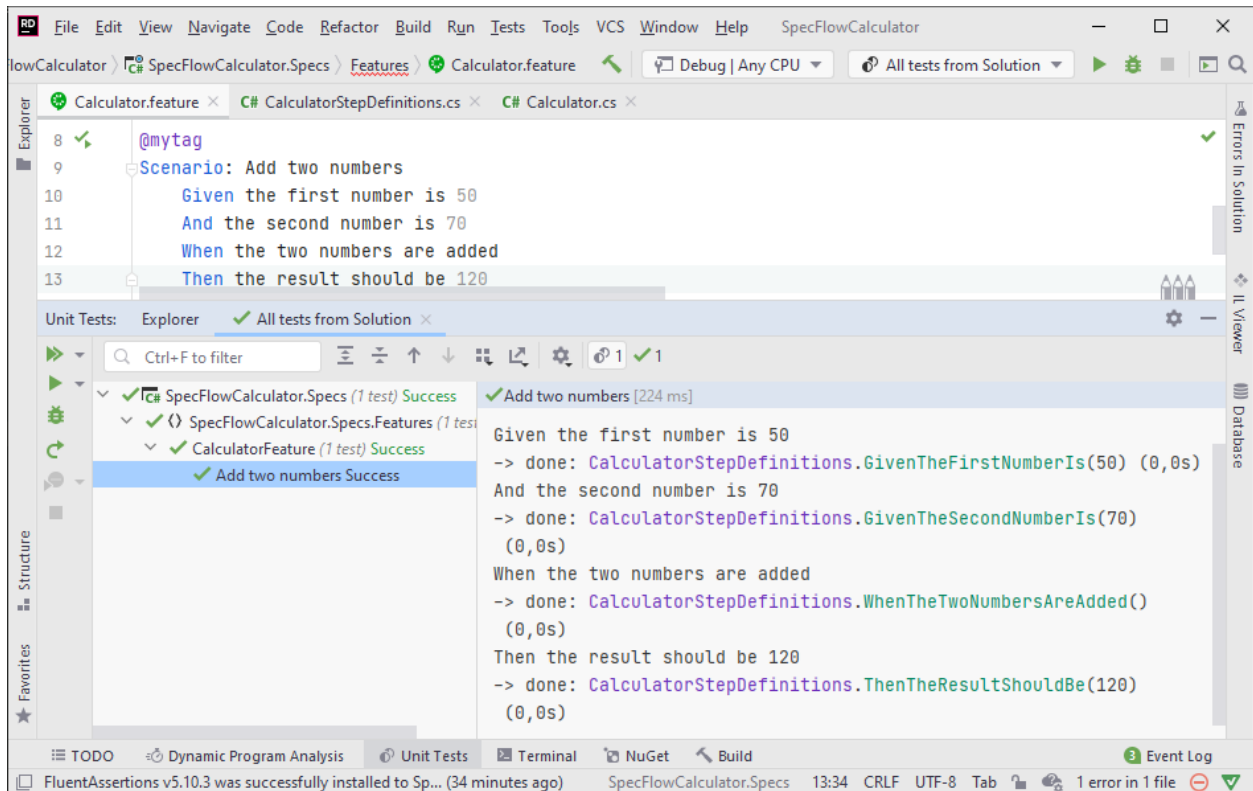
```
public int Add()
{
    return FirstNumber + SecondNumber;
}
```

## Welcome to the Step-By-Step Getting Started Guide!



2- Build the solution. The build should succeed.

3- Run the test. The test should now execute and succeed with the green tick marks indicating no errors:





The automation phase is finished, in the next step you'll learn how to generate living documentation for reporting purposes.



