
Welcome to SpecFlow's documentation!

The SpecFlow Team

Jul 08, 2022

GETTING STARTED

1	SpecFlow components	3
2	Let's get started	5

SpecFlow is a test automation solution for .NET built upon the BDD paradigm. Use SpecFlow to define, manage and automatically execute human-readable acceptance tests in .NET projects (Full Framework and .NET Core).

SpecFlow tests are written using [Gherkin](#), which allows you to write test cases using natural languages. SpecFlow uses the official Gherkin parser, which supports over 70 languages. These tests are then tied to your application code using so-called *[bindings](#)*, allowing you to execute the tests using the testing framework of your choice. You can also execute your tests using SpecFlow's own dedicated test runner, SpecFlow+ Runner.

SPECFLOW COMPONENTS

- **SpecFlow (open-source)**: This is the core of SpecFlow, providing the functions for binding Gherkin feature files.
- **SpecFlow+ Runner** (closed-source): This is SpecFlow's dedicated test runner, and provides additional features such as [advanced execution options](#) and [execution reports](#) (HTML, XML, JSON). SpecFlow+ Runner is free of charge, and only requires a free [SpecFlow Account](#).
- **SpecFlow+ LivingDoc** (closed-source): This is a set of tools that renders your Gherkin Feature Files in an easily readable format with syntax highlighting and allows you to quickly share and collaborate on Gherkin Feature Files with stakeholders that are not familiar with developer tools (such as Visual Studio).
- **SpecFlow+ LivingDoc Generator** is available set of plugins and tools for SpecFlow to generate a local or self-hosted documentation out of your Gherkin feature files, which can be easily shared. No SpecFlow account needed.
- **SpecFlow+ LivingDoc Azure DevOps** is an extension for Azure DevOps/TFS. You can view the output directly in Azure DevOps/TFS, meaning that anyone with access to the system can easily review your specifications when needed. SpecFlow+ LivingDoc Azure DevOps is free of charge, and only requires a free [SpecFlow Account](#).

SpecFlow also includes a [Visual Studio extension](#) that adds a number of helpful features to Visual Studio (e.g. Intellisense, feature file templates, context menu entries). However, SpecFlow is not tied to Visual Studio; you can use SpecFlow with Mono or VSCode as well.

LET'S GET STARTED

You can find a number of step- by- step guides to start with SpecFlow [here](#). There are guides available for both complete beginners and more advanced users.

2.1 Selenium with Page Object Model Pattern

Selenium is a free (open-source) automation framework used for web applications across different browsers and platforms, you can read more about them [here](#). It is often used in connection with SpecFlow to test your web application via their user interface.

The Page Object Model Pattern is an often used pattern to abstract your page into separate classes. With it you have your element selectors at dedicated locations and not scattered around your automation code.

2.1.1 Sample Project Setup

You can download this entire sample project from [Github](#).

Base of this sample project is the default project that is created from the SpecFlow project template.

Additional used NuGet package to the standard packages:

- [Selenium.Support](#) - Main package for Selenium
- [Selenium.WebDriver.ChromeDriver](#) - Package that contains the ChromeDriver so Selenium is able to control the Chrome browser

2.1.2 Sample Scenario

The web application we are testing in this example is a simple calculator implementation hosted [here](#). Feel free to use this for practice if you like to.

We are testing the web application by simply adding two numbers together and checking the results.

In order to test more than just the two initial numbers in the default feature file from the project template we have added an extra Scenario Outline with the parameters `First number`, `Second number`, and `Expected result`. Now we can use an example table to include as many numbers as we like.

Here is a snippet of the feature file:

Calculator.feature

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

Scenario Outline: Add two numbers permutations

Given the first number is <First number>
And the second number is <Second number>
When the two numbers are added
Then the result should be <Expected result>

Examples:

First number	Second number	Expected result
0	0	0
-1	10	9
6	9	15

2.1.3 Starting and quitting the Browser

We start with configuring the browser behavior, the opening and closing of Google Chrome for our tests:

This class handles it for us. When you access the `Current` property the first time, the browser will be opened. If we did this, the browser will be automatically closed after the scenario finished.

BrowserDriver.cs

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

namespace CalculatorSelenium.Specs.Drivers
{
    /// <summary>
    /// Manages a browser instance using Selenium
    /// </summary>
    public class BrowserDriver : IDisposable
    {
        private readonly Lazy<IWebDriver> _currentWebDriverLazy;
        private bool _isDisposed;

        public BrowserDriver()
        {
            _currentWebDriverLazy = new Lazy<IWebDriver>(CreateWebDriver);
        }

        /// <summary>
        /// The Selenium IWebDriver instance
        /// </summary>
        public IWebDriver Current => _currentWebDriverLazy.Value;
    }
}
```

(continues on next page)

(continued from previous page)

```

    /// <summary>
    /// Creates the Selenium web driver (opens a browser)
    /// </summary>
    /// <returns></returns>
    private IWebDriver CreateWebDriver()
    {
        //We use the Chrome browser
        var chromeDriverService = ChromeDriverService.CreateDefaultService();

        var chromeOptions = new ChromeOptions();

        var chromeDriver = new ChromeDriver(chromeDriverService, chromeOptions);

        return chromeDriver;
    }

    /// <summary>
    /// Disposes the Selenium web driver (closing the browser) after the Scenario.
    ↪ completed
    /// </summary>
    public void Dispose()
    {
        if (_isDisposed)
        {
            return;
        }

        if (_currentWebDriverLazy.IsValueCreated)
        {
            Current.Quit();
        }

        _isDisposed = true;
    }
}

```

2.1.4 Using Page Objects

Since we are using Page Object Model Pattern we are **not** adding our UI automation directly in bindings.

Using the Selenium WebDriver we simulate a user interacting with the webpage. The element IDs on the page are used to identify the fields we want to enter data into. Other functions here are basically simulating a user entering numbers into the calculator, adding them up, waiting for results, and moving on to the next test.

The code is well commented so you can understand what each line is for:

CalculatorPageObject.cs

```

using System;
using OpenQA.Selenium;
using OpenQA.Selenium.Support.UI;

```

(continues on next page)

(continued from previous page)

```
namespace CalculatorSelenium.Specs.PageObjects
{
    /// <summary>
    /// Calculator Page Object
    /// </summary>
    public class CalculatorPageObject
    {
        ///The URL of the calculator to be opened in the browser
        private const string CalculatorUrl = "https://specflowoss.github.io/Calculator-
↵Demo/Calculator.html";

        ///The Selenium web driver to automate the browser
        private readonly IWebDriver _webDriver;

        ///The default wait time in seconds for wait.Until
        public const int DefaultWaitInSeconds = 5;

        public CalculatorPageObject(IWebDriver webDriver)
        {
            _webDriver = webDriver;
        }

        ///Finding elements by ID
        private IWebElement FirstNumberElement => _webDriver.FindElement(By.Id("first-
↵number"));
        private IWebElement SecondNumberElement => _webDriver.FindElement(By.Id("second-
↵number"));
        private IWebElement AddButtonElement => _webDriver.FindElement(By.Id("add-button
↵"));
        private IWebElement ResultElement => _webDriver.FindElement(By.Id("result"));
        private IWebElement ResetButtonElement => _webDriver.FindElement(By.Id("reset-
↵button"));

        public void EnterFirstNumber(string number)
        {
            ///Clear text box
            FirstNumberElement.Clear();
            ///Enter text
            FirstNumberElement.SendKeys(number);
        }

        public void EnterSecondNumber(string number)
        {
            ///Clear text box
            SecondNumberElement.Clear();
            ///Enter text
            SecondNumberElement.SendKeys(number);
        }

        public void ClickAdd()
        {

```

(continues on next page)

(continued from previous page)

```

        //Click the add button
        AddButtonElement.Click();
    }

    public void EnsureCalculatorIsOpenAndReset()
    {
        //Open the calculator page in the browser if not opened yet
        if (_webDriver.Url != CalculatorUrl)
        {
            _webDriver.Url = CalculatorUrl;
        }
        //Otherwise reset the calculator by clicking the reset button
        else
        {
            //Click the rest button
            ResetButtonElement.Click();

            //Wait until the result is empty again
            WaitForEmptyResult();
        }
    }

    public string WaitForNonEmptyResult()
    {
        //Wait for the result to be not empty
        return WaitUntil(
            () => ResultElement.GetAttribute("value"),
            result => !string.IsNullOrEmpty(result));
    }

    public string WaitForEmptyResult()
    {
        //Wait for the result to be empty
        return WaitUntil(
            () => ResultElement.GetAttribute("value"),
            result => result == string.Empty);
    }

    /// <summary>
    /// Helper method to wait until the expected result is available on the UI
    /// </summary>
    /// <typeparam name="T">The type of result to retrieve</typeparam>
    /// <param name="getResult">The function to poll the result from the UI</param>
    /// <param name="isResultAccepted">The function to decide if the polled result
    ↪ is accepted</param>
    /// <returns>An accepted result returned from the UI. If the UI does not return
    ↪ an accepted result within the timeout an exception is thrown.</returns>
    private T WaitUntil<T>(Func<T> getResult, Func<T, bool> isResultAccepted) where
    ↪ T: class
    {
        var wait = new WebDriverWait(_webDriver, TimeSpan.
    ↪ FromSeconds(DefaultWaitInSeconds));

```

(continues on next page)

(continued from previous page)

```
        return wait.Until(driver =>
        {
            var result = getResult();
            if (!isResultAccepted(result))
                return default;

            return result;
        });
    }
}
```

Here is the code of the step definition file. Note the usage of the *calculatorPageObject* and *Browserdriver*.

CalculatorStepDefinitions.cs

```
using CalculatorSelenium.Specs.Drivers;
using CalculatorSelenium.Specs.PageObjects;
using FluentAssertions;
using TechTalk.SpecFlow;

namespace CalculatorSelenium.Specs.Steps
{
    [Binding]
    public sealed class CalculatorStepDefinitions
    {
        //Page Object for Calculator
        private readonly CalculatorPageObject _calculatorPageObject;

        public CalculatorStepDefinitions(BrowserDriver browserDriver)
        {
            _calculatorPageObject = new CalculatorPageObject(browserDriver.Current);
        }

        [Given("the first number is (.*)")]
        public void GivenTheFirstNumberIs(int number)
        {
            //delegate to Page Object
            _calculatorPageObject.EnterFirstNumber(number.ToString());
        }

        [Given("the second number is (.*)")]
        public void GivenTheSecondNumberIs(int number)
        {
            //delegate to Page Object
            _calculatorPageObject.EnterSecondNumber(number.ToString());
        }

        [When("the two numbers are added")]
        public void WhenTheTwoNumbersAreAdded()
        {
            //delegate to Page Object
            _calculatorPageObject.ClickAdd();
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

    }

    [Then("the result should be (.*)")]
    public void ThenTheResultShouldBe(int expectedResult)
    {
        //delegate to Page Object
        var actualResult = _calculatorPageObject.WaitForNonEmptyResult();

        actualResult.Should().Be(expectedResult.ToString());
    }
}

```

> **Note:** The Then step here is the “testing” part where we compare the results from the Page Object Model Pattern with the expected results. As there is a time delay between hitting the add button and getting the result, we need to handle this behavior with the `WaitForNonEmptyResult()` method.

2.1.5 Using the same browser for all scenarios

In order to avoid having multiple browsers opened up during the test run and save some time per scenario, we use the **same** browser to run all the tests. For that we have introduced the below *Hook*. The major trade off here is you lose the ability to run test in parallel since you are using a single browser instance.

SharedBrowserHooks.cs

```

using BoDi;
using CalculatorSelenium.Specs.Drivers;
using TechTalk.SpecFlow;

namespace CalculatorSelenium.Specs.Hooks
{
    /// <summary>
    /// Share the same browser window for all scenarios
    /// </summary>
    /// <remarks>
    /// This makes the sequential execution of scenarios faster (opening a new browser_
    ↪ window each time would take more time)
    /// As a tradeoff:
    /// - we cannot run the tests in parallel
    /// - we have to "reset" the state of the browser before each scenario
    /// </remarks>
    [Binding]
    public class SharedBrowserHooks
    {
        [BeforeTestRun]
        public static void BeforeTestRun(ObjectContainer testThreadContainer)
        {
            //Initialize a shared BrowserDriver in the global container
            testThreadContainer.BaseContainer.Resolve<BrowserDriver>();
        }
    }
}

```

If you don't want this, simply delete the class.

2.1.6 Resetting the Web Application

Because we reuse the browser instance, we have to reset the web app for every scenario. We are using again a hook to do this.

CalculatorHooks.cs

```
using CalculatorSelenium.Specs.Drivers;
using CalculatorSelenium.Specs.PageObjects;
using TechTalk.SpecFlow;

namespace CalculatorSelenium.Specs.Hooks
{
    /// <summary>
    /// Calculator related hooks
    /// </summary>
    [Binding]
    public class CalculatorHooks
    {
        ///<summary>
        /// Reset the calculator before each scenario tagged with "Calculator"
        /// </summary>
        [BeforeScenario("Calculator")]
        public static void BeforeScenario(BrowserDriver browserDriver)
        {
            var calculatorPageObject = new CalculatorPageObject(browserDriver.Current);
            calculatorPageObject.EnsureCalculatorIsOpenAndReset();
        }
    }
}
```

2.1.7 Further Reading

If you want to get into more details, have a look at the following documentation pages:

- [Hooks](#)All about Hooks, the lifecycle events in SpecFlow
- [Driver Pattern](#)More details and examples about the Driver Pattern, which we used for the Browser Lifecycle handling
- [Page Object Model Pattern](#)More details and examples for the Page Object Model Patter

2.2 Using Page Object Model

The Page Object Model is a pattern, that is often used to abstract your Web UI with Selenium to easier automate it.

So to automate following HTML snippet

```
<input id="txtUrl" name="Url" type="text" value="">
```

you have following class to control it

```
public class PageObject
{
    public IWebElement TxtUrl {get;}
}
```

When you are working with Selenium, you are always working with WebElements to access the different elements on your Website. You can find them with the FindElement and FindElements methods on the WebDriver class. If you are always using these methods directly in your automation code, you will get a lot of code duplication. This is the moment when you should start using the Page Object Model. You hide the calls to the FindElement(s) methods in a class.

This has following advantages:

- the classes are easier reusable
- if you need to change an id of your element, you need to change only one place
- your bindings are less dependent on your HTML structure

2.2.1 Simple Implementation

HTML:

```
<input id="txtUrl" name="Url" type="text" value="">
```

Code:

```
public class PageObject
{
    private IWebDriver _webDriver;

    public PageObject(IWebDriver webDriver)
    {
        _webDriver = webDriver;
    }

    public IWebElement txtUrl => _webDriver.FindElement(By.Id("txtUrl"));
}
```

You pass your WebDriver instance via constructor, and always when you access the TxtUrl property, the WebDriver searches on the whole page for an element with the id txtUrl. There is no caching involved.

2.2.2 Implementation with Caching

HTML:

```
<input id="txtUrl" name="Url" type="text" value="">
```

Code:

```
public class PageObject
{
    private IWebDriver _webDriver;
    private Lazy<IWebElement> _txtUrl;

    public PageObject(IWebDriver webDriver)
    {
        _webDriver = webDriver;
        _txtUrl = new Lazy<IWebElement>(() => _webDriver.FindElement(By.Id("txtUrl")));
    }

    public IWebElement txtUrl => _txtUrl.Value;
}
```

Again You pass your WebDriver instance via constructor. In this case we are using [Lazy](#) as a easy way to cache the result of the FindElement method. Only the first call to the txtUrl property, triggers a call to the FindElement function. All subsequent calls, will return the same value as before. This will save you some time in execution of your automation code, as the WebDriver needs to do search less often for the same element.

If you use a caching strategy like that, be careful with your lifetime of your page objects and your page. Don't reuse an old instance of your page model, if the page changed in the meantime.

2.2.3 Implementation with Hierarchy

HTML:

```
<div class='A'>
    <div class='B' />
</div>
<div class='B'>
</div>
```

Code:

```
public class ParentPageObject
{
    private IWebDriver _webDriver;

    public ParentPageObject(IWebDriver webDriver)
    {
        _webDriver = webDriver;
    }

    public IWebElement WebElement => _webDriver.FindElement(By.ClassName("A"));

    public ChildPageObject Child => new ChildPageObject(WebElement);
}
```

(continues on next page)

(continued from previous page)

```

}

public class ChildPageObject
{
    private IWebElement _webElement;
    private Lazy<IWebElement> _txtUrl;

    public ChildPageObject(IWebElement webElement)
    {
        _webElement = webElement;
    }

    public IWebElement WebElement => _webElement.FindElement(By.ClassName("B"));
}

```

In this example we have a slightly adjusted HTML document to work with. There are two div- elements with the same class B, but we only want the PageObject for the div- element with the class A and the child.

If we would use the same `WebDriver.FindElement` method we would get the div- element that is on the same level as the A div. But every `WebElement` has also the `FindElement(s)`- methods. This enable you to query the elements only in a part of your whole HTML DOM. To do that we are passing this time the parent- `WebElement` to the `ChildPageObject` class to only search for the element with the class B within the A- div.

This concept enables you to structure your PageObjects in a similar way you have your HTML DOM structure.

2.2.4 Further resources

- <https://www.browserstack.com/guide/page-object-model-in-selenium>
- https://www.selenium.dev/documentation/en/guidelines_and_recommendations/page_object_models/
- <https://martinfowler.com/bliki/PageObject.html>

2.3 Driver Pattern

The Driver Pattern is simply an additional layer between your step definitions and your automation code.

Over the years, we noticed that a good practice to organize your bindings and automation code is to keep the code in the bindings very short (around 10 lines) and easy understandable.

This gives you following benefits:

- easier to maintain your test automation code As you split your code into multiple parts, it gets easier to maintain
- easy to reuse methods in different step definitions or combine multiple steps into a single step We often see, a group of steps that are in a lot of Scenarios. As you have now the automation code in separate classes, chaining the method calls is really easy.
- easier to read step definitions This makes it possible, that also non- technical people can understand what is happening in a step definition. This makes your life in bigger projects easier, because nobody will remember what every single step is doing.

The Driver pattern is heavily using *Context- Injection* to connect the multiple classes together.

2.3.1 Example

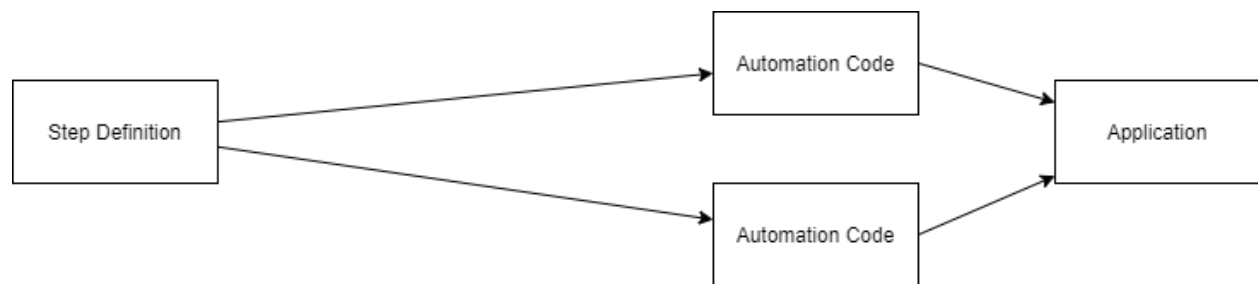
In this example you see how the code looks before and after refactoring with the Driver pattern.

Before:

This is some automation code that uses the [Page Object Model](#) and checks if some WebElements are existing.

```
[Then(@"it is possible to enter a '(.*)' with label '(.*)'")]
public void ThenItIsPossibleToEnterAWithLabel(string inputType, string expectedLabel)
{
    var submissionPageObject = new SubmissionPageObject(webDriverDriver);

    switch (inputType.ToUpper())
    {
        case "URL":
            submissionPageObject.UrlWebElement.Should().NotBeNull();
            submissionPageObject.UrlLabel.Should().Be(expectedLabel);
            break;
        case "TYPE":
            submissionPageObject.TypeWebElement.Should().NotBeNull();
            submissionPageObject.TypeLabel.Should().Be(expectedLabel);
            break;
        default:
            throw new NotImplementedException(inputType + " not implemented");
    }
}
```



After:

With moving the automation code into a driver class, we could reduce the number of lines in the step definition to one. Also we can now use a method-name (`CheckExistenceOfInputElement`), that is understandable by everybody in your team.

To get an instance of the driver class (`SubmissionSteps`), we are using the [Context- Injection](#) Feature of SpecFlow.