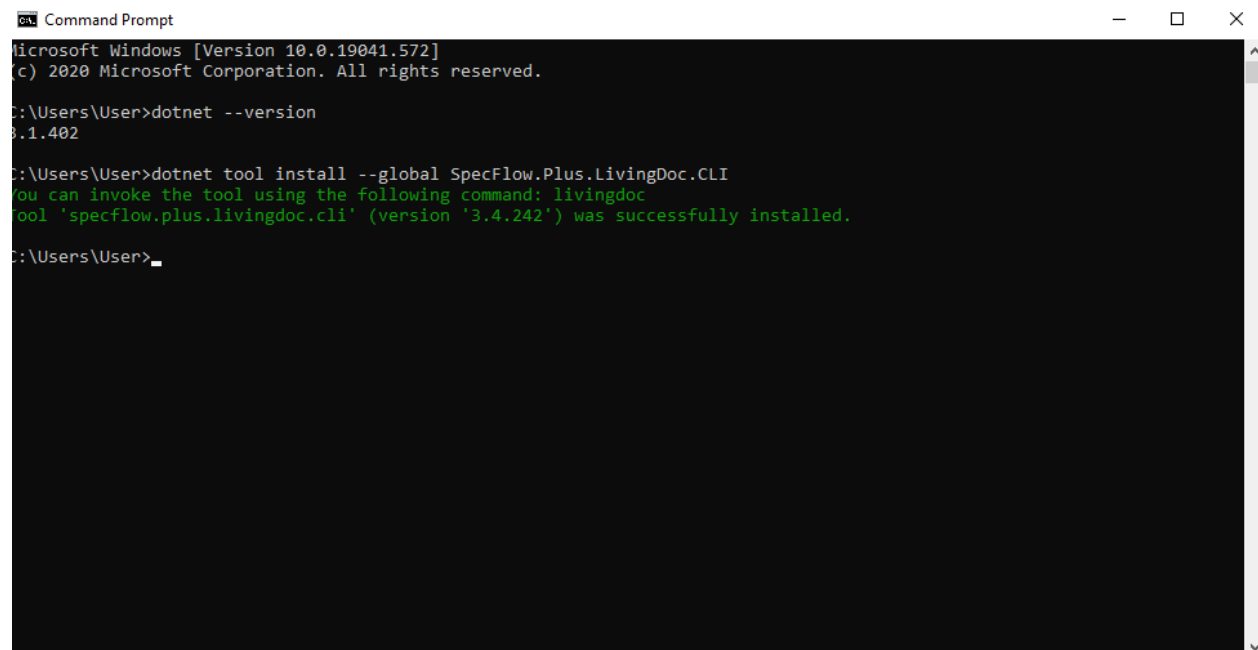
## Add Living Documentation

5 minutes

In this step you'll learn how to generate a living documentation from your test execution results so you can easily share them with your team.

- 1- Open a Command Prompt.
- 2- Install the LivingDoc CLI as a global dotnet tool.

```
dotnet tool install --global SpecFlow.Plus.LivingDoc.CLI
```



```
Command Prompt
Microsoft Windows [Version 10.0.19041.572]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\User>dotnet --version
3.1.402

C:\Users\User>dotnet tool install --global SpecFlow.Plus.LivingDoc.CLI
You can invoke the tool using the following command: livingdoc
Tool 'specflow.plus.livingdoc.cli' (version '3.4.242') was successfully installed.

C:\Users\User>
```

- 3- Navigate to the **output directory of the SpecFlow project**. In this example the solution was setup in the `C:\projects` folder.

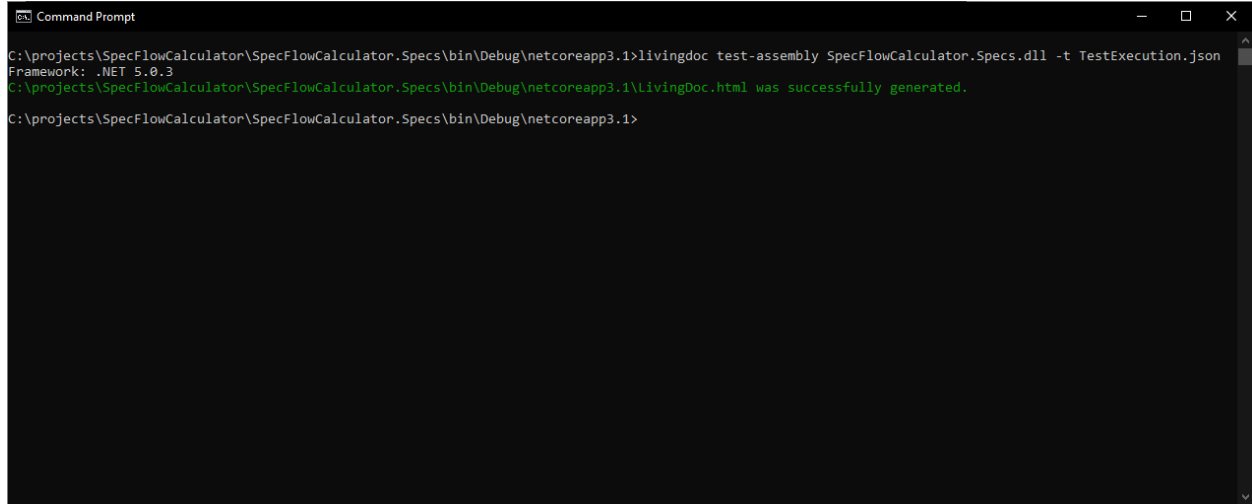
## Welcome to the Step-By-Step Getting Started Guide!