```
[Binding]
public class SubmissionSteps
{
    private readonly SubmissionPageDriver submissionPageDriver;

    public SubmissionSteps(SubmissionPageDriver submissionPageDriver)
    {
        this.submissionPageDriver = submissionPageDriver;
    }
}
```

(continues on next page)

(continued from previous page)

```
[Then(@"it is possible to enter a '(.*)' with label '(.*)'")]
public void ThenItIsPossibleToEnterAWithLabel(string inputType, string expectedLabel)
{
    submissionPageDriver.CheckExistenceOfInputElement(inputType, expectedLabel);
}

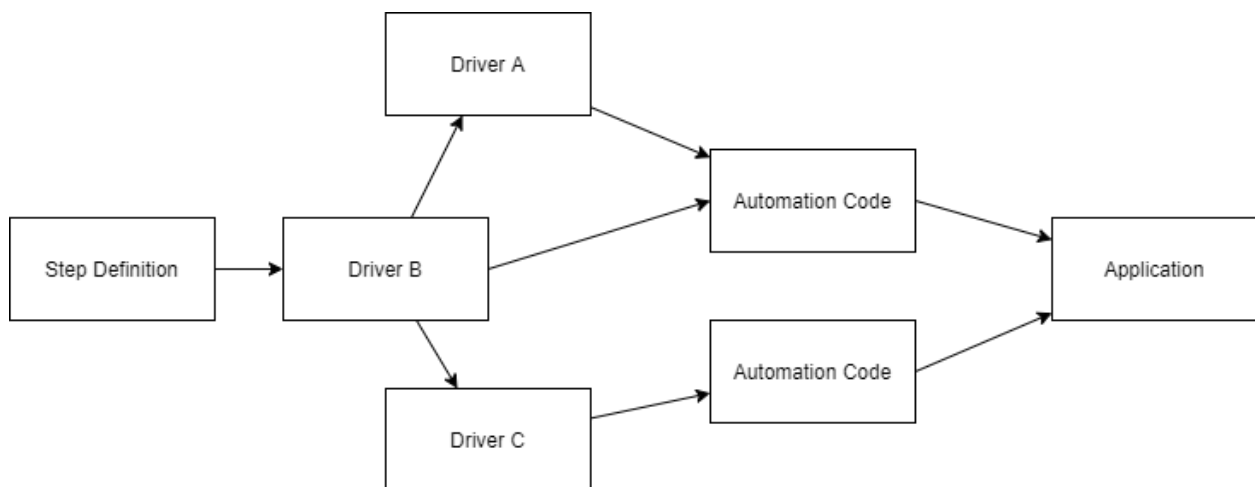
// ...
```

```
public class SubmissionPageDriver
{
    // ...

    public void CheckExistenceOfInputElement(string inputType, string expectedLabel)
    {
        var submissionPageObject = new SubmissionPageObject(webDriverDriver);

        switch (inputType.ToUpper())
        {
            case "URL":
                submissionPageObject.UrlWebElement.Should().NotBeNull();
                submissionPageObject.UrlLabel.Should().Be(expectedLabel);
                break;
            case "TYPE":
                submissionPageObject.TypeWebElement.Should().NotBeNull();
                submissionPageObject.TypeLabel.Should().Be(expectedLabel);
                break;
            default:
                throw new NotImplementedException(inputType + " not implemented");
        }
    }

    // ...
}
```



2.3.2 Further Resources

- <http://leitner.io/2015/11/14/driver-pattern-empowers-your-specflow-step-definitions>

2.4 Targeting Multiple Browser with a Single Test

If you are testing a web app (e.g. with Selenium), you will normally want to test it in a range of browsers, e.g. Chrome, IE/Edge and Firefox. However, writing tests for all the browsers can be a time-consuming process. It would be much easier to just write just one test and run that test in all browsers.

SpecFlow+ Runner allows you to do this by using **Targets**. Targets are defined in your SpecFlow+ Runner profile. They allow you to define different environment settings, filters and deployment transformation steps for each target. Another common use case is to define separate targets for X64 and x86.

Defining targets for each browser allows us to execute the same test in all browsers. You can see this in action in the Selenium sample project available on [GitHub](#). If you download the solution and open `Default.srprofile`, you will see 3 different targets defined at the end of the file:

```
<Targets>
  <Target name="IE">
    <Filter>Browser_IE</Filter>
    <DeploymentTransformationSteps>
      <EnvironmentVariable variable="Test_Browser" value="IE" />
    </DeploymentTransformationSteps>
  </Target>
  <Target name="Chrome">
    <Filter>Browser_Chrome</Filter>
    <DeploymentTransformationSteps>
      <EnvironmentVariable variable="Test_Browser" value="Chrome" />
    </DeploymentTransformationSteps>
  </Target>
  <Target name="Firefox">
    <Filter>Browser_Firefox</Filter>
    <DeploymentTransformationSteps>
      <EnvironmentVariable variable="Test_Browser" value="Firefox" />
    </DeploymentTransformationSteps>
  </Target>
</Targets>
```

Each of the targets has a name and an associated filter (e.g. "Browser_IE"). The filter ensures that only tests with the corresponding tag are executed for that target.

For each target, we **transform** the `Test_browser` environment variable to contain the name of the target. This will allow us to know the current target and access the corresponding web driver for each browser. `WebDriver.cs` (located in the `Drivers` folder of the `TestApplication.UiTests` project) uses this key to instantiate a web driver of the appropriate type (e.g. `InternetExplorerDriver`). Based on the value of this environment variable, the appropriate web driver is returned by `GetWebDriver()` is passed to `BrowserConfig`, used in the switch statement:

```
private IWebDriver GetWebDriver()
{
    switch (Environment.GetEnvironmentVariable("Test_Browser"))
    {
        case "IE": return new InternetExplorerDriver(new InternetExplorerOptions {
            IgnoreZoomLevel = true }) { Url = SeleniumBaseUrl };
    }
}
```

(continues on next page)

(continued from previous page)

```

    case "Chrome": return new ChromeDriver { Url = SeleniumBaseUrl };
    case "Firefox": return new FirefoxDriver { Url = SeleniumBaseUrl };
    case string browser: throw new NotSupportedException($"{browser} is not a supported_
↪browser");
    default: throw new NotSupportedException("not supported browser: <null>");
  }
}

```

Depending on the target, the driver is instantiated as either the *InternetExplorerDriver*, *ChromeDriver* or *FirefoxDriver* driver type. The bindings code simply uses the required web driver for the target to execute the test; there is no need to write separate tests for each browser. You can see this at work in the [Browser.cs](#) and [CalculatorFeatureSteps.cs](#) files:

```

[Binding]
public class CalculatorFeatureSteps
{
    private readonly WebDriver _webDriver;

    public CalculatorFeatureSteps(WebDriver webDriver)
    {
        _webDriver = webDriver;
    }

    [Given(@"I have entered (.*) into (.*) calculator")]
    public void GivenIHaveEnteredIntoTheCalculator(int p0, string id)
    {
        _webDriver.Wait.Until(d => d.FindElement(By.Id(id))).SendKeys(p0.ToString());
    }
}

```

To ensure that the tests are executed, you still need to ensure that the tests have the appropriate tags (*@Browser_Chrome*, *@Browser_IE*, *@Browser_Firefox*). 2 scenarios have been defined in [CalculatorFeature.feature](#):

```

@Browser_Chrome
@Browser_IE
@Browser_Firefox
Scenario: Basepage is Calculator
    Given I navigated to /
    Then browser title is Calculator

@Browser_IE
@Browser_Chrome
Scenario Outline: Add Two Numbers
    Given I navigated to /
    And I have entered <SummandOne> into summandOne calculator
    And I have entered <SummandTwo> into summandTwo calculator
    When I press add
    Then the result should be <Result> on the screen

Scenarios:

```

	SummandOne	SummandTwo	Result
	50	70	120
	1	10	11

Using targets in this way can significantly reduce the number of tests you need to write and maintain. You can reuse the same test and bindings for multiple browsers. Once you've set up your targets and web driver, all you need to do is tag your scenarios correctly. If you select "Traits" under **Group By Project** in the Test Explorer, the tests are split up by browser tag. You can easily run a test in a particular browser and identify which browser the tests failed in. The test report generated by SpecFlow+ Runner also splits up the test results by target/browser.

Remember that targets can be used for a lot more than executing the same test in multiple browsers with Selenium. Don't forget to read the documentation on [targets](#), as well as the sections on [filters](#), [target environments](#) and [deployment transformations](#).

2.5 External Data Plugin

You can easily apply standardized test cases across a wide range of features to significantly reduce redundant data for large test suites. By reusing execution flows, you can also speed up exploratory and approval testing for ranges of examples. SpecFlow makes all of this possible by introducing support for loading external data into scenarios easily.

The [SpecFlow ExternalData plugin](#) lets teams separate test data from test scenarios, and reuse examples across a large set of scenarios. This is particularly helpful when a common set of examples needs to be consistently verified in different scenarios.

Simply download the [NuGet package](#) and add it to your specflow projects to use it.

2.5.1 Supported Data Sources

- CSV files (format 'CSV', extension .csv)

> **Note:** Standard RFC 4180 CSV format is supported with a header line (plugin uses [CsvHelper](#) to parse the files).

- Excel files (format Excel, extensions .xlsx, .xls, .xlsb)

> **Note:** Both XLSX and XLS is supported (plugin uses [ExcelDataReader](#) to parse the files).

2.5.2 Tags

The following tags can be used to specify the external source:

- `@DataSource:path-to-file` - This tag is the main tag that you can add to a scenario or a scenario outline to specify the data source you wish to use.

> **Important:** The path is a relative path to the folder of the *feature files*.

- `@DisableDataSource` - The `@DataSource` tag can be added to the feature node, turning all scenarios in the file to scenario outlines. This method is useful when the entire feature file uses the same data source. Use the `@DisableDataSource` If you want a select few scenarios in the feature file to **not** use the data source tagged at feature node level.
- `@DataFormat:format` - This tag only needs to be used if the format cannot be identified from the file extension.
- `@DataSet:data-set-name` - This tag is applicable to *Excel files only*. It is used to select the worksheet of the Excel file you wish to use. By **default**, the first worksheet in an Excel file is targeted.
- `@DataField:name-in-feature-file=name-in-source-file` - This tag can be used to "rename" columns of the external data source.

General notes on tags:

- Tags can be added on feature, scenario, scenario outline or scenario outline examples.

- Tags can inherit from the feature node, but you can override them with another tag or disable them by using the `@DisableDataSource` tag on the scenario level.
- As tags cannot contain spaces, generally the underscore (`_`) character can be used to represent a space. It is currently not supported to access a file that contains spaces in the file name or in the relative path.

2.5.3 Examples

CSV files

The below examples all use the same *products.csv* file. The file contains three products and their corresponding prices:

	A
1	product,price
2	Chocolate,2.5
3	Apple,1.0
4	Orange,1.2

- This scenario will be treated as a scenario outline with the products from the CSV file replacing the `<product>` parameter in the given statement:

```
@DataSource:products.csv
Scenario: Valid product prices are calculated
    Given the customer has put 1 piece of <product> in the basket
    When the basket price is calculated
    Then the basket price should be greater than zero
```

- This scenario will be treated as a scenario outline similar to the above example but uses both `<product>` and `<price>` from the CSV file:

```
@DataSource:products.csv
Scenario: The basket price is calculated correctly
    Given the price of <product> is €<price>
    And the customer has put 1 pcs of <product> to the basket
    When the basket price is calculated
    Then the basket price should be €<price>
```

- This scenario shows how you can extend the product list using the example table with the ones from the CSV file. A total of 4 products will be added here, 3 from the CSV file plus “Cheesecake” from the example table:

```
@DataSource:products.csv
Scenario Outline: Valid product prices are calculated (Outline)
    Given the customer has put 1 pcs of <product> to the basket
    When the basket price is calculated
    Then the basket price should be greater than zero
Examples:
    | product      |
    | Cheesecake   |
```

You may also add the `@DataSource` above the example table if you wish to:

```
Scenario Outline: Valid product prices are calculated (Outline, example annotation)
    Given the customer has put 1 pcs of <product> to the basket
```

(continues on next page)

(continued from previous page)

```
When the basket price is calculated
Then the basket price should be greater than zero
@DataSource:products.csv
Examples:
| product |
| Cheesecake |
```

- In this scenario the parameters names do not match the column names in the CSV file but we can address that by using the @DataField:product-name=product and @DataField:price-in-EUR=price tags:

```
@DataSource:products.csv @DataField:product-name=product @DataField:price-in-EUR=price
Scenario: The basket price is calculated correctly (renamed fields)
Given the price of <product-name> is €<price-in-EUR>
And the customer has put 1 piece of <product-name> in the basket
When the basket price is calculated
Then the basket price should be €<price-in-EUR>
```

- This scenario is similar to the above scenario with the renaming of the parameters, but the difference is the use of space in the parameter name. Spaces are **not** supported and must be replaced with underscore (_):

```
@DataSource:products.csv @DataField:product_name=product @DataField:price-in-EUR=price
Scenario: The basket price is calculated correctly

Given the customer has put 1 piece of <product name> in the basket
When the basket price is calculated
Then the basket price should be greater than zero
Examples:
| product name |
| Cheesecake |
```

Excel files

You can use Excel files the same way as you do with CSV files with some minor differences:

- Only simple worksheets are supported, where the **header is in the first row** and the data comes right below that. Excel files that contain tables, graphics, etc. are not supported.
- Excel files with multiple worksheets are supported, you can use the @DataSet:sheet-name to select the worksheets you wish to target. The plugin uses the **first** worksheet by **default**.
- Use underscores in the @DataSet tag instead of spaces if the worksheet name contains spaces.

The below example shows an Excel file with multiple worksheets and we wish to target the last worksheet labelled "other products". We do this by using the @DataSet:other_products tag. Note the use of () instead of space:

	A	B	C
1	product	price	color
2	Cookie	€ 2,50	brown
3	Bananas	€ 1,00	yellow
4			
5			
6			
7			

products users **other products**

@DataSource:products.xlsx @DataSet:other_products

Scenario: The basket price is calculated correctly for other products

Given the price of <product> is €<price>

And the customer has put 1 piece of <product> in the basket

When the basket price is calculated

Then the basket price should be €<price>

2.5.4 Language Settings

The decimal and date values read from an Excel file will be exported using the language of the feature file (specified using the #language setting in the feature file or in the SpecFlow configuration file). This setting affects for example the decimal operator as in some countries comma (,) is used as decimal separator instead of dot (.). To specify not only the language but also the country use the #language: language-country tag, e.g. #language: de-AT for Deutsch-Austria.

Example: Hungarian uses comma (,) as decimal separator instead of dot (.), so SpecFlow will expect the prices in format 1,23:

This sample shows that the language settings are applied for the data that is being read by the external data plugin.

#language: hu-HU

Jellemző: External Data from Excel file (Hungarian)

@DataSource:products.xlsx

Forgatókönyv: The basket price is calculated correctly

Amennyiben the price of <product> is €<price>

És the customer has put 1 pcs of <product> to the basket

Amikor the basket price is calculated

Akkor the basket price should be €<price>

2.6 Bookshop Example

Follow our step by step guide to get started, learn, explore and experiment with a simple web application project using SpecFlow and the SpecFlow+ Runner.

2.6.1 Prerequisites to run the application

- .NET Core 3.1 SDK
- Visual Studio 2019 or Visual Studio Code is recommended

If you use Visual Studio 2019, please install the [SpecFlow extension](#) for Visual Studio.

2.6.2 Get the BookShop Example

The SpecFlow sample applications are publicly available in the [SpecFlow-Examples](#) GitHub repository.

You can clone the repository in Visual Studio 2019 by selecting the “Clone a repository” option on the start screen. Use the GitHub URL <https://github.com/SpecFlowOSS/SpecFlow-Examples.git> as repository location.

Alternatively you can clone the repository from the command line:

```
git clone https://github.com/SpecFlowOSS/SpecFlow-Examples.git
```

This guide will walk you through the **BookShop example** that you can find in the **ASP.NET-MVC/BookShop** folder.

2.6.3 Setup the application

- Open the solution `BookShop.sln` in Visual Studio.
- Set the `BookShop.Mvc` project as startup project.
- Run the application (hit F5).

The web application should start, a new browser window should be opened, and you should see a list of books on the start page of the app.

Book shop application

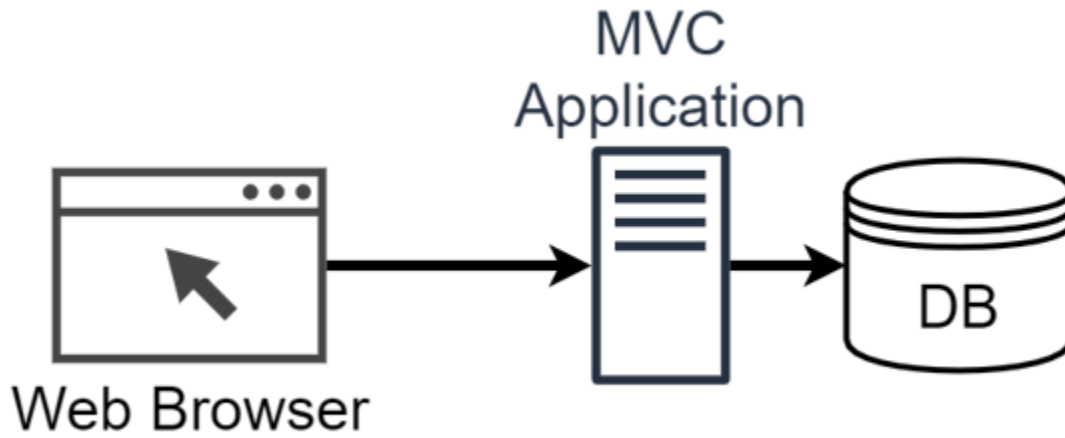
The example application is a web application, where users can search and buy BDD books. The implementation focuses on the first steps of the following user journey.



Feel free to explore the application: try to search for a book, check the details of a selected book, add it to the shopping card, manipulate the quantity.

Architecture

The application is implemented as an ASP.NET Core MVC web application and it uses Entity Framework Core for the database access.



Note: To keep the setup simple the Bookshop application uses an in-memory database.

2.6.4 Automated SpecFlow Acceptance Tests

Feature files

With SpecFlow you formulate your acceptance criteria in *.feature* files in [Given-When-Then](#) style, using the [Gherkin](#) language. Using SpecFlow these acceptance criteria can be validated with [Automated Acceptance Tests](#).

In this example the `BookShop.AcceptanceTests` project contains the feature files for the Bookshop application. These describe the implemented behaviour of the Bookshop in the form of [Features](#) and [Scenarios](#).

Open the `Book Details.feature` file to see the acceptance criteria of the **Displaying book details** feature.

Step definitions

Step definitions are implemented as .NET code in plain old .NET classes (see the .cs files in the folder *StepDefinitions*). These step definitions (also known as “bindings”) define, how the individual scenario steps should be automated.

In Visual Studio you can easily navigate from the scenario steps to the step definition that automates the step using the standard “Go To Definition” command (default hotkey: “F12”).

In the `Book Details.feature` file put the caret in the line “Given the following books” and press “F12” to jump to the step definition of this step. You can see `Given/When/Then` attributes on the C# methods and a `Binding` attribute on the class that establish the connection between the Gherkin steps and the step definitions.

Executable tests

When you build the solution SpecFlow generates executable tests from the acceptance criteria scenarios. The generated tests use the step definitions that you need to implement.

In Visual Studio you can find the generated tests files as sub-items under each feature file (see e.g. the `Book Details.feature.cs` under the `Book Details.feature` file).

***Note:** The tests in the `feature.cs` files are always generated by SpecFlow from the feature files. You should never manually modify the generated tests.*

As SpecFlow is not a unit test runner on its own, it can generate tests for a number of third party unit test runners like MsTest, NUnit, XUnit and SpecFlow+ Runner.

The Bookshop example project is configured to generate unit tests for SpecFlow+ Runner, which is a test runner provided by the SpecFlow team specialized for running acceptance/integration tests.

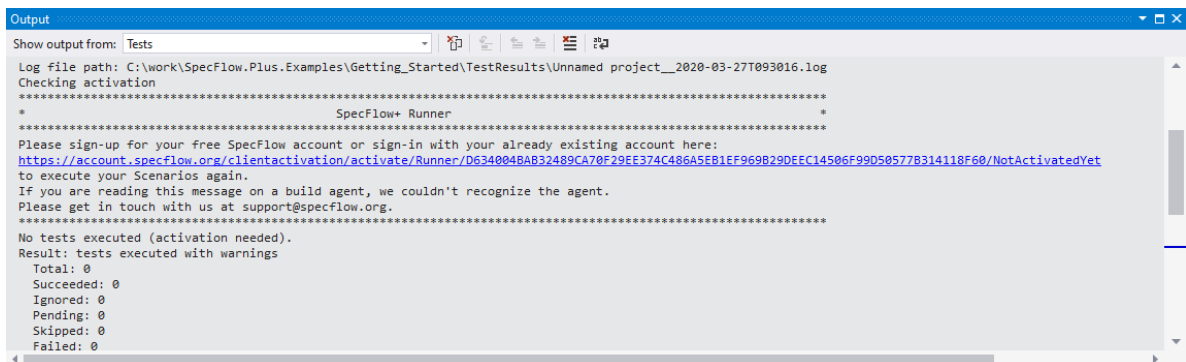
You could easily switch to other unit test providers (such as [NUnit](#), [XUnit](#), etc.) by uninstalling the current test provider NuGet package (`SpecRun.SpecFlow`) and installing another (e.g. `SpecFlow.MsTest`). However, the Bookshop example leverages some unique features of SpecFlow+ Runner, hence changing to another unit test provider would require some additional changes in the step definitions.

2.6.5 Executing SpecFlow+ Runner the first time

In this example we use [SpecFlow+ Runner](#) to execute the SpecFlow tests, but you can use a number of other test execution frameworks, including NUnit, xUnit or MSTest. SpecFlow+ Runner's advantages include integration with Visual Studio Test Runner and extensive integrated reports available from within Visual Studio.

SpecFlow+ Runner is available free of charge. Only a quick initial activation is necessary to run your scenarios.

1. Build your solution.
2. Select **Test | Windows | Test Explorer** in Visual Studio to open the Test Explorer
3. Click on **Run All** to run your test.
4. You will be asked to sign up for a [SpecFlow account](#) or to sign in with your existing account. To see the output of the SpecFlow+ Runner please open the Output window (View- > Output) and select "Tests" in the "Show output from" dropdown:



```
Output
Show output from: Tests
Log file path: C:\work\SpecFlow.Plus.Examples\Getting_Started\TestResults\Unnamed project__2020-03-27T093016.log
Checking activation
*****
SpecFlow+ Runner
*****
Please sign-up for your free SpecFlow account or sign-in with your already existing account here:
https://account.specflow.org/clientactivation/activate/Runner/D634804B832489CA70F29EE374C486A5EB1EF969B29DEEC14586F99D505778314118F60/NotActivatedYet
to execute your Scenarios again.
If you are reading this message on a build agent, we couldn't recognize the agent.
Please get in touch with us at support@specflow.org.
*****
No tests executed (activation needed).
Result: tests executed with warnings
Total: 0
Succeeded: 0
Ignored: 0
Pending: 0
Skipped: 0
Failed: 0
```

5. Open the URL in the message in your browser. In Visual Studio you can also click the link while pressing the CTRL-key, in this case Visual Studio opens the link in your default browser.

***Note:** Depending on your local system configuration the link might open a new tab in an already running browser instance and it might be not "brought to front" by Visual Studio. If seemingly nothing happens when CTRL-clicking the link switch to your running browser instance and check if the page was opened there.*

6. In the browser you are displayed with a “Welcome Page”. Click on **Sign in with Microsoft** to continue.
7. Sign in with your Microsoft account. It can be a personal or corporate/enterprise account. If you are already signed in, this should happen automatically – *you might need additional permissions from your Active Directory admin. [Learn more about admin consents](#)*
8. You will be taken to a setup page where you can set up your SpecFlow account. Enter your details to sign up for a free SpecFlow account.
9. Return to Visual Studio and click on “**Run all**” again.
10. The acceptance tests should all pass.

2.6.6 Test execution report

When you execute your acceptance tests with SpecFlow+ Runner a special test execution report is generated automatically.

To see the output of the SpecFlow+ Runner please open the Output window (View- > Output) and select “Tests” in the “Show output from” dropdown. The hyperlink to the HTML execution report should be shown there.

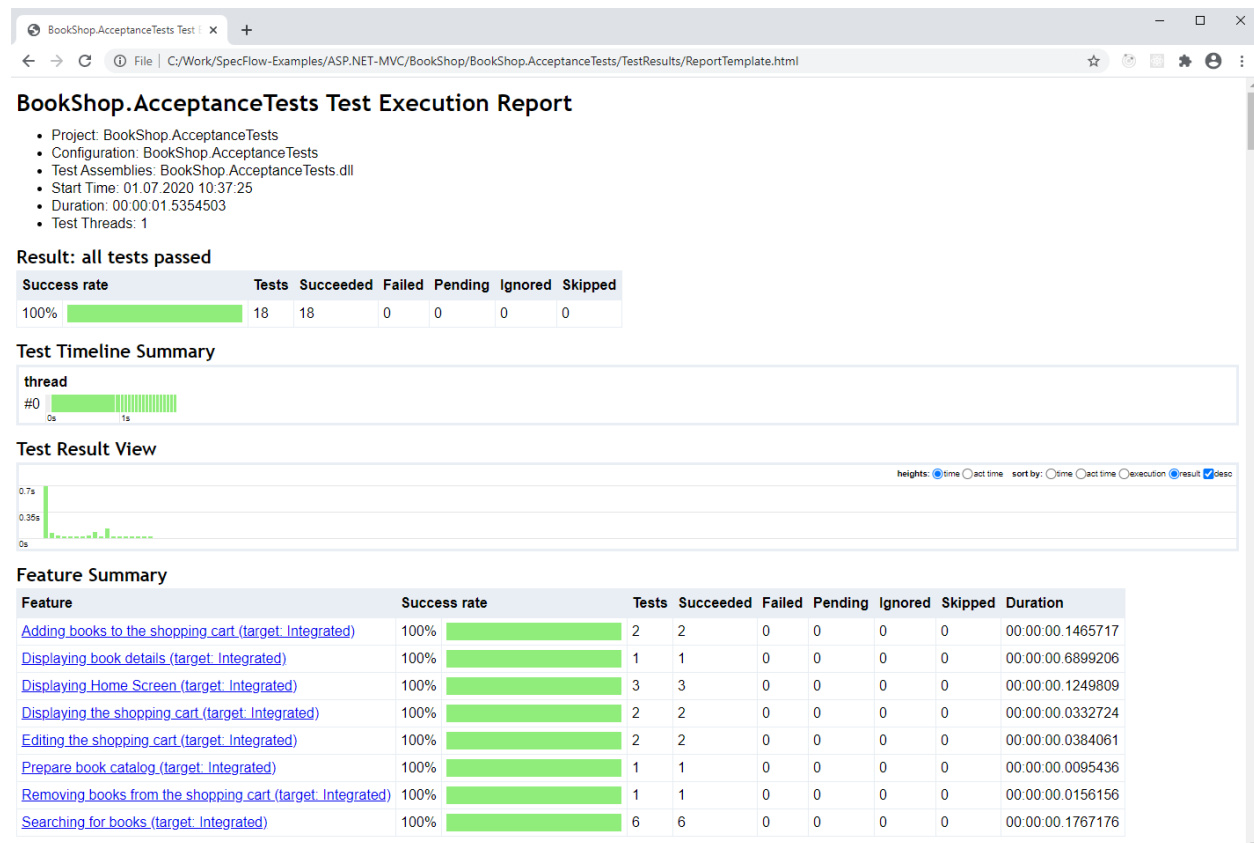
```

Scenario: Changing quantity of book to 0 should remove book from shopping cart in Editing the shopping cart (target: Integrated) -> Succeeded on thread #0
Scenario: A type of book can be entirely removed from the shopping cart in Removing books from the shopping cart (target: Integrated) -> Succeeded on thread #0
Scenario: Setup basic example books in Prepare book catalog (target: Integrated) -> Succeeded on thread #0
test run finished
publishing test results
Not publishing results.
test results published
generating reports
Starting external report generation process
Successfully generated reports.
Successfully generated reports
Result: all tests passed
Total: 18
Succeeded: 18
Ignored: 0
Pending: 0
Skipped: 0
Failed: 0

Execution Time: 00:00:01.5443238
Report file: file:///C:/Work/SpecFlow-Examples/ASP.NET-MVC/BookShop/TestResults/ReportTemplate.html
Adding attachments to VSTest
adding Test Execution Report 'C:/Work/SpecFlow-Examples/ASP.NET-MVC/BookShop/TestResults/ReportTemplate.html'
Log file: file:///C:/Work/SpecFlow-Examples/ASP.NET-MVC/BookShop/TestResults/BookShop.AcceptanceTests_2020-06-24T144348.log
SpecFlow-Runner execution finished
***** Test run finished: 18 Tests run in 3,3 sec (18 Passed, 0 Failed, 0 Skipped) *****

```

The report contains information about the overall test results as well as a break down of each individual scenario execution.



2.6.7 Automating the Bookshop application with SpecFlow

SpecFlow is completely independent of what level or which interface of the system is automated. When you implement the step bindings you have to decide what the Given/When/Then steps should do to exercise the system and to validate the acceptance criteria.

Unit level automation

In a project where complex business logic is encapsulated in a bunch of classes there might be even an option to validate some acceptance criteria on “unit level”. This level can be also automated with SpecFlow, writing the step definitions accordingly. In this case the Given step could instantiate those classes based on the given preconditions, the When step could execute a method on those classes performing some key business logic, and the Then step could check if the result of the method call meets the expectations.

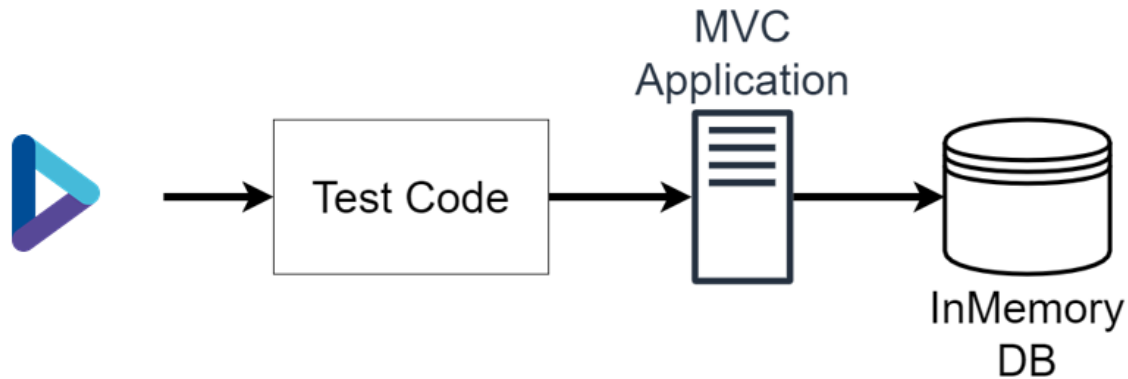
However, unit tests usually focus on implementation details far below the the abstraction level of an acceptance criterion and then it is not feasible to automate those unit tests with SpecFlow.

In the Bookshop example we added some classic unit tests in the `BookShop.UnitTest` project. These are implemented with xUnit and are NOT bound to SpecFlow scenarios.

Controller level automation

The Bookshop example automates the tests directly through the Controller of the MVC web application with SpecFlow (sometimes called *automation below the skin*).

Automating below the skin provides several benefits: less brittle tests, less efforts for automation, better performance of the test suite.



Inside the Controller bindings

Let's examine the scenario in `Book Details.feature` and navigate to the step definitions of the steps (shortcut "F12").

The `Given the following books` step is bound to the `GivenTheFollowingBooks` step definition method in the `BookStep` class. The step definition classes use the *Driver pattern* and *Dependency Injection* to better structure the code into reusable layers and parts. Following the flow of execution to the `DatabaseDriver` the books are inserted into the Entity Framework `DbContext` (using an in-memory database):

```
_dbContext.Books.Add(book);
...
_dbContext.SaveChanges();
```

The `When I open the details of 'Analysis Patterns'` step is bound to the `WhenIOpenTheDetailsOfBook` step definition method in the `BookSteps` class, passing the name of the book as parameter. The implementation is delegated to an `IBookDetailsDriver` implementation, and with the default configuration the `IntegratedBookDetailsDriver` is used. We're calling the `OpenBookDetails` method. Here we can see that our automation directly instantiates the Controller class, calls the action method, and stores the result for the subsequent assertions.

```
using var controller = new CatalogController(_bookLogic);
_result = controller.Details(book.Id);
```

It is important that Controller is instantiated with appropriate dependencies, to ensure that the Given/When/Then steps rely on the same database context and other shared resources.

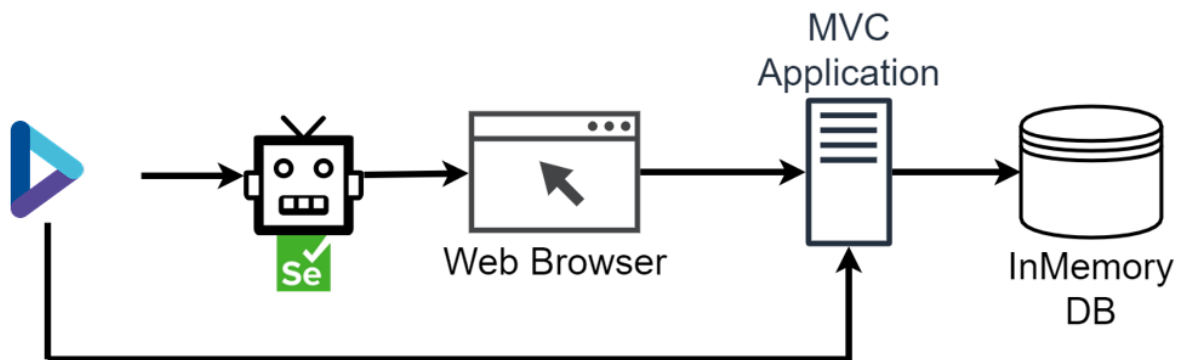
Finally the `Then the book details should show` step is bound to the `ThenTheBookDetailsShouldShow` method in the `BookSteps` class, that again delegates to the `IntegratedBookDetailsDriver`, where we can assert on the previously stored action result.

Note that the reason why these test run relatively fast is that the automation steps perform cheaper in-memory operations, basically working with .NET objects within a single process.

UI level automation with Selenium

Sometimes the behaviour that should be validated cannot be observed on the controller level, but only on the UI. This might range from client side javascript behavior up to server side middleware that is not executed when calling the action methods of the controller classes directly. In those cases the automation of the user interface might be a solution.

In case of e2e UI automation the Given steps can open a browser with Selenium and perform the necessary preparation steps. Still, the boundaries of automation are not necessarily strict. Sometimes ensuring all preconditions through the user interface would be very hard, and it is a feasible tradeoff to manipulate the database or other underlying components directly. The When steps typically perform those key user actions on the UI that are in the focus of the scenario. And finally the Then steps can either validate the results on the UI or, again, could look into the database or internal component directly to validate the expected result.



To demonstrate this approach as well, the Bookshop example contains an alternative automation implementation for all scenarios using Selenium.

Configure the Selenium automation

To enable the tests using Selenium UI automation, you need to add (uncomment) the Chrome target in the Default.srprofile configuration file, while you need to remove (comment) the Integrated target.

```
<Target name="Chrome">
  <DeploymentTransformationSteps>
    <EnvironmentVariable variable="Mode" value="Chrome"/>
  </DeploymentTransformationSteps>
</Target>
```

You also need to have the correct version of Chrome installed, that can be driven by the Selenium version used in this example. It might be necessary to update Chrome or the Selenium version used in this example, to make the UI automation work.

Execute the acceptance tests from the Test Explorer. This time the tests will open a Chrome window and automate the application through Selenium. Notice, however, that the execution of the tests takes significantly longer.

***Note:** You can also enable multiple targets at the same time. In this case SpecFlow+ Runner will generate a unique tests for each combination of target and scenario. In the Test Explorer the name of the test will*

be the same (the title of the scenario), and you can distinguish the tests by their “Traits” (e.g. `Target [Chrome]` vs. `Target [Integrated]`) **Note:** You can also experiment with headless Chrome or Firefox by uncommenting the corresponding targets in the `Default.srprofile` configuration file. However, while the headless Chrome automation is faster than Chrome, the Firefox automation runs very slowly.

Inside the Selenium bindings

Let's examine the same scenario in `Book Details.feature` again and compare the Selenium automation with the Controller automation.

We have seen before that the `Given the following books` step is bound to the `GivenTheFollowingBooks` step definition method and at the end the `DatabaseDriver` inserts the books into the database. There is no difference here.

However, in case of the `When I open the details of 'Analysis Patterns'` step now a different implementation of `IBookDetailsDriver` interface is configured due to our changes in the configuration file. Instead of the `IntegratedBookDetailsDriver` the `SeleniumBookDetailsDriver` is used. In the `OpenBookDetails` method of `SeleniumBookDetailsDriver` we can see that our automation interacts with the `BrowserDriver` and `WebServerDriver`, where the first one automates the browser opening the appropriate URL, while the second one automates the web server starting a new instance of the Bookshop application with Kestrel.

The `Then the book details should show` step is also routed to the `SeleniumBookDetailsDriver`. In the `ShowBookDetails` method the result is validated in the browser. We use the *page object pattern* to encapsulate the UI details in the `BookDetailPageObject` class, e.g. how the title of the book can be found in the rendered page with Selenium. This way the driver can formulate the expectations on a higher level:

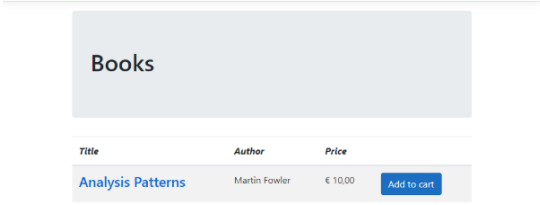
```
var bookDetailPageObject = new BookDetailPageObject(_browserDriver.Current);

if (expectedBook.Title != null)
{
    bookDetailPageObject.Title.Should().Be(expectedBook.Title);
}
```

Notice that the phrasing of the scenarios didn't have to be changed, in order to automate on a different layer. This is a good practice, as SpecFlow scenarios shouldn't express technical details of the automation, but the intention and behaviour to be validated.

Extended report with screenshots from the UI

The Bookshop example extends the SpecFlow+ Runner execution report with screenshots from the user interface taken during the UI automation. This is especially useful if a UI automated scenario breaks, because the screenshot might provide an immediate clue about the root cause of the failure.

When I search for books by the phrase 'Fowler'	done: SearchSteps.WhenISearchForBooksByThePhrase("Fowler") (0,3s)	Succeeded in 0.535s
 <p>The screenshot shows a web application titled 'Books'. It features a table with columns 'Title', 'Author', and 'Price'. A single row is displayed with the title 'Analysis Patterns', author 'Martin Fowler', and price '€ 10,00'. There is a blue 'Add to cart' button next to the price. The application has a navigation bar with links for 'BookShop.Mvc', 'Home', and 'Privacy'.</p>		

After each scenario step a screenshot is taken from the browser and saved into the output directory as a new file. For the implementation details see the `Screenshots.MakeScreenshotAfterStep` method with the `[AfterStep]` attribute. The name of the screenshot file is written into the trace output using `Console.WriteLine`.

The default report template is overridden in the `Default.srprofile` configuration in the `TestProfile/Report/Template` element. The customized `ReportTemplate.cshtml` replaces the screenshot text in the trace output with the image link.

2.6.8 Executing tests from the command line

While Visual Studio provides several convenience features when working with SpecFlow (syntax coloring, navigation, integration with the Test Explorer, etc.), you can easily run the automated tests from the command line too.

- Open a command line terminal where you can execute `.NET Core CLI` commands
- Set the current directory to the root directory of the Bookshop example, where the `BookShop.sln` solution file is located:

```
cd SpecFlow-Examples\ASP.NET-MVC\BookShop
```

- Build the solution

```
dotnet build
```

- Run all tests in the solution

```
dotnet test
```

***Note:** You can also skip the `dotnet build` step and run the tests immediately with `dotnet test`, because this command also (re-)builds the project. However it hides the details of the build output. We outlined the build as a separate step here as a best practice when examining a new project, because separating the steps makes the understanding of the output and potential troubleshooting easier.*

Note that if you run `dotnet test` for the entire Bookshop solution then both the unit tests and the acceptance tests are executed.

The SpecFlow+ Runner execution reports and logs are generated in the “results directory” of the `dotnet test` command. The default is the `TestResults` folder in the directory of the solution/project, but it can be overridden with the `-r|--results-directory <PATH>` option of `dotnet test`.

Please consult the documentation of the `dotnet test` command for further details.

Further `dotnet test` examples

The following examples guide you through some typical questions/scenarios when running the Bookshop acceptance tests from the command line using `dotnet test`. Feel free to experiment with other combinations of parameters and consult the documentation of `dotnet test`.

Run all acceptance test

Run only the acceptance tests (and ignore the unit tests) from the root folder of the Bookshop sample:

```
dotnet test BookShop.AcceptanceTests
```

Note: the default `TestResults` test results directory of `dotnet test` is relative to the project, hence in this case the reports and logs are generated into the `BookShop.AcceptanceTests\TestResults` folder.

Alternatively you can run on the tests for the entire solution and use a filter to include the acceptance tests only:

```
dotnet test --filter BookShop.AcceptanceTests
```

Note: in this case `dotnet test` still discovers both the unit test and acceptance test projects separately and emits a warning for the unit tests that “*no test matches the given testcase filter*”:

You can also specify the project file explicitly:

```
dotnet test .\BookShop.AcceptanceTests\BookShop.AcceptanceTests.csproj
```

Or you can specify the test assembly (dll) explicitly:

```
dotnet test .\BookShop.AcceptanceTests\bin\Debug\netcoreapp3.1\BookShop.AcceptanceTests.  
↪dll
```

Run acceptance tests without re-building the project

Assuming the project was built successfully already:

```
dotnet test BookShop.AcceptanceTests --no-build
```

This speeds up the test execution command as the build step is skipped. It is also useful to limit the output of the command to the test execution details.

Set output detail level

Run tests with more detailed output (similar detail level like the Visual Studio output):

```
dotnet test BookShop.AcceptanceTests --no-build -v n
```

Note: if you omit the `--no-build` option the output will also contain the detailed output of the build.

Set the output folder

Save the execution report and logs to a different folder:

```
dotnet test BookShop.AcceptanceTests -r C:\CentralTestResults\Bookshop
```

Filter tests

Please also consult the documentation of [filter options](#) of the `dotnet test` command for more insights. With SpecFlow+ Runner the `Name` and `TestCategory` properties can be used to filter the acceptance tests.

***Tip:** You can list all acceptance tests with the `dotnet test BookShop.AcceptanceTests -t command`. The tests are listed by the `Name` property. This can help to check the naming convention and to construct the desired filter by `Name` (e.g. `--filter Name~"Author should be matched in Searching for books"`).*

Run the “automated” scenarios (tagged as @automated)

```
dotnet test BookShop.AcceptanceTests --filter TestCategory=automated
```

See the `@automated` tag in the feature files

Run scenarios associated with work item 11 (tagged as @WI11)

```
dotnet test BookShop.AcceptanceTests --filter TestCategory=WI11`
```

See the `@WI11` tag on the feature in `Features\Shopping Cart\Add to.feature`

Run scenarios associated with work item 12 or 13 (tagged as @WI12 or @WI13)

```
dotnet test BookShop.AcceptanceTests --filter "TestCategory=WI12|TestCategory=WI13"
```

See the `@WI12` and `@WI13` tags on the scenarios in `Features\Shopping Cart\Add to.feature`.

We combined two filter expressions with the `|` (OR) operator. See the [filter options](#) documentation of `dotnet test` for the string matching and conditional operators.

Run all scenarios related to the “shopping cart”

```
dotnet test BookShop.AcceptanceTests --filter Name~"shopping cart"
```

This command runs all scenarios where the feature or the scenario title contains the term “shopping cart”.

Note: you have to use the `~` (contains) operator to match the `Name` property.

Run all scenarios of the feature “Adding books to the shopping card”

```
dotnet test BookShop.AcceptanceTests --filter Name~"Adding books to the shopping card"
```

See the feature in `Features\Shopping Cart\Add to.feature`. Note: you have to use the `~` (contains) operator to match the `Name` property. Note: in practice feature titles and scenario titles are so unique that it is unlikely that another scenario/feature title contains the *whole title* of your selected feature.

Run a single scenario "Author should be matched" in the "Searching for books" feature

Let's look at 3 different solutions, as the used matching strategy gets more and more strict.

- Filter by scenario title only

```
dotnet test BookShop.AcceptanceTests --filter Name~"Author should be matched"
```

- Note: you have to use the ~ (contains) operator to match the Name property.
- Note: in practice feature titles and scenario titles are so unique that it is unlikely that another scenario/feature title contains the *whole title* of your selected scenario.

- Filter by scenario title AND feature title

```
dotnet test BookShop.AcceptanceTests --filter Name~"Author should be matched in_
↳ Searching for books"
```

- Note: you have to add the in word between the scenario title and feature title. This is how the Name property of the test is built.
- Note: you have to use the ~ (contains) operator to match the Name property.

- Filter by scenario title AND feature title AND target (= the full name)

```
dotnet test BookShop.AcceptanceTests --filter Name="Author should be matched in_
↳ Searching for books \ (target: Integrated)\"
```

- When using the targets feature of SpecFlow+ Runner the same scenario can be executed on different targets, hence the target is also included in the name of the test.
- Note: For this example the Integrated target must be enabled in the Default.srprofile.
- Note: here you can filter with exact match using the = (equals) operator to match the Name property, because you use the full name in the filter.
- Note: the filter syntax of dotnet test recognizes parenthesis to enclose conditional operators. To match the string (target: Integrated) in the name we have to escape the parenthesis with a preceding \ (backslash) character.
- Provided that you enabled also the Chrome target in the Default.srprofile you can execute the same test with the Chrome UI automation as:

```
dotnet test BookShop.AcceptanceTests --filter Name="Author should be matched in_
↳ Searching for books \ (target: Chrome)\"
```

Run all scenarios with the Controller level automation (and skip UI automation targets) to get a quick test result

```
dotnet test BookShop.AcceptanceTests --filter Name~"target: Integrated"
- Note: For this example the `Integrated` target must be enabled in the `Default.
↳ srprofile
```

Filter by FullyQualifiedName

With `dotnet test` it is possible to filter by the `FullyQualifiedName` property as `--filter FullyQualifiedName~"some text"` and this is equivalent with the shorthand form of `--filter "some text"` (when omitting the property name in the filter).

However, the `FullyQualifiedName` property has a more complex naming convention due to technical requirements. Unfortunately it is not possible to list the tests by the `FullyQualifiedName` property with `dotnet test`, which makes the practical usage of a `FullyQualifiedName` filter even harder.

Hence in the command line **we recommend to filter by the Name property** if you (ad-hoc) want to run a feature or scenario during development, **or by the TestCategory property** when running a defined sets of scenarios using tags. It is also a common practice to add a `@wip` tag to the scenarios under active development in your workspace, because this way you can easily identify and run them (you can also change the grouping in Visual Studio Text Explorer to group the tests by Traits).

To demonstrate some of the challenges with `FullyQualifiedName` try to filter for the “Author should be matched” scenario. The following filter **does not find the scenario**, because the `FullyQualifiedName` does not literally contain the title of the scenario in this form:

```
# NOT WORKING:
dotnet test BookShop.AcceptanceTests --filter "Author should be matched"
```

You can fix the filter as follows:

```
dotnet test BookShop.AcceptanceTests --filter "Author+should+be+matched"
```

The following command exactly matches the `FullyQualifiedName` of the scenario to demonstrate the structure of the `FullyQualifiedName` property:

```
dotnet test BookShop.AcceptanceTests --filter FullyQualifiedName="BookShop.
↳AcceptanceTests.Searching for books.#\(\)::Target:Integrated/TestAssembly:BookShop.
↳AcceptanceTests/Feature:Searching+for+books/Scenario:Author+should+be+matched"
```

2.6.9 Living Documentation

To get the most out of SpecFlow, **get feedback early on** and providing the basis for further discussions about the behavior of your system, we recommend to share your Gherkin Feature Files with all your stakeholders and team members.

An easy way to share your Gherkin Feature Files is to use the **free** SpecFlow+ LivingDoc:

- [Generator](#) for local or self-hosted documentation
- [Azure DevOps Extension](#) to quickly generate a living documentation from your Gherkin Feature Files on Azure DevOps.

Demo: Try out our [SpecFlow+ LivingDoc Generator Demo](#) which is hosted on GitHub Pages.

The generated documentation can finally be shared per Email, per Microsoft Teams or Slack without the need for Visual Studio.

Sounds interesting? **Let's get started** with [SpecFlow+ LivingDoc](#).

2.7 Requirements

2.7.1 .NET

.NET

- .NET 5

.NET Core

- .NET Core 2.1
- .NET Core 3.1

.NET Framework

- >= .NET Framework 4.6.1

2.7.2 Visual Studio

Version

- Visual Studio 2017 **or**
- Visual Studio 2019

We recommend using always the latest version of Visual Studio

Workloads

- ASP.NET and web development **or**
- .NET Desktop environment **or**
- .NET Core cross- platform development

2.7.3 Project Setup

- Project folder should not be too deep in the filesystem, as you will get problems with Windows 255 character limit in file paths

2.8 Installation

Note: If you are new to SpecFlow, we recommend checking out the [Getting Started guide](#) first. It will take you through the process of installing SpecFlow and setting up your first project and tests in Visual Studio.

SpecFlow consists of three components:

- The **IDE Integration** that provides a customized editor and test generation functions within your IDE.
- The **generator** that can turn Gherkin specifications into executable test classes, available from NuGet.
- The **runtime** required for executing the generated tests. There are different runtime assemblies compiled for different target platforms. These packages are also available from NuGet.

In order to install everything you need, you first have to install the IDE integration and then set up your project to work with SpecFlow using the NuGet packages.

2.8.1 Installing the IDE Integration

The process of installing the IDE Integration packages depends on your IDE.

Visual Studio

We recommend installing the SpecFlow Visual Studio extension (IDE Integration), as this is the most convenient way of working with SpecFlow. Find out how to install and use this extension [here](#).

Rider

Rider has an official SpecFlow plugin, find out how to install and use it [here](#).

MonoDevelop/XamarinStudio/Visual Studio for Mac

We don't maintain our own extension for MonoDevelop/XamarinStudio/Visual Studio for Mac. But our amazing community created one at <https://github.com/straightight/SpecFlow-VS-Mac-Integration>.

VSCode

We currently don't have our own extension for VSCode.

For creating new projects, we recommend to use our [.NET templates](#).

To improve your developing and scenario writing experience, we recommend following VSCode extensions:

- [Cucumber \(Gherkin\) Full Support](#) - for editing feature files
- [.NET Core Test Explore](#) - for executing scenarios

2.8.2 Project Setup

The generator and runtime are usually installed together for each project. To install the NuGet packages:

1. Right-click on your project in Visual Studio, and select **Manage NuGet Packages** from the menu.
2. Switch to the **Browse** tab.
3. Enter "SpecFlow" in the search field to list the available packages for SpecFlow.
4. Install the required NuGet packages. Depending on your chosen unit test provider, you have to use different packages. See [this list](#) to find the correct package

2.9 NuGet Packages

There are a number of [NuGet](#) packages supplied for SpecFlow:

- **SpecFlow** : The main SpecFlow package. Add this package to your project to install SpecFlow.
- **Unit test provider packages**: These packages are used to configure your unit test provider from SpecFlow 3. You should only install **one** of the packages. Installing more than one package will result in an error. The following packages are available:
 - SpecRun.SpecFlow-3.3.0
 - SpecFlow.xUnit
 - SpecFlow.MsTest
 - SpecFlow.NUnit
- **SpecFlow.Tools.MsBuild.Generation**: This package generates the code-behind files required by SpecFlow using MSBuild. This package is not required prior to SpecFlow 3, but we **strongly recommend** using MSBuild to generate your code behind files with all versions of SpecFlow!

Note: The SpecFlow NuGet package only contains SpecFlow's generator and the runtime components. You still need to [install](#) the IDE integration.

The easiest way to add these packages to your project is to right-click your project and select **Manage NuGet Packages**. You can add SpecFlow to your project with the NuGet Package Management Console with

```
Install-Package SpecFlow -ProjectName myproject
```

The [SpecFlow.CustomPlugin](#) NuGet package can be used to implement custom *plugins* for SpecFlow.

2.9.1 NuGet packages after 3.0

SpecFlow

<https://www.nuget.org/packages/SpecFlow/>

This is the main package of SpecFlow and contains all parts needed at Runtime.

SpecFlow.Tools.MsBuild.Generation

<https://www.nuget.org/packages/SpecFlow.Tools.MsBuild.Generation/>

This package enables the code-behind file generation at build time.

>= 3.0

It is **mandatory** for projects to use. After SpecFlow 3.3.30 this is a dependency of the SpecFlow.xUnit, SpecFlow.NUnit, SpecFlow.MSTest and SpecRun.SpecFlow.3-3-0 packages, hence the package is automatically installed with the unit test provider packages and you don't have to install it manually.

< 3.0

This package is optional if the code-behind file generation is enabled in the Visual Studio Extension. However, we recommend to upgrade to the *MSBuild code behind file generation*.

SpecFlow.xUnit

<https://www.nuget.org/packages/SpecFlow.xUnit/>

>= 3.0

If you want to use SpecFlow with xUnit, you have to use this packages, as it does the configuration for this.

We don't support older versions than xUnit 2.4.0.

< 3.0

This package is optional to use, as all steps can be done manually. It changes automatically the app.config to use xUnit for you and has a dependency on xUnit (>= 2.0).

SpecFlow.MsTest

<https://www.nuget.org/packages/SpecFlow.MsTest/>

>= 3.0

If you want to use SpecFlow with MsTest V2, you have to use this packages, as it does the configuration for this.

We don't support older versions than MsTest V2 1.3.2.

< 3.0

This package is optional to use, as all steps can be done manually. It changes automatically the `app.config` to use MsTest. No additional dependencies are added.

We support MsTest V1 and V2.

SpecFlow.NUnit

<https://www.nuget.org/packages/SpecFlow.NUnit/>

> 3.0

If you want to use SpecFlow with NUnit, you have to use this packages, as it does the configuration for this.

We don't support older versions than NUnit 3.13.1.

< 3.0

This package is optional to use, as all steps can be done manually. It changes automatically the `app.config` to use NUnit and has a dependency on NUnit (≥ 3.0). If you want to use earlier version of NUnit, you have to do the changes manually.

We support NUnit 2 & NUnit 3.

SpecFlow.NUnit.Runners

<https://www.nuget.org/packages/SpecFlow.NUnit.Runners/>

This is a meta-package to install the NUnit.Console package additionally.

SpecFlow.CustomPlugin

<https://www.nuget.org/packages/SpecFlow.CustomPlugin/>

This package is for writing your own runtime or generator plugins.

2.10 Configuration

SpecFlow's behavior can be configured extensively. How to configure SpecFlow depends on the version of SpecFlow you are using.

Note: bear in mind that, although this article is meant to address features default language configuration, you may define language directly on the top of your feature. For further details, refer to [Gerkin's #language directive](#)

2.10.1 SpecFlow 3.x

Starting with SpecFlow 3, you can use the `specflow.json` file to configure it. It is mandatory for .NET Core projects and it is recommended for .NET Framework projects. When using the .NET Framework, you can still use the `app.config` file, as with earlier versions of SpecFlow.

If both the `specflow.json` and `app.config` files are available in a project, `specflow.json` takes precedence.

Please make sure that the **Copy to Output Directory property** of `specflow.json` is set to either **Copy always** or **Copy if newer**. Otherwise `specflow.json` might not get copied to the Output Directory, which results in the configuration specified in `specflow.json` taking no effect during test execution.

2.10.2 SpecFlow 2.x

SpecFlow 2 is configured in your standard .NET configuration file, `app.config`, which is automatically added to your project. This method is not supported by .NET Core, and SpecFlow 2 does not include .NET Core support.

We recommend using `specflow.json` in new projects.

2.10.3 Configuration Options

Both configuration methods use the same options and general structure. The only difference is that SpecFlow 2 only uses the `app.config` file (XML) and SpecFlow 3 requires the `specflow.json` file (JSON) for .NET Core projects.

Configuration examples

The following 2 examples show the same option defined in the `specflow.json` and `app.config` in formats:

specflow.json example:

```
{
  "language": {
    "feature": "de-AT"
  }
}
```

app.config example:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <section name="specFlow" type="TechTalk.SpecFlow.Configuration.
    ConfigurationSectionHandler, TechTalk.SpecFlow" />
  </configSections>
  <specFlow>
    <language feature="de-AT" />
  </specFlow>
</configuration>
```

You can find more examples in the [sample projects](#) for SpecFlow.

2.10.4 Default Configuration

All SpecFlow configuration options have a default setting. Simple SpecFlow projects may not require any further configuration.

2.10.5 Configuring Your Unit Test Provider

SpecFlow 3

You can only configure your unit provider by adding the corresponding packages to your project. You will therefore need to add **one** of the following NuGet packages to your project to configure the unit test provider:

- SpecRun.SpecFlow
- SpecFlow.xUnit
- SpecFlow.MsTest
- SpecFlow.NUnit

Note: Make sure you do not add more than one of the unit test plugins to your project. If you do, an error message will be displayed.

SpecFlow 2

SpecFlow 2 is configured using the `app.config` file (Full Framework only). Enter your unit test provider in the `unitTestProvider` element in the `specflow` section, e.g.:

```
<specFlow>
  <unitTestProvider name="MsTest" />
</specFlow>
```

2.10.6 Configuration Elements

The same configuration elements are available in both the XML (`app.config`) and JSON (`specflow.json`) formats.

language

Use this section to define the default language for feature files and other language-related settings. For more details on language settings, see [Feature Language](#).

bindingCulture

Use this section to define the culture for executing binding methods and converting step arguments. For more details on language settings, see [Feature Language](#).

generator

Use this section to define unit test generation options.

runtime

Use this section to specify various test execution options.

trace

Use this section to determine the SpecFlow trace output.

stepAssemblies

This section can be used to configure additional assemblies that contain *external binding assemblies*. The assembly of the SpecFlow project (the project containing the feature files) is automatically included. The binding assemblies must be placed in the output folder (e.g. bin/Debug) of the SpecFlow project, for example by adding a reference to the assembly from the project.

The following example registers an additional binding assembly (MySharedBindings.dll).

specflow.json example:

```
{
  "stepAssemblies": [
    {
      "assembly": "MySharedBindings"
    }
  ]
}
```

app.config example:

```
<specFlow>
  <stepAssemblies>
    <stepAssembly assembly="MySharedBindings" />
  </stepAssemblies>
</specFlow>
```

The <stepAssemblies> can contain multiple <stepAssembly> elements (one for each assembly), with the following attributes.