---

```
cd C:\projects\SpecFlowCalculator\SpecFlowCalculator.Specs\bin\Debug\netcoreapp3.1
```

4- Run the LivingDoc CLI by using the below command to generate the HTML report.

```
livingdoc test-assembly SpecFlowCalculator.Specs.dll -t TestExecution.json
```



```
Command Prompt
C:\projects\SpecFlowCalculator\SpecFlowCalculator.Specs\bin\Debug\netcoreapp3.1>livingdoc test-assembly SpecFlowCalculator.Specs.dll -t TestExecution.json
Framework: .NET 5.0.3
C:\projects\SpecFlowCalculator\SpecFlowCalculator.Specs\bin\Debug\netcoreapp3.1\LivingDoc.html was successfully generated.
C:\projects\SpecFlowCalculator\SpecFlowCalculator.Specs\bin\Debug\netcoreapp3.1>
```

*\*Note if you run into issues here, e.g your JSON file name is FeatureData.JSON instead of TestExecution.JSON, this indicates you have an older version of the CLI tool. Please check our [migration guide here](#) to upgrade to the latest version.*

5- Open the generated HTML with your favorite browser. The HTML file is stored in the same folder as the **output directory of the SpecFlow project**.

```
C:\projects\SpecFlowCalculator\SpecFlowCalculator.Specs\bin\Debug\netcoreapp3.1>
↪ 1\LivingDoc.html
```

Review the living documentation of the calculator features that you have implemented. Select the “Calculator” feature in the tree. On the right pane check the detailed description of the feature and the scenarios. You can also see the “green” test execution result of the scenarios and steps.

The screenshot displays the SpecFlowCalculator.Specs Living Documentation interface. The browser address bar shows the URL: `C:/projects/SpecFlowCalculator/SpecFlowCalculator.Specs/bin/Debug/netcoreapp3.1/LivingDoc.html#/document/Standalone/feature/2f3...`. The page title is "SpecFlowCalculator.Specs" and it was generated on Apr 12, 2021, at 12:01 PM GMT+2.

The interface has two tabs: "Living Documentation" (active) and "Analytics". The "Living Documentation" tab shows a tree view on the left with the following structure:

- SpecFlowCalculator.Specs
  - Features
    - Calculator
      - Add two numbers

The "Calculator" feature is expanded, showing a green checkmark and the text "Add two numbers".

The main content area displays the "Feature: Calculator" with a green checkmark and the text "Help us: [Want to edit?](#) [Want to comment?](#)". Below this is a calculator icon and the text "Simple calculator for adding **two** numbers". A link to a feature is provided: "Link to a feature: [Calculator](#)". A "Further read" link is also present: "Further read: [Learn more about how to generate Living Documentation](#)".