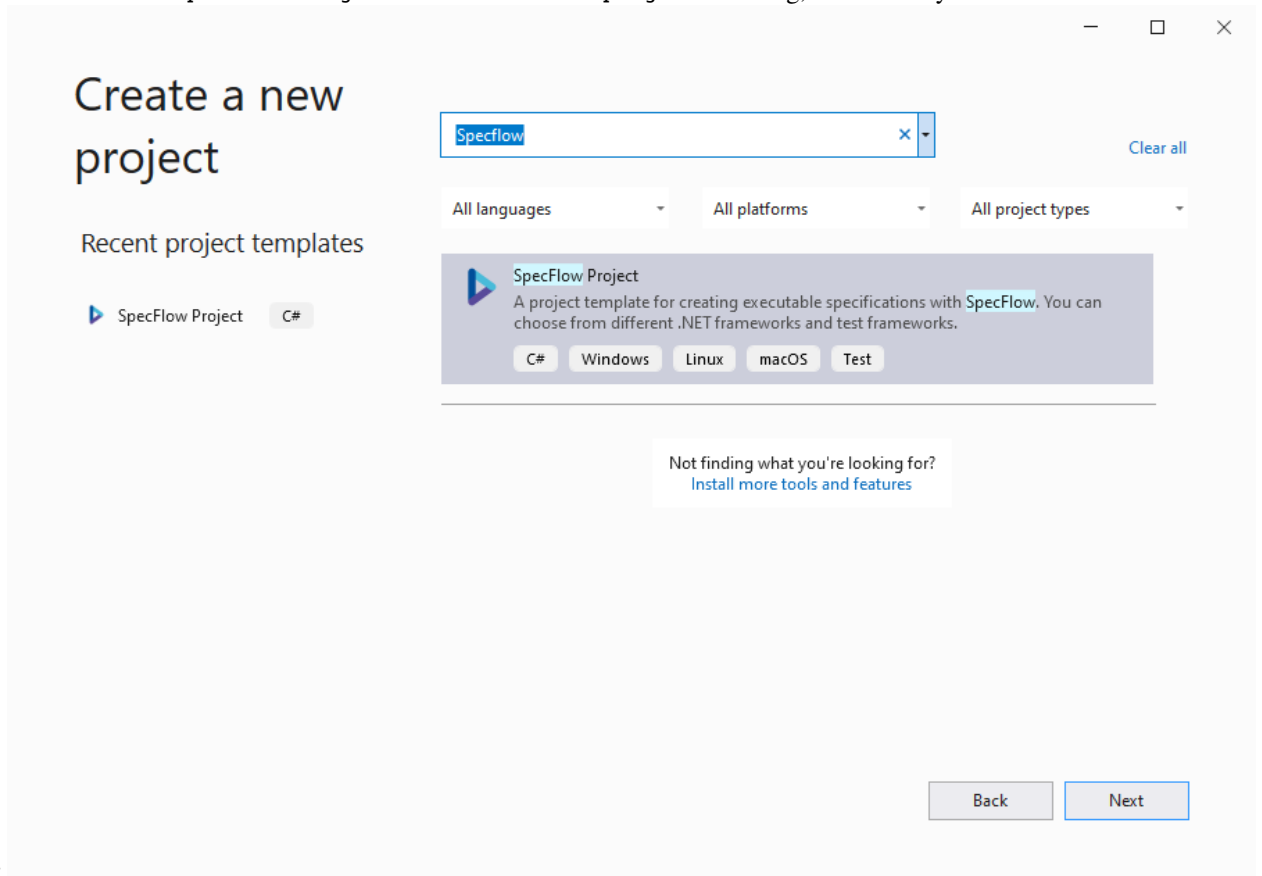
2.11 Project and Item Templates

2.11.1 Visual Studio Templates

It is required to have the [SpecFlow Visual Studio Extension](#) installed.

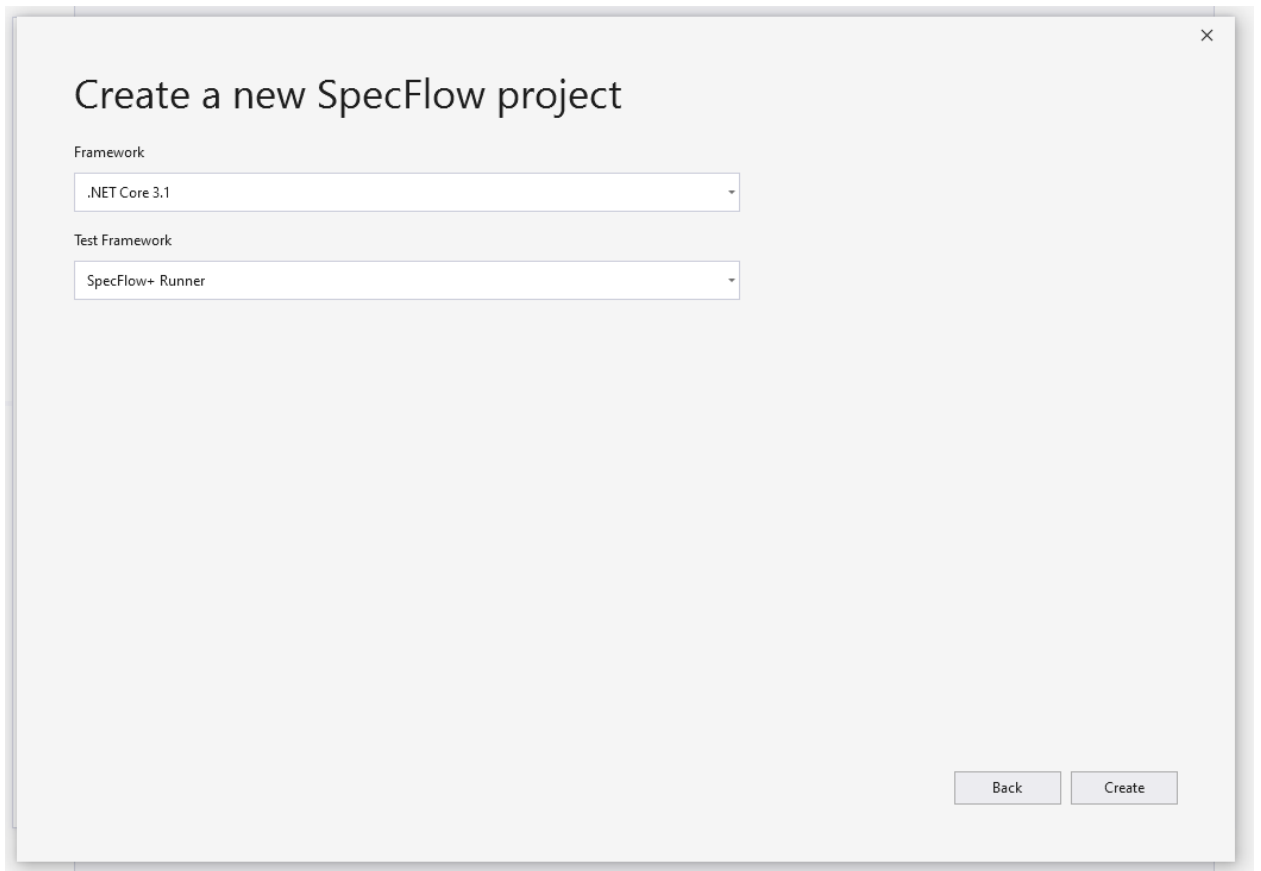
Project Template

You can find the SpecFlow Project in the New project dialog, when you search for



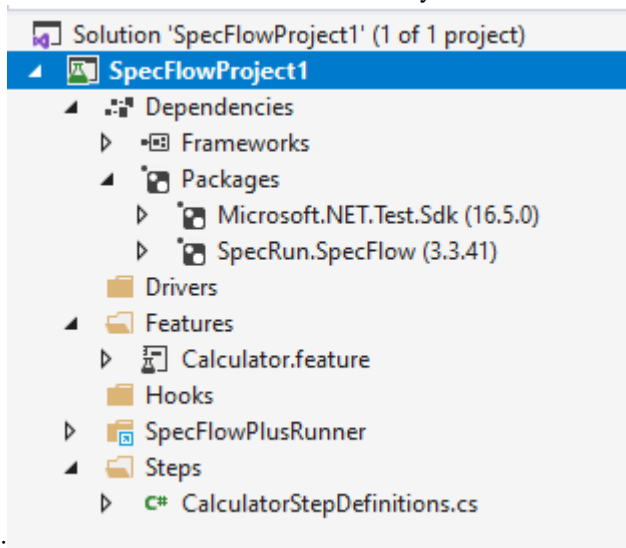
SpecFlow.

After the normal configuration of your project (location, name, ...), you will get to a step to choose your .NET version and test framework for the new



project.

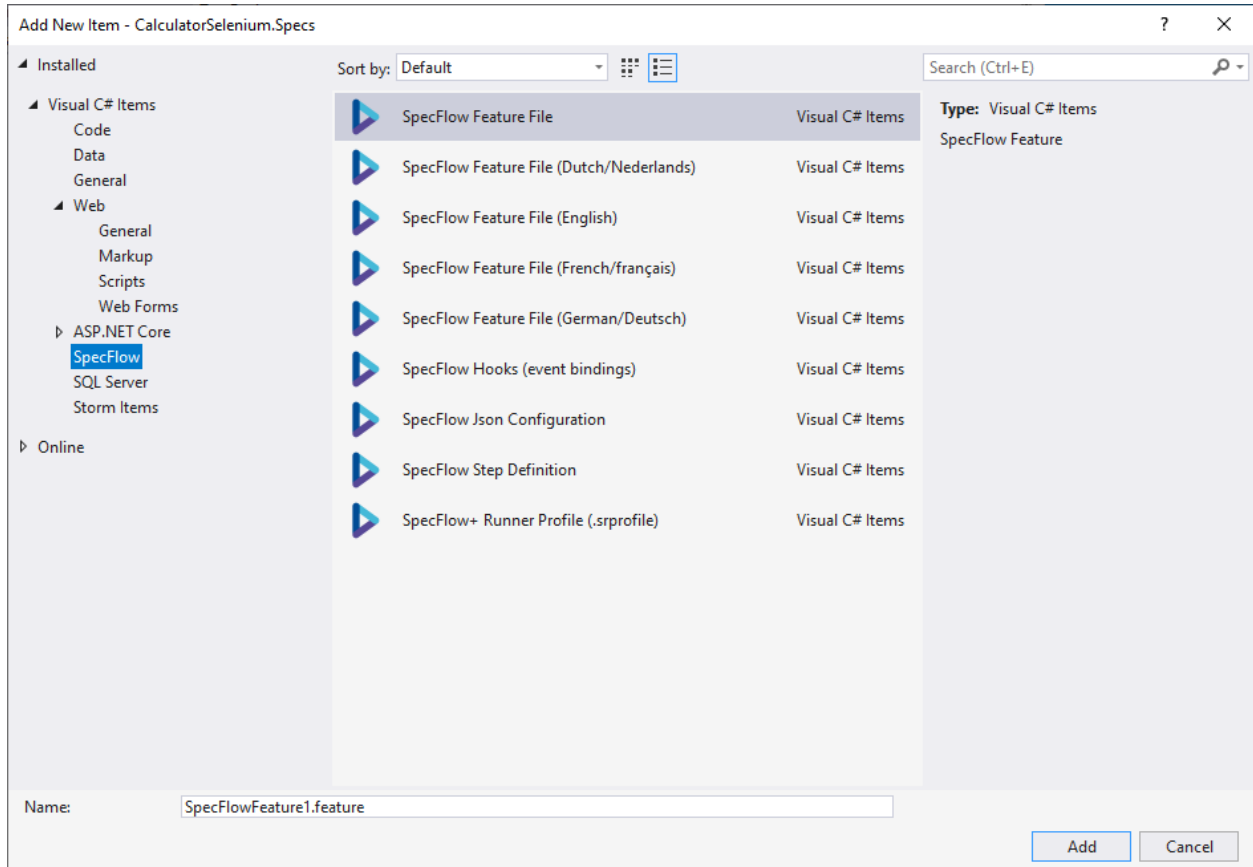
Clicking on Next will create you a new project with all required NuGet



packages:

Item Template

We provide various item templates:



2.11.2 .NET CLI Templates

Note: The [.NET Core SDK](#) is required to be installed in order to use project templates.

- **SpecFlow Feature File:** Gherkin Feature file with an example
- **SpecFlow Feature File (Dutch/Nederlands):** empty Gherkin feature file in dutch
- **SpecFlow Feature File (English):** empty Gherkin feature file in english
- **SpecFlow Feature File (French/français):** empty Gherkin feature file in french
- **SpecFlow Feature File (German/Deutsch):** empty Gherkin feature file in german
- **SpecFlow Hooks (event bindings):** template class with BeforeScenario and AfterScenario hooks
- **SpecFlow Json Configuration:** template for `specflow.json`
- **SpecFlow Step Definition:** step definitions class
- **SpecFlow+ Runner Profile (.srProfile):** configuration file for SpecFlow+ Runner

Installing the Project Template

To install the project template:

1. Open the command line interface of your choice (e.g. cmd or powershell).
2. Execute the following command: `dotnet new -i SpecFlow.Templates.DotNet`
3. The template is installed locally. Once the installation is complete, you can use the template to create new SpecFlow projects.

Creating a New Project from the Template

After installing the templates, you can create a new project using the following command:

```
dotnet new specflowproject
```

By default, a .NET Core project is created with SpecFlow+ Runner configured as the test runner. You can create a project for a different test runner and/or target framework using the following optional parameters:

- **framework:** Determine the target framework for the project. The following options are available:
 - `netcoreapp3.1` (default): .NET Core 3.1
 - `net5.0`: .NET 5.0
 - `net461`: .NET Framework 4.6.1
 - `net462`: .NET Framework 4.6.2
 - `net47`: .NET Framework 4.7
 - `net471`: .NET Framework 4.7.1
 - `net472`: .NET Framework 4.7.2
 - `net48`: .NET Framework 4.8
- **unittestprovider:** Determines the test runner. The following options are available:
 - `specflowplusrunner` (default): SpecFlow+ Runner
 - `xunit`: XUnit
 - `nunit`: NUnit
 - `mstest`: MSTest

Example: `dotnet new specflowproject --unittestprovider xunit --framework netcoreapp3.1`

This creates a new project with XUnit as the unit test provider, and targetting .NET Core 3.1. The project is created with a pre-defined structure that follows best practices. The project includes a single feature file (in the Features folder) and its associated steps (in the Steps folder).

Item Templates

The template pack also includes a few item templates:

- `specflow-feature`: `.feature` file in English
- `specflow-json`: `specflow.json` configuration file
- `specflow-plus-profile`: `Default.srProfile` (SpecFlow+Runner configuration)

If you have additional ideas for the templates, please open a GitHub issue [here](#).

2.12 Unit Test Providers

SpecFlow supports several unit test framework you can use to execute your tests.

To use a specific unit test provider, you have to add it's dedicated NuGet package to your project. You can only have one of these packages added to your project at once.

2.13 Breaking changes with SpecFlow 3.0

2.13.1 Supported test frameworks

We support the following test frameworks/runners:

- SpecFlow+Runner ≥ 3.0
- NUnit ≥ 3.10
- MSTest V2 $\geq 1.3.2$
- xUnit $\geq 2.4.0$

2.13.2 Unit Test Provider is configured via plugin

The `unittestprovider` is no longer configured in `app.config`, but using plugins for the various test frameworks.

It is therefore mandatory to use `SpecRun.SpecFlow-3.0.0`, `SpecFlow.xUnit`, `SpecFlow.MsTest` or `SpecFlow.NUnit` in your project to configure the `unittestprovider`.

2.13.3 GeneratorPlugins are configured with MSBuild

Generator plugins are no longer configured in `app.config`. To load a plugin, you have to add it to the `SpecFlowGeneratorPlugins` item group via MSBuild.

The easiest way is to package your Generator Plugin in a NuGet package and use the automatic import of the `props` and `target` files in them. A good example are the plugins for the different test frameworks.

For example, if we look at the xUnit Plugin: The Generator Plugin is located at `/Plugins/TechTalk.SpecFlow.xUnit.Generator.SpecFlowPlugin`.

In the `/Plugins/TechTalk.SpecFlow.xUnit.Generator.SpecFlowPlugin/build/SpecFlow.xUnit.props` you can add your entry to the `SpecFlowGeneratorPlugins` ItemGroup. Be careful, because dependent on the version of MSBuild you use (Full Framework or .NET Core version), you have to put different assemblies in the ItemGroup. The best way to

do this is in the `/Plugins/TechTalk.SpecFlow.xUnit.Generator.SpecFlowPlugin/build/SpecFlow.xUnit.targets` files. You can use the `MSBuildRuntimeType` property to decide which assembly you want to use.

2.13.4 RuntimePlugins are configured by References

Runtime plugins are also no longer configured in `app.config`. SpecFlow loads now all files in the folder of the test assembly and in the current working directory that end with `.SpecFlowPlugin.dll`.

Because .NET Core doesn't copy references to the target directory, you have to add it to the `None` ItemGroup and set its `CopyToOutputDirectory` to `PreserveNewest`.

You have to do the same decisions as with the generator plugins

2.13.5 specflow.exe is gone

The `specflow.exe` was removed. To generate the code-behind files, please use the `SpecFlow.Tools.MsBuild.Generation` NuGet package.

Reports were removed from the main code-base and aren't currently available for SpecFlow 3.0. For details why we did it and where to find the extracted code is written in GitHub Issue <https://github.com/techtalk/SpecFlow/issues/1036>

2.13.6 Configuration app.config/specflow.json

Configuring SpecFlow via `app.config` is only available when you are using the Full Framework. If you are using .NET Core, you have to use the new `specflow.json` configuration file.

2.14 Gherkin Reference

Gherkin uses a set of special *keywords* to give structure and meaning to executable specifications. Each keyword is translated to many spoken languages; in this reference we'll use English.

Most lines in a Gherkin document start with one of the *keywords*.

Comments are only permitted at the start of a new line, anywhere in the feature file. They begin with zero or more spaces, followed by a hash sign (`#`) and some text.

Block comments are currently not supported by Gherkin.

Either spaces or tabs may be used for indentation. The recommended indentation level is two spaces. Here is an example:

```
Feature: Guess the word

# The first example has two steps
Scenario: Maker starts a game
    When the Maker starts a game
    Then the Maker waits for a Breaker to join

# The second example has three steps
Scenario: Breaker joins a game
    Given the Maker has started a game with the word "silky"
    When the Breaker joins the Maker's game
    Then the Breaker must guess a word with 5 characters
```

The trailing portion (after the keyword) of each step is matched to a code block, called a *step definition*.

2.14.1 Keywords

Each line that isn't a blank line has to start with a Gherkin *keyword*, followed by any text you like. The only exceptions are the feature and scenario descriptions.

The primary keywords are:

- Feature
- Rule (as of Gherkin 6)
- Example (or Scenario)
- Given, When, Then, And, But for steps (or *)
- Background
- Scenario Outline (or Scenario Template)
- Examples

There are a few secondary keywords as well:

- "" (Doc Strings)
- | (Data Tables)
- @ (Tags)
- # (Comments)

Localisation Gherkin is localised for many *spoken languages*; each has their own localised equivalent of these keywords.

Feature

The purpose of the Feature keyword is to provide a high-level description of a software feature, and to group related scenarios.

The first primary keyword in a Gherkin document must always be Feature, followed by a : and a short text that describes the feature.

You can add free-form text underneath Feature to add more description.

These description lines are ignored by SpecFlow at runtime, but are available for reporting (They are included by default in html reports).

Feature: Guess the word

The word guess game is a turn-based game for two players.
The Maker makes a word for the Breaker to guess. The game
is over when the Breaker guesses the Maker's word.

Scenario: Maker starts a game

The name and the optional description have no special meaning to SpecFlow. Their purpose is to provide a place for you to document important aspects of the feature, such as a brief explanation and a list of business rules (general acceptance criteria).

The free format description for Feature ends when you start a line with the keyword Background, Rule, Example or Scenario Outline (or their alias keywords).

You can place tags above Feature to group related features, independent of your file and directory structure.

Tags

Tags are markers that can be assigned to features and scenarios. Assigning a tag to a feature is equivalent to assigning the tag to all scenarios in the feature file.

If supported by the *unit test framework*, SpecFlow generates categories from the tags. The generated category name is the same as the tag's name, but without the leading @. You can filter and group the tests to be executed using these unit test categories. For example, you can tag crucial tests with @important, and then execute these tests more frequently.

If your unit test framework does not support categories, you can still use tags to implement special logic for tagged scenarios in *bindings* by querying the `ScenarioContext.Current.ScenarioInfo.Tags` property.

Scenario and Feature level tags are available by querying the `ScenarioInfo.ScenarioAndFeatureTags` property.

SpecFlow treats the @ignore tag as a special tag. SpecFlow generates an *ignored unit test* method from scenarios with this tag.

Descriptions

Free-form descriptions (as described above for Feature) can also be placed underneath Example/Scenario, Background, Scenario Outline and Rule.

You can write anything you like, as long as no line starts with a keyword.

Rule

The (optional) Rule keyword has been part of Gherkin since v6.

Support in SpecFlow Currently the Visual Studio Extension does not support the Rule keyword. If you use it, your syntax highlighting will be broken.

The purpose of the Rule keyword is to represent one *business rule* that should be implemented. It provides additional information for a feature. A Rule is used to group together several scenarios that belong to this *business rule*. A Rule should contain one or more scenarios that illustrate the particular rule.

For example:

```
# -- FILE: features/gherkin.rule_example.feature
Feature: Highlander

    Rule: There can be only One

        Scenario: Only One -- More than one alive
            Given there are 3 ninjas
            And there are more than one ninja alive
            When 2 ninjas meet, they will fight
            Then one ninja dies (but not me)
            And there is one ninja less alive

        Scenario: Only One -- One alive
            Given there is only 1 ninja alive
```

(continues on next page)

(continued from previous page)

```
Then he (or she) will live forever ;-)
```

```
Rule: There can be Two (in some cases)
```

```
Scenario: Two -- Dead and Reborn as Phoenix
```

```
...
```

Scenario

This is a *concrete example* that *illustrates* a business rule. It consists of a list of *steps*.

The keyword **Scenario** is a synonym of the keyword **Example**.

You can have as many steps as you like, but we recommend you keep the number at 3-5 per example. Having too many steps in an example, will cause it to lose its expressive power as specification and documentation.

In addition to being a specification and documentation, an example is also a *test*. As a whole, your examples are an *executable specification* of the system.

Examples follow this same pattern:

- Describe an initial context (**Given** steps)
- Describe an event (**When** steps)
- Describe an expected outcome (**Then** steps)

Steps

Each step starts with **Given**, **When**, **Then**, **And**, or **But**.

SpecFlow executes each step in a scenario one at a time, in the sequence you've written them in. When SpecFlow tries to execute a step, it looks for a matching step definition to execute.

Keywords are not taken into account when looking for a step definition. This means you cannot have a **Given**, **When**, **Then**, **And** or **But** step with the same text as another step.

SpecFlow considers the following steps duplicates:

```
Given there is money in my account
```

```
Then there is money in my account
```

This might seem like a limitation, but it forces you to come up with a less ambiguous, more clear domain language:

```
Given my account has a balance of £430
```

```
Then my account should have a balance of £430
```

Given

Given steps are used to describe the initial context of the system - the *scene* of the scenario. It is typically something that happened in the *past*.

When SpecFlow executes a Given step, it will configure the system to be in a well-defined state, such as creating and configuring objects or adding data to a test database.

The purpose of Given steps is to **put the system in a known state** before the user (or external system) starts interacting with the system (in the **When** steps). Avoid talking about user interaction in Given's. If you were creating use cases, Given's would be your preconditions.

It's okay to have several Given steps (use **And** or **But** for number 2 and upwards to make it more readable).

Examples:

- Mickey and Minnie have started a game
- I am logged in
- Joe has a balance of £42

When

When steps are used to describe an event, or an *action*. This can be a person interacting with the system, or it can be an event triggered by another system.

It's strongly recommended you only have a single When step per Scenario. If you feel compelled to add more, it's usually a sign that you should split the scenario up into multiple scenarios.

Examples:

- Guess a word
- Invite a friend
- Withdraw money

Imagine it's 1922 Most software does something people could do manually (just not as efficiently).

Try hard to come up with examples that don't make any assumptions about technology or user interface. Imagine it's 1922, when there were no computers.

Implementation details should be hidden in the *step definitions*.

Then

Then steps are used to describe an *expected* outcome, or result.

The *step definition* of a Then step should use an *assertion* to compare the *actual* outcome (what the system actually does) to the *expected* outcome (what the step says the system is supposed to do).

An outcome *should* be on an **observable** output. That is, something that comes *out* of the system (report, user interface, message), and not a behavior deeply buried inside the system (like a record in a database).

Examples:

- See that the guessed word was wrong
- Receive an invitation
- Card should be swallowed

While it might be tempting to implement Then steps to look in the database - resist that temptation!

You should only verify an outcome that is observable for the user (or external system), and changes to a database are usually not.

And, But

If you have successive Given's, When's, or Then's, you *could* write:

```
Scenario: Multiple Givens
  Given one thing
  Given another thing
  Given yet another thing
  When I open my eyes
  Then I should see something
  Then I shouldn't see something else
```

Or, you could make the example more fluidly structured by replacing the successive Given's, When's, or Then's with And's and But's:

```
Scenario: Multiple Givens
  Given one thing
  And another thing
  And yet another thing
  When I open my eyes
  Then I should see something
  But I shouldn't see something else
```

*

Gherkin also supports using an asterisk (*) in place of any of the normal step keywords. This can be helpful when you have some steps that are effectively a *list of things*, so you can express it more like bullet points where otherwise the natural language of And etc might not read so elegantly.

For example:

```
Scenario: All done
  Given I am out shopping
  And I have eggs
  And I have milk
  And I have butter
  When I check my list
  Then I don't need anything
```

Could be expressed as:

```
Scenario: All done
  Given I am out shopping
  * I have eggs
  * I have milk
  * I have butter
  When I check my list
  Then I don't need anything
```

Background

Occasionally you'll find yourself repeating the same Given steps in all of the scenarios in a Feature.

Since it is repeated in every scenario, this is an indication that those steps are not *essential* to describe the scenarios; they are *incidental details*. You can literally move such Given steps to the background, by grouping them under a Background section.

A Background allows you to add some context to the scenarios that follow it. It can contain one or more Given steps, which are run before *each* scenario, but after any *Before hooks*.

A Background is placed before the first Scenario/Example, at the same level of indentation.

For example:

```
Feature: Multiple site support
  Only blog owners can post to a blog, except administrators,
  who can post to all blogs.

  Background:
    Given a global administrator named "Greg"
    And a blog named "Greg's anti-tax rants"
    And a customer named "Dr. Bill"
    And a blog named "Expensive Therapy" owned by "Dr. Bill"

  Scenario: Dr. Bill posts to his own blog
    Given I am logged in as Dr. Bill
    When I try to post to "Expensive Therapy"
    Then I should see "Your article was published."

  Scenario: Dr. Bill tries to post to somebody else's blog, and fails
    Given I am logged in as Dr. Bill
    When I try to post to "Greg's anti-tax rants"
    Then I should see "Hey! That's not your blog!"

  Scenario: Greg posts to a client's blog
    Given I am logged in as Greg
    When I try to post to "Expensive Therapy"
    Then I should see "Your article was published."
```

For a less explicit alternative to Background, check out *scoped step definitions*.

Tips for using Background

- Don't use Background to set up **complicated states**, unless that state is actually something the client needs to know.
 - For example, if the user and site names don't matter to the client, use a higher-level step such as Given I am logged in as a site owner.
- Keep your Background section **short**.
 - The client needs to actually remember this stuff when reading the scenarios. If the Background is more than 4 lines long, consider moving some of the irrelevant details into higher-level steps.
- Make your Background section **vivid**.

- Use colourful names, and try to tell a story. The human brain keeps track of stories much better than it keeps track of names like "User A", "User B", "Site 1", and so on.
- Keep your scenarios **short**, and don't have too many.
- If the Background section has scrolled off the screen, the reader no longer has a full overview of what's happening. Think about using higher-level steps, or splitting the *.feature file.

Scenario Outline

The Scenario Outline keyword can be used to run the same Scenario multiple times, with different combinations of values.

The keyword Scenario Template is a synonym of the keyword Scenario Outline.

Copying and pasting scenarios to use different values quickly becomes tedious and repetitive:

```
Scenario: eat 5 out of 12
  Given there are 12 cucumbers
  When I eat 5 cucumbers
  Then I should have 7 cucumbers

Scenario: eat 5 out of 20
  Given there are 20 cucumbers
  When I eat 5 cucumbers
  Then I should have 15 cucumbers
```

We can collapse these two similar scenarios into a Scenario Outline.

Scenario outlines allow us to more concisely express these scenarios through the use of a template with < >-delimited parameters:

```
Scenario Outline: eating
  Given there are <start> cucumbers
  When I eat <eat> cucumbers
  Then I should have <left> cucumbers

Examples:
| start | eat | left |
| 12   | 5   | 7   |
| 20   | 5   | 15  |
```

A Scenario Outline must contain an Examples (or Scenarios) section. Its steps are interpreted as a template which is never directly run. Instead, the Scenario Outline is run *once for each row* in the Examples section beneath it (not counting the first header row).

The steps can use <> delimited *parameters* that reference headers in the examples table. SpecFlow will replace these parameters with values from the table *before* it tries to match the step against a step definition.

> **Note:** Tables used in Examples must have **unique headers**. Using duplicate headers will result in errors.

Hint: In certain cases, when generating method names using the regular expression method, SpecFlow is unable to generate the correct parameter signatures for unit test logic methods without a little help. Placing single quotation marks (') around placeholders (eg. '<placeholder>') improves SpecFlow's ability to parse the scenario outline and generate more accurate regular expressions and test method signatures.

You can also use parameters in *multiline step arguments*.

2.14.2 Step Arguments

In some cases you might want to pass more data to a step than fits on a single line. For this purpose Gherkin has Doc Strings and Data Tables.

Doc Strings

Doc Strings are handy for passing a larger piece of text to a step definition.

The text should be offset by delimiters consisting of three double-quote marks on lines of their own:

```
Given a blog post named "Random" with Markdown body
    """
    Some Title, Eh?
    =====
    Here is the first paragraph of my blog post. Lorem ipsum dolor sit amet,
    consectetur adipiscing elit.
    """
```

In your step definition, there's no need to find this text and match it in your pattern. It will automatically be passed as the last argument in the step definition.

Indentation of the opening """ is unimportant, although common practice is two spaces in from the enclosing step. The indentation inside the triple quotes, however, is significant. Each line of the Doc String will be dedented according to the opening """. Indentation beyond the column of the opening """ will therefore be preserved.

Data Tables

Data Tables are handy for passing a list of values to a step definition:

```
Given the following users exist:
| name | email | twitter |
| Aslak | aslak@cucumber.io | @aslak_hellesoy |
| Julien | julien@cucumber.io | @jbpros |
| Matt | matt@cucumber.io | @mattwynne |
```

Just like Doc Strings, Data Tables will be passed to the step definition as the first argument.

SpecFlow provides a rich API for manipulating tables from within step definitions. See the [Assist Namespace](#) reference for more details.

2.14.3 Markdown

You can include markdown code in your feature files with the standard Markdown features such as bold, italic, lists etc. This is useful when generating documentation via the SpecFlow+ LivingDoc generator.

Click [here](#) to find out more.

2.14.4 Spoken Languages

The language you choose for Gherkin should be the same language your users and domain experts use when they talk about the domain. Translating between two languages should be avoided.

This is why Gherkin has been translated to over 70 languages.

Here is a Gherkin scenario written in Norwegian:

```
# language: no
Funksjonalitet: Gjett et ord

Eksempel: Ordmaker starter et spill
  Når Ordmaker starter et spill
  Så må Ordmaker vente på at Gjetter blir med

Eksempel: Gjetter blir med
  Gitt at Ordmaker har startet et spill med ordet "bløtt"
  Når Gjetter blir med på Ordmakers spill
  Så må Gjetter gjette et ord på 5 bokstaver
```

A `# language:` header on the first line of a feature file tells SpecFlow what spoken language to use - for example `# language: fr` for French. If you omit this header, SpecFlow will default to English (`en`).

You can also define the language in the [configuration file](#).

Gherkin Dialects

In order to allow Gherkin to be written in a number of languages, the keywords have been translated into multiple languages. To improve readability and flow, some languages may have more than one translation for any given keyword.

Overview

You can find all translation of Gherkin [on GitHub](#). This is also the place to add or update translations.

Note Big parts of this page were taken over from <https://cucumber.io/docs/gherkin/reference/>.

2.15 Feature Language

To avoid communication errors introduced by translations, it is recommended to keep the specification and the acceptance test descriptions in the language of the business. The Gherkin format supports many natural languages besides English, like German, Spanish or French. More details on the supported natural languages are available [here](#).

The language of the feature files can either be specified globally in your configuration (see [language element](#)), or in the feature file's header using the `#language` syntax. Specify the language using the ISO language names used by the `CultureInfo` class of the .NET Framework (e.g. `en-US`).

```
#language: de-DE
Funktionalität: Addition
...
```

SpecFlow uses the feature file language to determine the set of keywords used to parse the file, but the language setting is also used as the default setting for converting parameters by the SpecFlow runtime. The culture for binding execution and parameter conversion can be specified explicitly, see [bindingCulture element](#).

As data conversion can only be done using a specific culture in the .NET Framework, we recommend using the specific culture name (e.g. `en-US`) instead of the neutral culture name (e.g. `en`). If a neutral culture is used, SpecFlow uses a specific default culture to convert data (e.g. `en-US` is used to convert data if the `en` language was used).

2.16 Bindings

The *Gherkin feature files* are closer to free-text than to code – they cannot be executed as they are. The automation that connects the specification to the application interface has to be developed first. The automation that connects the Gherkin specifications to source code is called a *binding*. The binding classes and methods can be defined in the SpecFlow project or in *external binding assemblies*.

There are several kinds of bindings in SpecFlow.

2.16.1 Step Definitions

This is the most important one. The *step definition* that automates the scenario at the step level. This means that instead of providing automation for the entire scenario, it has to be done for each separate step. The benefit of this model is that the step definitions can be reused in other scenarios, making it possible to (partly) construct further scenarios from existing steps with less (or no) automation effort.

It is required to add the `[Binding]` attribute to the classes where you define your step definitions.

2.16.2 Hooks

Hooks can be used to perform additional automation logic on specific events, e.g. before executing a scenario.

> **Note:** *Bindings (step definitions, hooks) are global for the entire SpecFlow project.*

2.17 Step Definitions

The step definitions provide the connection between your feature files and application interfaces. You have to add the `[Binding]` attribute to the class where your step definitions are:

```
[Binding]
public class StepDefinitions
{
    ...
}
```

> **Note:** *Bindings (step definitions, hooks) are global for the entire SpecFlow project.*

For better reusability, the step definitions can include parameters. This means that it is not necessary to define a new step definition for each step that just differs slightly. For example, the steps `When I perform a simple search on 'Domain'` and `When I perform a simple search on 'Communication'` can be automated with a single step definition, with `'Domain'` and `'Communication'` as parameters.

The following example shows a simple step definition that matches to the step `When I perform a simple search on 'Domain'`:


```
[When(@"I perform a simple search on '(.*)'")]
public void WhenIPerformASimpleSearchOn(string searchTerm)
{
    var controller = new CatalogController();
    ActionResult = controller.Search(searchTerm);
}
```

Here the method is annotated with the [When] attribute, and includes the regular expression used to match the step's text. This [regular expression](#) uses ((. *)) to define parameters for the method.

2.17.1 Supported Step Definition Attributes

- [Given(regex)] or [Given] - TechTalk.SpecFlow.GivenAttribute
- [When(regex)] or [When] - TechTalk.SpecFlow.WhenAttribute
- [Then(regex)] or [Then] - TechTalk.SpecFlow.ThenAttribute
- [StepDefinition(regex)] or [StepDefinition] - TechTalk.SpecFlow.StepDefinitionAttribute, matches for given, when or then attributes

You can annotate a single method with multiple attributes in order to support different phrasings in the feature file for the same automation logic.

```
[When(@"I perform a simple search on '(.*)'")]
[When(@"I search for '(.*)'")]
public void WhenIPerformASimpleSearchOn(string searchTerm)
{
    ...
}
```

2.17.2 Other Attributes

The [Obsolete] attribute from the system namespace is also supported, check [here](#) for more details.

```
[Given(@"Stuff is done")]
[Obsolete]
public void GivenStuffIsDone()
{
    var x = 2+3;
}
```

2.17.3 Step Definition Methods Rules

- Must be in a public class, marked with the [Binding] attribute.
- Must be a public method.
- Can be either a static or an instance method. If it is an instance method, the containing class will be instantiated once for every scenario.
- Cannot have out or ref parameters.
- Shouldn't have a return type or have Task as return type.

2.17.4 Step Matching Styles & Rules

The rules depend on the step definition style you use.

Regular expressions in attributes

This is the classic and most used way of specifying the step definitions. The step definition method has to be annotated with one or more step definition attributes with regular expressions.

```
[Given(@"I have entered (.*) into the calculator")]
public void GivenIHaveEnteredNumberIntoTheCalculator(int number)
{
    ...
}
```

Regular expression matching rules:

- Regular expressions are always matched to the entire step, even if you do not use the ^ and \$ markers.
- The capturing groups ((...)) in the regular expression define the arguments for the method in order (the result of the first group becomes the first argument etc.).
- You can use non-capturing groups (?:regex) in order to use groups without a method argument.

Method name - underscores

Most of the step definitions can also be specified without regular expression: the method should be named with underscored naming style (or pascal-case, see below) and should be annotated with an empty [Given], [When], [Then] or [StepDefinition] attribute. You can refer to the method parameters with either the parameter name (ALL-CAPS) or the parameter index (zero-based) with the P prefix, e.g. P0.

```
[Given]
public void Given_I_have_entered_NUMBER_into_the_calculator(int number)
{
    ...
}
```

Matching rules:

- The match is case-insensitive.
- Underscore character is matched to one or more non-word character (eg. whitespace, punctuation): \W+.
- If the step contains accented characters, the method name should also contain the accented characters (no substitution).
- The step keyword (e.g. Given) can be omitted: [Given] public void I_have_entered_NUMBER_....
- The step keyword can be specified in the local Gherkin language, or English. The default language can be specified in the *app config* as the feature language or binding culture. The following step definition is therefore a valid "Given" step with German language settings: [When] public void Wenn_ich_addieren_drücke()

More detailed examples can be found in our specs: [StepDefinitionsWithoutRegex.feature](#).

Method name - Pascal-case

Similarly to the underscored naming style, you can also specify the step definitions with Pascal-case method names.

```
[Given]
public void GivenIHaveEnteredNUMBERIntoTheCalculator(int number)
{
    ...
}
```

Matching rules:

- All rules of “Method name - underscores” style applied.
- Any potential word boundary (e.g. number-letter, uppercase-lowercase, uppercase-uppercase) is matching to zero or more non-word character (eg. whitespace, punctuation): `\W*`.
- You can mix this style with underscores. For example, the parameter placeholders can be highlighted this way: `GivenIHaveEntered_P0_IntoTheCalculator(int number)`

Method name - regex (F# only)

F# allows providing any characters in method names, so you can make the regular expression as the method name, if you use [F# bindings](#).

```
let [<Given>] ``I have entered (.*) into the calculator``(number:int) =
    Calculator.Push(number)
```

2.17.5 Parameter Matching Rules

- Step definitions can specify parameters. These will match to the parameters of the step definition method.
- The method parameter type can be `string` or other .NET type. In the later case a [configurable conversion](#) is applied.
- With regular expressions
 - The match groups `((...))` of the regular expression define the arguments for the method based on the order (the match result of the first group becomes the first argument, etc.).
 - You can use non-capturing groups `(?:regex)` in order to use groups without a method argument.
- With method name matching
 - You can refer to the method parameters with either the parameter name (ALL-CAPS) or the parameter index (zero-based) with the P prefix, e.g. `P0`.

2.17.6 Table or Multi-line Text Arguments

If the step definition method should match for steps having *table or multi-line text arguments*, additional `Table` and/or string parameters have to be defined in the method signature to be able to receive these arguments. If both table and multi-line text argument are used for the step, the multi-line text argument is provided first.

Given the following books

Author	Title	
Martin Fowler	Analysis Patterns	
Gojko Adzic	Bridging the Communication Gap	

```
[Given(@"the following books")]
public void GivenTheFollowingBooks(Table table)
{
    ...
}
```

2.18 Hooks

Hooks (event bindings) can be used to perform additional automation logic at specific times, such as any setup required prior to executing a scenario. In order to use hooks, you need to add the `Binding` attribute to your class:

```
[Binding]
public class MyClass
{
    ...
}
```

Hooks are global, but can be restricted to run only for features or scenarios by defining a *scoped binding*, which can be filtered with *tags*. The execution order of hooks for the same type is undefined, unless specified explicitly.

2.18.1 SpecFlow+ Runner Restrictions

When running tests in multiple threads with SpecFlow+ Runner, Before and After hooks such as `BeforeTestRun` and `AfterTestRun` are executed once for each thread. This is a limitation of the current architecture.

If you need to execute specific steps once per test run, rather than once per thread, you can do this using *deployment transformations*. An example can be found [here](#).

2.18.2 Supported Hook Attributes

You can annotate a single method with multiple attributes.

2.18.3 Using Hooks with Constructor Injection

You can use [context injection](#) to access scenario level dependencies in your hook class using constructor injection. For example you can get the `ScenarioContext` injected in the constructor:

```
[Binding]
public class MyHooks
{
    private ScenarioContext _scenarioContext;

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

    [BeforeScenario]
    public void SetupTestUsers()
    {
        //_scenarioContext...
    }
}
```

Note: for static hook methods you can use parameter injection.

2.18.4 Using Hooks with Parameter Injection

You can add parameters to your hook method that will be automatically injected by SpecFlow. For example you can get the `ScenarioContext` injected as parameter in the `BeforeScenario` hook.

```
[Binding]
public class MyHooks
{
    [BeforeScenario]
    public void SetupTestUsers(ScenarioContext scenarioContext)
    {
        //scenarioContext...
    }
}
```

Parameter injection is especially useful for hooks that must be implemented as static methods.

```
[Binding]
public class Hooks
{
    [BeforeFeature]
    public static void SetupStuffForFeatures(FeatureContext featureContext)
    {
        Console.WriteLine("Starting " + featureContext.FeatureInfo.Title);
    }
}
```

In the `BeforeTestRun` hook you can resolve test thread specific or global services/dependencies as parameters.

```
[BeforeTestRun]
public static void BeforeTestRunInjection(ITestRunnerManager testRunnerManager,
    ↪ ITestRunner testRunner)
{
    //All parameters are resolved from the test thread container automatically.
    //Since the global container is the base container of the test thread container,
    ↪ globally registered services can be also injected.

    //ITestRunManager from global container
    var location = testRunnerManager.TestAssembly.Location;

    //ITestRunner from test thread container
    var threadId = testRunner.ThreadId;
}
```

Depending on the type of the hook the parameters are resolved from a container with the corresponding lifecycle.