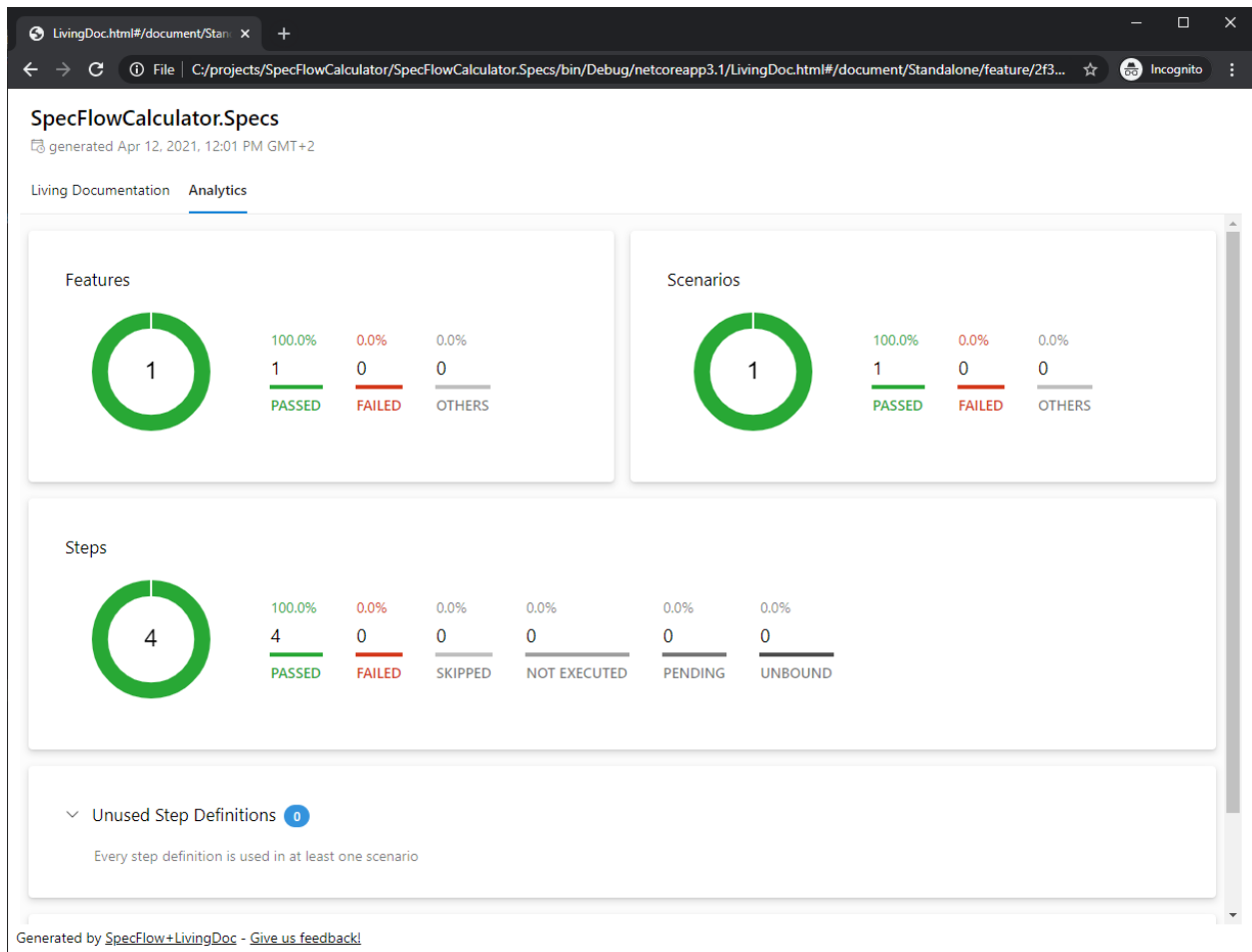
The "Scenario: Add two numbers" is listed with a green checkmark and a duration of 10ms. The scenario steps are:

- Given** the first number is 50
- And** the second number is 70
- When** the two numbers are added
- Then** the result should be 120

The footer text reads: "Generated by SpecFlow+LivingDoc - [Give us feedback!](#)".

Check the test result summary by clicking on the “Analytics” tab:

## Welcome to the Step-By-Step Getting Started Guide!



SpecFlow+LivingDoc is packed with great features that truly bring your documentation to life! To read more about SpecFlow+LivingDoc and its features, please visit our [LivingDoc documentation page](#).

## CHAPTER 18

---

### Final

---

CONGRATULATIONS!



You have now successfully created and tested your first SpecFlow project.

We have put together a little exercise for you to test your newly acquired skills, [check it out here](#).

Check out our [examples](#) page if you are looking for additional sample projects. We have also put together a sample project using Selenium for UI automation, you can find it [here](#).

To keep up to date with the latest on SpecFlow [Join the SpecFlow Community](#).





## CHAPTER 19

---

### Exercise

---

15 minutes

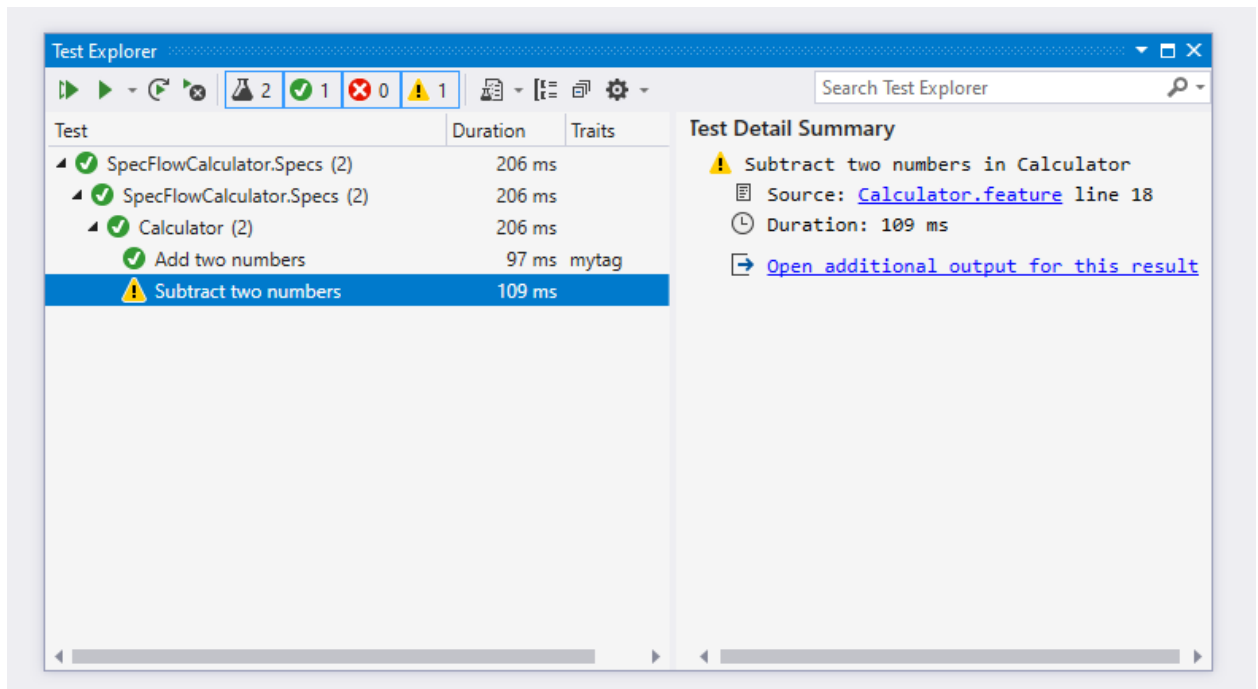
In this step it is your turn to implement the subtraction feature of the calculator.

**1-** Add the following scenario to the `Calculator.feature` file.

```
Scenario: Subtract two numbers
  Given the first number is 120
  And the second number is 70
  When the two numbers are subtracted
  Then the result should be 50
```