2.18.5 Hook Execution Order

By default the hooks of the same type (e.g. two `[BeforeScenario]` hook) are executed in an unpredictable order. If you need to ensure a specific execution order, you can specify the `Order` property in the hook's attributes.

```
[BeforeScenario(Order = 0)]
public void CleanDatabase()
{
    // we need to run this first...
}

[BeforeScenario(Order = 100)]
public void LoginUser()
{
    // ...so we can log in to a clean database
}
```

The number indicates the order, not the priority, i.e. the hook with the lowest number is always executed first.

If no order is specified, the default value is 10000. However, we do not recommend on relying on the value to order your tests and recommend specifying the order explicitly for each hook.

Note: If a hook throws an unhandled exception, subsequent hooks of the same type are not executed. If you want to ensure that all hooks of the same types are executed, you need to handle your exceptions manually.

Note: If a `BeforeScenario` throws an unhandled exception then all the scenario steps will be marked as skipped and the `ScenarioContext.ScenarioExecutionStatus` will be set to `TestError`.

2.18.6 Tag Scoping

Most hooks support tag scoping. Use tag scoping to restrict hooks to only those features or scenarios that have *at least one* of the tags in the tag filter (tags are combined with OR). You can specify the tag in the attribute or using [scoped bindings](#).

2.19 Asynchronous Bindings

If you have code that executes an [asynchronous task](#), you can define asynchronous bindings to execute the corresponding code using the `async` and `await` keywords.

A sample project using asynchronous bindings can be found [here](#). The `When` binding in `WebSteps.cs` is asynchronous, and defined as follows:

```
[When(@"I want to get the web page '(.*)'")]
public async Task WhenIWantToGetTheWebPage(string url)
{
    await _webDriver.HttpClientGet(url);
}
```

The `HttpClientGet` asynchronous task is defined in `WebDriver.cs` as follows:

```
public async Task HttpClientGet(string url)
{
    _httpResponseMessage = await _httpClient.GetAsync(url);
}
```

2.20 Scoped Step Definitions

Bindings (step definitions, hooks) are global for the entire SpecFlow project. This means that step definitions bound to a very generic step text (e.g. “When I save the changes”) become challenging to implement. The general solution for this problem is to phrase the scenario steps in a way that the context is clear (e.g. “When I save the **book details**”).

In some cases however, it is necessary to restrict when step definitions or hooks are executed based on certain conditions. SpecFlow's scoped bindings can be used for this purpose.

You can restrict the execution of scoped bindings by:

- tag
- feature (using the feature title)
- scenario (using the scenario title)

The following tags are taken into account for scenario, scenarioblock or step hooks:

- tags defined for the feature
- tags defined for the scenario
- tags defined for the scenario outline
- tags defined for the scenario outline example set (Examples:)

Be careful! Coupling your step definitions to features and scenarios is an anti-pattern. [Read more about it on the Cucumber Wiki](#)

Use the [Scope] attribute to define the scope:

```
[Scope(Tag = "mytag", Feature = "feature title", Scenario = "scenario title")]
```

Navigation from feature files to scoped step definitions is currently not supported by the Visual Studio extension.

2.20.1 Scoping Rules

Scope can be defined at the method or class level.

If multiple criteria (e.g. both tag and feature) are specified in the same [Scope] attribute, they are combined with AND, i.e. all criteria need to match.

Example AND tag scope decoration:

```
[Scope(Tag = "thisTag", Feature = "myFeature")]
```

If multiple [Scope] attributes are defined for the same method or class, the attributes are combined with OR, i.e. at least one of the [Scope] attributes needs to match.

Examples for OR tag scope decoration:

```
[Scope(Tag = "thisTag")] [Scope(Tag = "OrThisTag")]  
[Scope(Tag = "thisTag"), Scope(Tag = "OrThisTag")]
```

Note: Scopes on a different level (class and method) will be combined with OR: defining a [Scope] attribute on class level and defining another [Scope] at method level will cause the attributes to be combined with OR. If you want an AND combination, use a single Scope, e.g.:

```
[Scope(Feature = "feature title", Scenario = "scenario title")]
```

If a step can be matched to both a step definition without a [Scope] attribute as well as a step definition with a [Scope] attribute, the step definition with the [Scope] attribute is used (no ambiguity).

If a step matches several scoped step definitions, the one with the most restrictions is used. For example, if the first step definition contains [Scope(Tag = "myTag")] and the second contains [Scope(Tag = "myTag", Feature = "myFeature")] the second step definition (the more specific one) is used if it matches the step.

If you have multiple scoped step definition with the same number of restrictions that match the step, you will get an ambiguous step binding error. For example, if you have a step definition containing [Scope(Tag = "myTag1", Scenario = "myScenario")] and another containing [Scope(Tag = "myTag2", Scenario = "myScenario")], you will receive an ambiguous step binding error if the myScenario has **both** the "myTag1" and "myTag2" tags.

2.20.2 Scope Examples

Scoped BeforeScenario Hook

The following example starts Selenium for scenarios marked with the @web tag.

```
[BeforeScenario("web")]  
public static void BeforeWebScenario()  
{  
    StartSelenium();  
}
```


Different Steps for Different Tags

The following example defines a different scope for the same step depending on whether UI automation ("web" tag) or controller automation ("controller" tag) is required:

```
[When(@"I perform a simple search on '(.*)'", Scope(Tag = "controller"))]
public void WhenIPerformASimpleSearchOn(string searchTerm)
{
    var controller = new CatalogController();
    ActionResult = controller.Search(searchTerm);
}

[When(@"I perform a simple search on '(.*)'", Scope(Tag = "web"))]
public void PerformSimpleSearch(string title)
{
    selenium.GoToThePage("Home");
    selenium.Type("searchTerm", title);
    selenium.Click("searchButton");
}
```

2.20.3 Scoping Tips & Tricks

The following example shows a way to “ignore” executing the scenarios marked with @manual. However SpecFlow's tracing will still display the steps, so you can work through the manual scenarios by following the steps in the report.

```
[Binding, Scope(Tag = "manual")]
public class ManualSteps
{
    [Given(".*"), When(".*"), Then(".*")]
    public void EmptyStep()
    {
    }

    [Given(".*"), When(".*"), Then(".*")]
    public void EmptyStep(string multiLineStringParam)
    {
    }

    [Given(".*"), When(".*"), Then(".*")]
    public void EmptyStep(Table tableParam)
    {
    }
}
```

2.20.4 Beyond Scope

You can define more complex filters using the `ScenarioContext` class. The following example starts selenium if the scenario is tagged with `@web` and `@automated`.

```
[Binding]
public class Binding
{
    ScenarioContext _scenarioContext;

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

    [BeforeScenario("web")]
    public static void BeforeWebScenario()
    {
        if(_scenarioContext.ScenarioInfo.Tags.Contains("automated"))
            StartSelenium();
    }
}
```

2.21 Sharing Data between Bindings

In many cases, different bindings need to exchange data during execution. SpecFlow provides several ways of sharing data between bindings.

2.21.1 Instance Fields

If the binding is an instance method, SpecFlow creates a new instance of the containing class for every scenario execution. Following the `entity-based step organization rule`, defining instance fields in the binding classes is an efficient way of sharing data between different steps of the same scenario that are related to the same entity.

The following example saves the result of the MVC action to an instance field in order to make assertions for it in a “then” step.

```
[Binding]
public class SearchSteps
{
    private ActionResult actionResult;

    [When(@"I perform a simple search on '(.*?)'")]
    public void WhenIPerformASimpleSearchOn(string searchTerm)
    {
        var controller = new CatalogController();
        actionResult = controller.Search(searchTerm);
    }

    [Then(@"the book list should exactly contain book '(.*?)'")]
    public void ThenTheBookListShouldExactlyContainBook(string title)
```

(continues on next page)

(continued from previous page)

```
{  
    var books = actionResult.Model<List<Book>>();  
    CustomAssert.Any(books, b => b.Title == title);  
}  
}
```

2.21.2 Context Injection

SpecFlow supports a very simple dependency framework that is able to instantiate and inject class instances for the scenarios. With this feature you can group the shared state to context-classes, and inject them into every binding class that is interested in that shared state.

See more about this feature in the [Context Injection](#) page.

The following example defines a context class to store referred books. The context class is injected to a binding class.

```
public class CatalogContext  
{  
    public CatalogContext()  
    {  
        ReferenceBooks = new ReferenceBookList();  
    }  
  
    public ReferenceBookList ReferenceBooks { get; set; }  
}  
  
[Binding]  
public class BookSteps  
{  
    private readonly CatalogContext _catalogContext;  
  
    public BookSteps(CatalogContext catalogContext)  
    {  
        _catalogContext = catalogContext;  
    }  
  
    [Given(@"the following books")]  
    public void GivenTheFollowingBooks(Table table)  
    {  
        foreach (var book in table.CreateSet<Book>())  
        {  
            SaveBook(book);  
            _catalogContext.ReferenceBooks.Add(book.Id, book);  
        }  
    }  
}
```

2.21.3 ScenarioContext and FeatureContext

SpecFlow provides two context instances.

The `ScenarioContext` is created for each individual scenario execution and it is disposed when the scenario execution has been finished.

The `FeatureContext` is created when the first scenario is executed from a feature and disposed when the execution of the feature's scenarios ends. In the rare case, when you need to preserve state in the context of a feature, the `FeatureContext.Current` instance can be used as a property bag.

2.21.4 Static Fields

Generally, using static fields can cause synchronization and maintenance issues and makes the unit testability hard. As the SpecFlow tests are executed synchronously and people usually don't write unit tests for the tests itself, these arguments are just partly valid for binding codes.

In some cases sharing a state through static fields can be an efficient solution.

2.22 Context Injection

SpecFlow supports a very simple dependency framework that is able to instantiate and inject class instances for scenarios. This feature allows you to group the shared state in context classes, and inject them into every binding class that needs access to that shared state.

To use context injection:

1. Create your POCOs (plain old CLR object), simple .NET classes, representing the shared data.
2. Define them as constructor parameters in every binding class that requires them.
3. Save the constructor argument to instance fields, so you can use them in the step definitions.

Rules:

- The life-time of these objects is limited to a scenario's execution.
- If the injected objects implement `IDisposable`, they will be disposed after the scenario is executed.
- The injection is resolved recursively, i.e. the injected class can also have dependencies.
- Resolution is done using public constructors only.
- If there are multiple public constructors, SpecFlow takes the first one.

The container used by SpecFlow can be customized, e.g. you can include object instances that have already been created, or modify the resolution rules. See the *Advanced options* section below for details.

2.22.1 Examples

In the first example we define a POCO for holding the data of a person and use it in a *given* and a *then* step that are placed in different binding classes.

```
public class PersonData // the POCO for sharing person data
{
    public string FirstName;
    public string LastName;
}

[Binding]
public class MyStepDefs
{
    private readonly PersonData personData;
    public MyStepDefs(PersonData personData) // use it as ctor parameter
    {
        this.personData = personData;
    }

    [Given]
    public void The_person_FIRSTNAME_LASTNAME(string firstName, string lastName)
    {
        personData.FirstName = firstName; // write into the shared data
        personData.LastName = lastName;
        //... do other things you need
    }
}

[Binding]
public class OtherStepDefs // another binding class needing the person
{
    private readonly PersonData personData;
    public OtherStepDefs(PersonData personData) // ctor parameter here too
    {
        this.personData = personData;
    }

    [Then]
    public void The_person_data_is_properly_displayed()
    {
        var displayedData = ... // get the displayed data from the app
        // read from shared data, to perform assertions
        Assert.AreEqual(personData.FirstName + " " + personData.LastName,
            displayedData, "Person name was not displayed properly");
    }
}
```

The following example defines a context class to store referred books. The context class is injected into a binding class.

```
public class CatalogContext
{
    public CatalogContext()
    {
```

(continues on next page)

(continued from previous page)

```

        ReferenceBooks = new ReferenceBookList();
    }

    public ReferenceBookList ReferenceBooks { get; set; }
}

[Binding]
public class BookSteps
{
    private readonly CatalogContext _catalogContext;

    public BookSteps(CatalogContext catalogContext)
    {
        _catalogContext = catalogContext;
    }

    [Given(@"the following books")]
    public void GivenTheFollowingBooks(Table table)
    {
        foreach (var book in table.CreateSet<Book>())
        {
            SaveBook(book);
            _catalogContext.ReferenceBooks.Add(book.Id, book);
        }
    }
}

```

2.22.2 Advanced options

The container used by SpecFlow can be customized, e.g. you can include object instances that have already been created, or modify the resolution rules.

You can customize the container from a *plugin* or a before scenario *hook*. The class customizing the injection rules has to obtain an instance of the scenario execution container (an instance of `BoDi.IObjectContainer`). This can be done through constructor injection (see example below).

The following example adds the Selenium web driver to the container, so that binding classes can specify `IWebDriver` dependencies (a constructor argument of type `IWebDriver`).

```

[Binding]
public class WebDriverSupport
{
    private readonly IObjectContainer objectContainer;

    public WebDriverSupport(IObjectContainer objectContainer)
    {
        this.objectContainer = objectContainer;
    }

    [BeforeScenario]
    public void InitializeWebDriver()
    {

```

(continues on next page)

(continued from previous page)

```

var webDriver = new FirefoxDriver();
objectContainer.RegisterInstanceAs<IWebDriver>(webDriver);
}
}

```

2.22.3 Custom Dependency Injection Frameworks

As mentioned above, the default SpecFlow container is `IObjectContainer` which is recommended for most scenarios. However, you may have situations where you need more control over the configuration of the dependency injection, or make use of an existing dependency injection configuration within the project you are testing, e.g. pulling in service layers for assisting with assertions in `Then` stages.

Consuming existing plugins

You can find the list of available plugins [here](#).

To make use of these plugins, you need to add a reference and add the plugin to your configuration in the `specflow` section:

```

<specFlow>
  <plugins>
    <add name="SpecFlow.Autofac" type="Runtime" />
  </plugins>
  <!-- Anything else -->
</specFlow>

```

This tells SpecFlow to load the runtime plugin and allows you to create an entry point to use this functionality, as shown in the [autofac example](#). Once set up, your dependencies are injected into steps and bindings like they are with the `IObjectContainer`, but behind the scenes it will be pulling those dependencies from the DI container you added.

One thing to note here is that each plugin has its own conventions for loading the entry point. This is often a static class with a static method containing an attribute that is marked by the specific plugin. You should check the requirements of the plugins you are using.

You can load all your dependencies within this handler section, or you can to inject the relevant IoC container into your binding sections like this:

```

[Binding]
public class WebDriverPageHooks
{
    private readonly IKernel _kernel;

    // Inject in our container (using Ninject here)
    public WebDriverPageHooks(IKernel kernel)
    { _kernel = kernel; }

    private IWebDriver SetupWebDriver()
    {
        var options = new ChromeOptions();
        options.AddArgument("--start-maximized");
        options.AddArgument("--disable-notifications");
        return new ChromeDriver(options);
    }
}

```

(continues on next page)

(continued from previous page)

```
}

[BeforeScenario]
public void BeforeScenario()
{
    var webdriver = SetupWebDriver();
    _kernel.Bind<IWebDriver>().ToConstant(webdriver);
}

[AfterScenario]
public void AfterScenario()
{
    var webDriver = _kernel.Get<IWebDriver>();

    // Output any screenshots or log dumps etc

    webDriver.Close();
    webDriver.Dispose();
}
}
```

This gives you the option of either loading types up front or creating types within your binding sections so you can dispose of them as necessary.

Creating your own

We recommend looking at the [autofac example](#) and [plugins documentation](#) and following these conventions.

Remember to adhere to the plugin documentation and have your assembly end in `.SpecFlowPlugin` e.g. `SpecFlow.AutoFac.SpecFlowPlugin`. Internal namespaces can be anything you want, but the assembly name must follow this naming convention or SpecFlow will be unable to locate it.

2.23 ScenarioContext

You may have at least seen the `ScenarioContext` from the code that SpecFlow generates when a missing step definition is found: `ScenarioContext.Pending()`;

`ScenarioContext` provides access to several functions, which are demonstrated using the following scenarios.

2.23.1 Accessing the ScenarioContext

In Bindings

To access the `ScenarioContext` you have to get it via [context injection](#).

Example:

```
[Binding]
public class Binding
{
```

(continues on next page)

(continued from previous page)

```

private ScenarioContext _scenarioContext;

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

```

Now you can access the ScenarioContext in all your Bindings with the `_scenarioContext` field.

In Hooks

Before/AfterTestRun

Accessing the ScenarioContext is not possible, as no Scenario is executed when the hook is called.

Before/AfterFeature

Accessing the ScenarioContext is not possible, as no Scenario is executed when the hook is called.

Before/AfterScenario

Accessing the ScenarioContext is done like in *normal bindings*

Before/AfterStep

Accessing the ScenarioContext is done like in *normal bindings*

Migrating from ScenarioContext.Current

With SpecFlow 3.0, we marked ScenarioContext.Current obsolete, to make clear that you should avoid using these properties in future. The reason for moving away from these properties is that they do not work when running scenarios in parallel.

So how do you now access ScenarioContext?

Before SpecFlow 3.0 this was common:

```

[Binding]
public class Bindings
{
    [Given(@"I have entered (.*) into the calculator")]
    public void GivenIHaveEnteredIntoTheCalculator(int number)
    {
        ScenarioContext.Current["Number1"] = number;
    }

    [BeforeScenario()]

```

(continues on next page)

(continued from previous page)

```
public void BeforeScenario()
{
    Console.WriteLine("Starting " + ScenarioContext.Current.ScenarioInfo.Title);
}
}
```

As of SpecFlow 3.0, you now need to use [Context-Injection](#) to acquire an instance of `ScenarioContext` by requesting it via the constructor.

```
public class Bindings
{
    private readonly ScenarioContext _scenarioContext;

    public Bindings(ScenarioContext scenarioContext)
    {
        _scenarioContext = scenarioContext;
    }
}
```

Once you have acquired the instance of `ScenarioContext`, you can use it with the same methods and properties as before. So our example will now look like this:

```
public class Bindings
{
    private readonly ScenarioContext _scenarioContext;

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

    [Given(@"I have entered (.*) into the calculator")]
    public void GivenIHaveEnteredIntoTheCalculator(int number)
    {
        _scenarioContext["Number1"] = number;
    }

    [BeforeScenario()]
    public void BeforeScenario()
    {
        Console.WriteLine("Starting " + _scenarioContext.ScenarioInfo.Title);
    }
}
```

2.23.2 ScenarioContext.Pending

See *Mark Steps as not implemented*

2.23.3 Storing data in the ScenarioContext

ScenarioContext helps you store values in a dictionary between steps. This helps you to organize your step definitions better than using private variables in step definition classes.

There are some type-safe extension methods that help you to manage the contents of the dictionary in a safer way. To do so, you need to include the namespace TechTalk.SpecFlow.Assist, since these methods are extension methods of ScenarioContext.