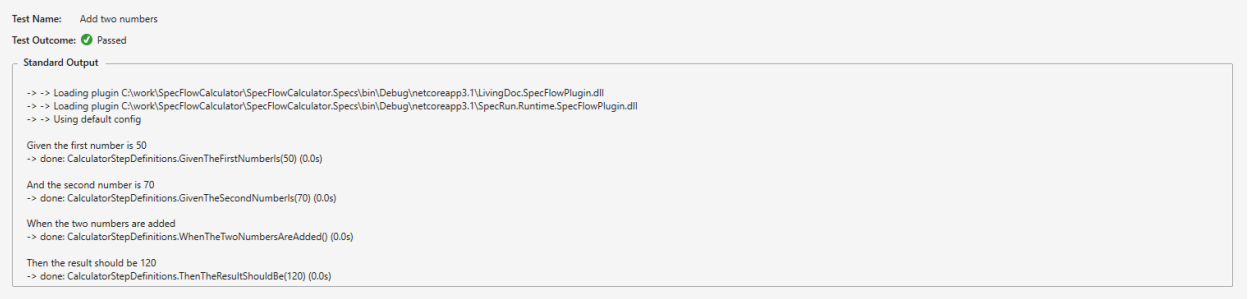
**2-** Build the solution. The build should succeed.

**3-** Run the tests. Notice that you have now 2 tests (corresponding to your two scenarios) and the second scenario is “Skipped” because of missing



bindings.

4- Click on the “Open additional output for this result” and review the details of the scenario



execution.

Now it is your turn to implement the subtraction feature in three short steps:

1. First add the missing binding (with the minimum code structure necessary) to get a red scenario.
2. Next turn the scenario green by actually implementing the subtraction logic in the calculator.
3. Refactor your implementation if necessary (scenarios should remain green).

Did it work out?

Check the next step for a possible solution.

## CHAPTER 20

---

### Exercise-solution

---

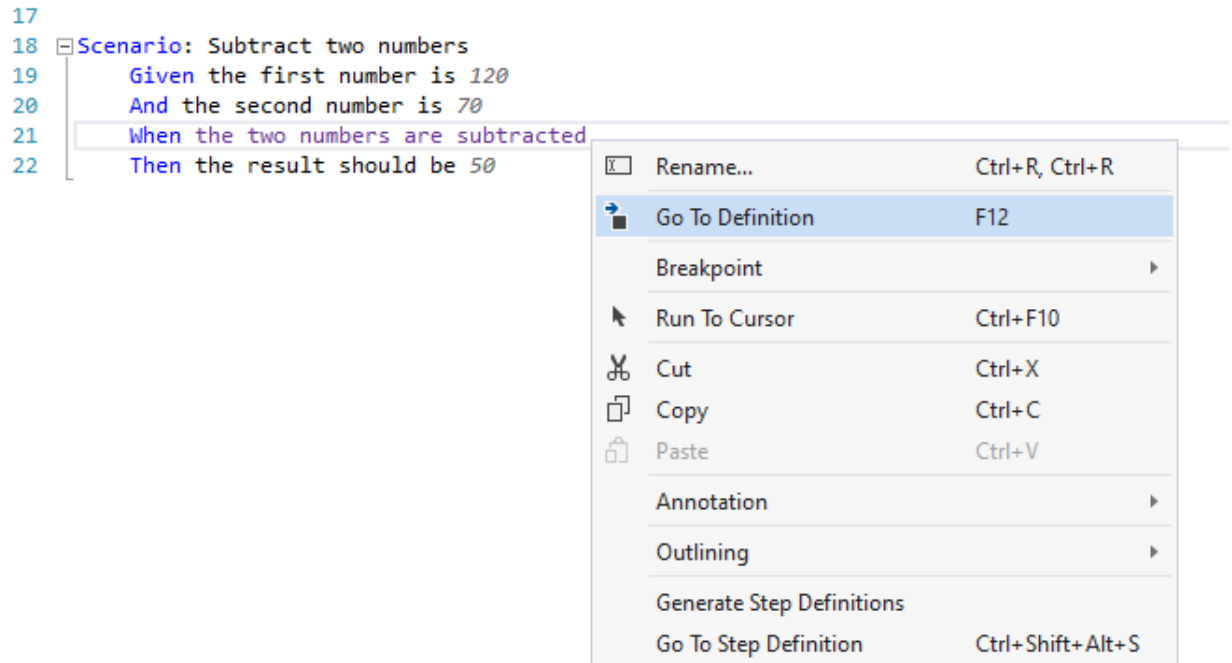
10 minutes

In this step you can review the solution for the challenge in the previous step (implementing the subtraction in the calculator).

To recap the 3 steps to implement the new feature:

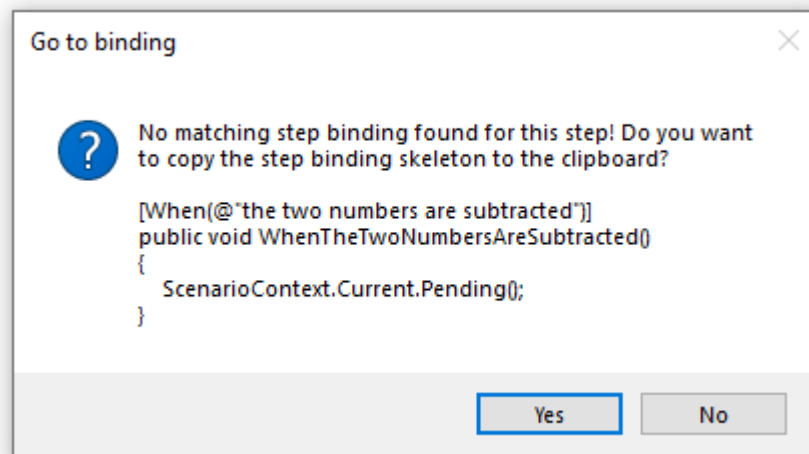
1. First add the missing bindings (with the minimum code structure necessary) to get a red scenario.
2. Next turn the scenario green by actually implementing the logic.
3. Refactor the implementation if necessary (scenarios should remain green).

The first step is to add the missing binding and necessary code to have a red scenario. A quick way of generating the necessary binding methods is to you right-click the unbound step in the feature file and select “Go To



Definition”.

If SpecFlow does not find the corresponding binding method it offers to generate the skeleton of the binding method



into your clipboard.

Now you can easily paste the method into the binding class and change the implementation.

```
[When(@"the two numbers are subtracted")]
public void WhenTheTwoNumbersAreSubtracted()
{
    _result = _calculator.Subtract();
}
```

To be able to implement the binding in a meaningful way you have to extend the public interface of the calculator as well to support the subtraction. However, in the first step, your only goal is to get to an executable red scenario. Hence you have to add a `Subtract` method to the calculator, but the implementation should be skipped e.g. by throwing a `NotImplementedException`. Note that in this case the scenario will fail in the “When” step already due to the exception and the “Then” step will be skipped.

```
using System;

namespace SpecFlowCalculator
{
    public class Calculator
    {
        public int FirstNumber { get; set; }
        public int SecondNumber { get; set; }

        public int Add()
        {
            return FirstNumber + SecondNumber;
        }

        public int Subtract()
        {
            throw new NotImplementedException();
        }
    }
}
```

Alternatively you can return a dummy value (e.g. constant 0). In this case the scenario will also run the “Then” step and fail on the assertion. This is especially beneficial if you’ve just created the binding of the Then step too and you want to make sure that the binding works as expected.

```
public int Subtract()
{
    return 0;
}
```

If you build the solution and run the tests the scenario should be red and you’re ready to move on to the second step. The second step is to implement the subtraction of the calculator to get the scenario green.

```
public int Subtract()
{
    return FirstNumber - SecondNumber;
}
```

If you run the tests again the scenario should be green.

The last step is to refactor the code while keeping all scenarios green. However, in this case the implementation is so simple that we can skip the refactoring step now.



## CHAPTER 21

---

### Additional resources

---

#### Want to learn more?

- Learn Gherkin
  - [What is Gherkin?](#)
  - [Given-When-Then with Style Challenge](#)
  - [Gherkin Reference](#)
- Architectures & Good Practices
  - [Getting started with the BookShop example](#)
- Automation Patterns
  - [Page Object Pattern](#)
  - [Driver Pattern](#)
- Example code
  - [SpecFlow Examples](#)
  - [Given-When-Then with Style challenges](#)
    - \* [Challenge 1-2](#)
    - \* [Challenge 7](#)
    - \* [Challenge 8](#)
- Example Projects
  - [Examples](#)

#### Need Help?

- [Forum](#)
- [Trainers](#)
- [Online Courses](#)