2.23.4 ScenarioContext.ScenarioInfo

ScenarioContext.ScenarioInfo allows you to access information about the scenario currently being executed, such as its title and scenario and feature tags:

In the .feature file:

```
@showUpInScenarioInfo @andThisToo

Scenario: Showing information of the scenario

When I execute any scenario
Then the ScenarioInfo contains the following information
  | Field | Value |
  | Tags  | showUpInScenarioInfo, andThisToo |
  | Title | Showing information of the scenario |
```

and in the step definition:

```
private class ScenarioInformation
{
    public string Title { get; set; }
    public string[] Tags { get; set; }
}

[When(@"I execute any scenario")]
public void ExecuteAnyScenario(){}

[Then(@"the ScenarioInfo contains the following information")]
public void ScenarioInfoContainsInterestingInformation(Table table)
{
    // Create our small DTO for the info from the step
    var fromStep = table.CreateInstance<ScenarioInformation>();
    fromStep.Tags = table.Rows[0]["Value"].Split(',');

    // Short-hand to the scenarioInfo
    var si = _scenarioContext.ScenarioInfo;

    // Assertions
    si.Title.Should().Equal(fromStep.Title);
```

(continues on next page)

(continued from previous page)

```
for (var i = 0; i < si.Tags.Length - 1; i++)
{
    si.Tags[i].Should().Equal(fromStep.Tags[i]);
}
```

ScenarioContext.ScenarioInfo also provides access to the current set of arguments from the scenario's examples in the form of an IDictionary:

Scenario: Accessing the current example

When I use examples in my scenario
Then the examples are available in ScenarioInfo

Examples:

Sport	TeamSize	
Soccer	11	
Basketball	6	

```
public class ScenarioExamplesDemo
{
    private ScenarioInfo _scenarioInfo;

    public ScenarioExamplesDemo(ScenarioInfo scenarioInfo)
    {
        _scenarioInfo = scenarioInfo;
    }

    [When(@"I use examples in my scenario")]
    public void IUseExamplesInMyScenario() {}

    [Then(@"the examples are available in ScenarioInfo")]
    public void TheExamplesAreAvailableInScenarioInfo()
    {
        var currentArguments = _scenarioInfo.Arguments;
        var currentSport = currentArguments["Sport"];
        var currentTeamSize = currentArguments["TeamSize"];
        Console.WriteLine($"The current sport is {currentSport}");
        Console.WriteLine($"The current sport allows teams of {currentTeamSize} players
↪");
    }
}
```

Another use is to check if an error has occurred, which is possible with the ScenarioContext.TestError property, which simply returns the exception.

You can use this information for “error handling”. Here is an uninteresting example:

in the .feature file:

```
#This is not so easy to write a scenario for but I've created an AfterScenario-hook
@showingErrorHandling
Scenario: Display error information in AfterScenario
```

(continues on next page)

(continued from previous page)

When an error occurs in a step

and the step definition:

```
[When("an error occurs in a step")]
public void AnErrorOccurs()
{
    "not correct".Should().Equal("correct");
}

[AfterScenario("showingErrorHandling")]
public void AfterScenarioHook()
{
    if(_scenarioContext.TestError != null)
    {
        var error = _scenarioContext.TestError;
        Console.WriteLine("An error occurred:" + error.Message);
        Console.WriteLine("It was of type:" + error.GetType().Name);
    }
}
```

This is another example, that might be more useful:

```
[AfterScenario]
public void AfterScenario()
{
    if(_scenarioContext.TestError != null)
    {
        WebBrowser.Driver.CaptureScreenShot(_scenarioContext.ScenarioInfo.Title);
    }
}
```

In this case, MvcContrib is used to capture a screenshot of the failing test and name the screenshot after the title of the scenario.

2.23.5 ScenarioContext.CurrentScenarioBlock

Use `ScenarioContext.CurrentScenarioBlock` to query the “type” of step (Given, When or Then). This can be used to execute additional setup/cleanup code right before or after Given, When or Then blocks.

in the .feature file:

```
Scenario: Show the type of step we're currently on
    Given I have a Given step
    And I have another Given step
    When I have a When step
    Then I have a Then step
```

and the step definition:

```
[Given("I have a (.*) step")]
[Given("I have another (.*) step")]
```

(continues on next page)

(continued from previous page)

```
[When("I have a (.*) step")]
[Then("I have a (.*) step")]
public void ReportStepTypeName(string expectedStepType)
{
    var stepType = _scenarioContext.CurrentScenarioBlock.ToString();
    stepType.Should().Equal(expectedStepType);
}
```

2.23.6 ScenarioContext.StepContext

Sometimes you need to access the currently executed step, e.g. to improve tracing. Use the `_scenarioContext.StepContext` property for this purpose.

2.24 FeatureContext

SpecFlow provides access to the current test context using both `FeatureContext` and the more commonly used `ScenarioContext`. `FeatureContext` persists for the duration of the execution of an entire feature, whereas `ScenarioContext` only persists for the duration of a scenario.

2.24.1 Accessing the FeatureContext

in Bindings

To access the `FeatureContext` you have to get it via [Context-Injection](#).

Example:

```
[Binding]
public class Binding
{
    private FeatureContext _featureContext;

    public Binding(FeatureContext featureContext)
    {
        _featureContext = featureContext;
    }
}
```

Now you can access the `FeatureContext` in all your Bindings with the `_featureContext` field.

in Hooks

Before/AfterTestRun

Accessing the `FeatureContext` is not possible, as no `Feature` is executed, when the hook is called.

Before/AfterFeature

You can get the `FeatureContext` via parameter of the static method.

Example:

```
[Binding]
public class Hooks
{
    [BeforeFeature]
    public static void BeforeFeature(FeatureContext featureContext)
    {
        Console.WriteLine("Starting " + featureContext.FeatureInfo.Title);
    }

    [AfterFeature]
    public static void AfterFeature(FeatureContext featureContext)
    {
        Console.WriteLine("Finished " + featureContext.FeatureInfo.Title);
    }
}
```

Before/AfterScenario

Accessing the `FeatureContext` is done like in *normal bindings*

Before/AfterStep

Accessing the `FeatureContext` is done like in *normal bindings*

2.24.2 Storing data in the FeatureContext

`FeatureContext` implements `Dictionary<string, object>`. So you can use the `FeatureContext` like a property bag.

2.24.3 Migrating from FeatureContext.Current

With SpecFlow 3.0, we marked `FeatureContext.Current` as obsolete, to make clear that you should avoid using these properties in future. The reason for moving away from these properties is that they do not work when running scenarios in parallel.

So how do you now access `FeatureContext`?

You can acquire the `FeatureContext` via [Context-Injection](#). However, if you want to use it together with Before/After Feature hooks, you need to acquire it via a function parameter.

Previously:

```
public class Hooks
{
    [AfterFeature()]
    public static void AfterFeature()
    {
        Console.WriteLine("Finished " + FeatureContext.Current.FeatureInfo.Title);
    }
}
```

Now:

```
public class Hooks
{
    [AfterFeature]
    public static void AfterFeature(FeatureContext featureContext)
    {
        Console.WriteLine("Finished " + featureContext.FeatureInfo.Title);
    }
}
```

2.24.4 FeatureContext.FeatureInfo

`FeatureInfo` provides more information than `ScenarioInfo`, but it works the same:

In the .feature file:

```
Scenario: Showing information of the feature

When I execute any scenario in the feature
Then the FeatureInfo contains the following information
  | Field          | Value                                     |
  | Tags           | showUpInScenarioInfo, andThisToo       |
  | Title          | FeatureContext features                 |
  | TargetLanguage | CSharp                                  |
  | Language       | en-US                                   |
  | Description    | In order to                             |
```

and in the step definition:

```
private class FeatureInformation
{
    public string Title { get; set; }
}
```

(continues on next page)

(continued from previous page)

```

public GenerationTargetLanguage TargetLanguage { get; set; }
public string Description { get; set; }
public string Language { get; set; }
public string[] Tags { get; set; }
}

[When(@"I execute any scenario in the feature")]
public void ExecuteAnyScenario() { }

[Then(@"the FeatureInfo contains the following information")]
public void FeatureInfoContainsInterestingInformation(Table table)
{
    // Create our small DTO for the info from the step
    var fromStep = table.CreateInstance<FeatureInformation>();
    fromStep.Tags = table.Rows[0]["Value"].Split(',');

    var fi = _featureContext.FeatureInfo;

    // Assertions
    fi.Title.Should().Equal(fromStep.Title);
    fi.GenerationTargetLanguage.Should().Equal(fromStep.TargetLanguage);
    fi.Description.Should().StartWith(fromStep.Description);
    fi.Language.IetfLanguageTag.Should().Equal(fromStep.Language);
    for (var i = 0; i < fi.Tags.Length - 1; i++)
    {
        fi.Tags[i].Should().Equal(fromStep.Tags[i]);
    }
}

```

FeatureContext exposes a Binding Culture property that simply points to the culture the feature is written in (en-US in our example).

2.25 Calling Steps from Step Definitions

Note: This feature will be deprecated with SpecFlow 3.1 and removed in a future version (probably 4.0). Details about it can be found [here](#).

It is possible to call steps from within Step Definitions:

```

[Binding]
public class CallingStepsFromStepDefinitionSteps : Steps
{
    [Given(@"the user (.*) exists")]
    public void GivenTheUserExists(string name)
    {
        // ...
    }

    [Given(@"I log in as (.*)")]
    public void GivenILogInAs(string name)
    {

```

(continues on next page)

(continued from previous page)

```
// ...  
}  
  
[Given(@"(.*) is logged in")]  
public void GivenIsLoggedIn(string name)  
{  
    Given(string.Format("the user {0} exists", name));  
    Given(string.Format("I log in as {0}", name));  
}  
}
```

Invoking steps from step definitions is practical if you have several common steps that you want to perform in several scenarios, or simply if you want to make your scenarios shorter and more declarative. This allows you to do the following in a Scenario:

Instead of having a lot of repetition:

Note: When using this approach to remove duplications from your feature files, the console output will contain both the master step and the delegated steps as follows:

2.25.1 Calling steps with multiline step arguments

Sometimes you want to call a step that has been designed to take [Multiline Step Arguments](../Gherkin/Using Gherkin Language in SpecFlow.md), for example:

Tables

```
[Given(@"an expense report for (.*) with the following posts:")]  
public void GivenAnExpenseReportForWithTheFollowingPosts(string date, Table postTable)  
{  
    // ...  
}
```

This can easily be called from a plain text step like this:

But what if you want to call this from a step definition? There are a couple of ways to do this:

```
[Given(@"A simple expense report")]  
public void GivenASimpleExpenseReport()  
{  
    string[] header = {"account" , "description", "amount"};  
    string[] row1 = {"INT-100" , "Taxi", "114"};  
    string[] row2 = {"CUC-101" , "Peeler", "22"};  
    var t = new Table(header);  
    t.AddRow(row1);  
    t.AddRow(row2);  
    Given("an expense report for Jan 2009 with the following posts:", t);  
}
```

2.26 Step Argument Conversions

Step bindings can use parameters to make them reusable for similar steps. The parameters are taken from either the step's text or from the values in additional examples. These arguments are provided as either strings or `TechTalk.SpecFlow.Table` instances.

To avoid cumbersome conversions in the step binding methods, SpecFlow can perform an automatic conversion from the arguments to the parameter type in the binding method. All conversions are performed using the culture of the feature file, unless the *bindingCulture element* is defined in your `app.config` file (see *Feature Language*). The following conversions can be performed by SpecFlow (in the following precedence):

- no conversion, if the argument is an instance of the parameter type (e.g. the parameter type is `object` or `string`)
- step argument transformation
- standard conversion

2.26.1 Step Argument Transformation

Step argument transformations can be used to apply a custom conversion step to the arguments in step definitions. The step argument transformation is a method that converts from text (specified by a regular expression) or a `Table` instance to an arbitrary .NET type.

A step argument transformation is used to convert an argument if:

- The return type of the transformation is the same as the parameter type
- The regular expression (if specified) matches the original (string) argument

Note: If multiple matching transformation are available, a warning is output in the trace and the first transformation is used.

The following example transforms a relative period of time (`in 3 days`) into a `DateTime` structure.

```
[Binding]
public class Transforms
{
    [StepArgumentTransformation(@"in (\d+) days?")]
    public DateTime InXDaysTransform(int days)
    {
        return DateTime.Today.AddDays(days);
    }
}
```

The following example transforms any string input (no regex provided) into an `XmlDocument`.

```
[Binding]
public class Transforms
{
    [StepArgumentTransformation]
    public XmlDocument XmlTransform(string xml)
    {
        XmlDocument result = new XmlDocument();
        result.LoadXml(xml);
        return result;
    }
}
```

The following example transforms a table argument into a list of `Book` entities (using the [SpecFlow Assist Helpers](#)).

```
[Binding]
public class Transforms
{
    [StepArgumentTransformation]
    public IEnumerable<Book> BooksTransform(Table booksTable)
    {
        return booksTable.CreateSet<Books>();
    }
}
```

```
[Binding]
public class Transforms
{
    [Given(@"Show messenger""(.*)""")]
    public void GiveShowMessenger()
    {
        string chave = nfe.Tab();
        Assert.IsNotNull(chave);
    }
}
```

2.26.2 Standard Conversion

A standard conversion is performed by SpecFlow in the following cases:

- The argument can be converted to the parameter type using `Convert.ChangeType()`
- The parameter type is an `enum` type and the (string) argument is an enum value
- The parameter type is `Guid` and the argument contains a full GUID string or a GUID string prefix. In the latter case, the value is filled with trailing zeroes.

2.27 Bindings from External Assemblies

[Bindings](#) can be defined in the main SpecFlow project or in other assemblies (*external binding assemblies*). If the bindings are used from external binding assemblies, the following notes have to be considered:

- The external binding assembly can be another project in the solution or a compiled library (dll).
- The external binding assembly can also use a different .NET language, e.g. you can write bindings for your C# SpecFlow project also in F# (As an extreme case, you can use your SpecFlow project with the feature files only and with all the bindings defined in external binding assemblies).
- The external binding assembly has to be referenced from the SpecFlow project to ensure it is copied to the target folder and listed in the `specflow.json` or `app.config` of the SpecFlow project (see below).
- The external binding assemblies can contain all kind of bindings: [step definition](#), [hooks](#) and also [step argument transformations](#).
- The bindings from assembly references are not fully supported in the Visual Studio integration of SpecFlow v1.8 or earlier: the step definitions from these assemblies will not be listed in the autocompletion lists.

- The external binding file must be in the root of the project being referenced. If it is in a folder in the project, the bindings will not be found.

2.27.1 Configuration

In order to use bindings from an external binding assembly, you have to list it (with the assembly name) in the `specflow.json` or `app.config` of the SpecFlow project. The SpecFlow project is always included implicitly. See more details on the configuration in the `<stepAssemblies>` section of [the configuration guide](#).

specflow.json example:

```
{
  "stepAssemblies": [
    {
      "assembly": "MySharedBindings"
    }
  ]
}
```

app.config example:

```
<specFlow>
  <stepAssemblies>
    <stepAssembly assembly="MySharedBindings" />
  </stepAssemblies>
</specFlow>
```

2.28 SpecFlow.Assist Helpers

A number of helpers implemented as extension methods of the `Table` class make it easier to implement steps that accept a `Table` parameter. To use these helpers, you need to add the `TechTalk.SpecFlow.Assist` namespace to the top of your file:

```
using TechTalk.SpecFlow.Assist;
```

When helper methods expect a generic type (usually denoted as `<T>` in the method signature), you can use:

- classes
- records (with C# 9)
- tuples

2.28.1 CreateInstance

The `CreateInstance<T>` extension method of the `Table` class will convert a table in your scenario to a single instance of a class. The class used to convert the table is specified by the generic type `T` in the method signature `CreateInstance<T>`. You can use two different table layouts in your scenarios with the `CreateInstance<T>` method.

- **Vertical Tables**

A vertical table consists of two columns where values in the first column match property names, and values of the second column are the values assigned to those properties. The header row of the table is ignored. Header cells may be named to suit your use case.

Given I entered the following data into the new account form:

Field	Value
Name	John Galt
Birthdate	2/2/1902
Height In Inches	72
Bank Account Balance	1234.56

^^^^^^^^^^^^^^^^^^^^^^^^^ ^^^^^^^^^^^^^^^^^^^^^^^^^

property names *property values*

This layout is desirable for tables containing many values making the vertical layout easier to read.

- **Horizontal Tables**

A horizontal table consists of a header row where the header cells match property names, and subsequent data rows contain the values assigned to those properties. In order to convert a horizontal table to a single instance of a class, the table must only contain the header row and one data row.

Given I entered the following data into the new account form:

Name	Birthdate	Height In Inches	Bank Account Balance	# Header
John Galt	2/2/1902	72	1234.56	# Data row

↪row (property names)

↪(property values)

This layout is desirable when the table does not require too many values. This helps save vertical space by consuming more horizontal space in your feature file.

Important: use the `CreateSet<T>` method described below to create a collection of objects if more than one data row is needed.

Deciding to use a vertical or horizontal table layout is subjective. Choose the layout that is easiest to read given the information in the table.

SpecFlow matches table values to property names regardless of letter case. To SpecFlow, “BankAccount”, “Bank Account”, “BANK ACCOUNT” and “bank account” will all map to a property named `BankAccount`. More information on column naming is below.

Using `CreateInstance` with a Class

You can map a table to a custom class you write. The following example will map the tables described above in the vertical or horizontal layouts. First, create the class:

```
// Class used to map table
class Account
{
    public string Name { get; set; }
    public int HeightInInches { get; set; }
    public decimal BankAccountBalance { get; set; }
}
```

Remember that property names should match values in the table in your scenario. SpecFlow requires properties to have both a public getter and a public setter. Most built-in .NET types are converted automatically. This includes the following types:

- int and int?
- decimal and decimal?
- bool and bool?
- DateTime and DateTime?

Plus **many more**.

The name of the class is put in place of the generic type T in the call to `CreateInstance<T>`. An example step definition is below.

```
[Given(@"Given I entered the following data into the new account form:")]
public void GivenIEnteredTheFollowingDataIntoTheNewAccountForm(Table table)
{
    var account = table.CreateInstance<Account>();
    //
    // account.Name is "John Galt"
    // account.HeightInInches is 72
    // account.BankAccountBalance is 1234.56
}
```

The `CreateInstance<T>` method will create the `Account` object and set properties according to what can be read from the table. Table cell values are strings by default. These strings are converted to the type specified for each property of the destination class. For example, the string "1234.56" in the table is converted to a decimal value before being assigned to the `BankAccountBalance` property.

Using CreateInstance with ValueTuple

Alternatively you can use `ValueTuples` and destructuring:

```
[Given(@"Given I entered the following data into the new account form:")]
public void GivenIEnteredTheFollowingDataIntoTheNewAccountForm(Table table)
{
    var account = table.CreateInstance<(string name, DateTime birthDate, int_
↪heightInInches, decimal bankAccountBalance)>();
    // account.name is "John Galt"
    // account.heightInInches is 72
    // account.bankAccountBalance is 1234.56
}
```

Important: In the case of tuples, *you need to have the same number of parameters and types; parameter names do not matter*, as `ValueTuples` do not hold parameter names at runtime using reflection.

Scenarios with more than 7 properties are not currently supported when converting to `ValueTuple`, and you will receive an exception if you try to map more than 7 properties.

The next section describes how to convert a horizontal table with more than one data row to a collection of objects.

2.28.2 CreateSet

The `CreateSet<T>` extension method of the `Table` class converts the table into an enumerable set of objects. For example, assume you have the following step:

```
Given these products exist
| Sku   | Name           | Price |
| BOOK1 | Atlas Shrugged | 25.04 |
| BOOK2 | The Fountainhead | 20.15 |
```

And you want to map rows in that table to the following class:

```
public class Product
{
    public string Sku { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
}
```

You can convert the table to a collection of `Product` objects in your step definition using `CreateSet<Product>()`:

```
[Given(@"Given these products exist")]
public void GivenTheseProductsExist(Table table)
{
    var products = table.CreateSet<Product>();
    // ...
}
```

The `CreateSet<T>` method returns an `IEnumerable<T>` based on the matching data in the table. It contains the values for each object, making appropriate type conversions from string to the related property. Column headers are matched to property names in the same way as `CreateInstance<T>`.

2.28.3 CompareToInstance

The `CompareToInstance<T>` extension method of the `Table` class makes it easy to compare the properties of an object to the table in your scenario. For example, you have a class like this:

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int YearsOld { get; set; }
}
```

You want to compare it to a table in a step like this:

```
Then the person should have the following values
| Field      | Value |
| First Name | John  |
| Last Name  | Galt  |
| Years Old  | 54    |
```

You can assert that the properties match with this simple step definition:


```
[Binding]
public class PersonSteps
{
    ScenarioContext _scenarioContext;

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

    [Then("the person should have the following values")]
    public void ThenThePersonShouldHaveTheFollowingValues(Table table){
        // you don't have to get person this way, this is just for demonstration purposes
        var person = _scenarioContext.Get<Person>();

        table.CompareToInstance<Person>(person);
    }
}
```

If FirstName is not “John”, LastName is not “Galt”, or YearsOld is not 54, a descriptive error showing the differences is thrown.

If the values match, no exception is thrown, and SpecFlow continues to process your scenario.

2.28.4 CompareToSet

The CompareToSet<T> extension method of the Table class works similarly to CompareToInstance<T>, except it compares a collection of objects. For example, you have a class like this:

```
public class Account
{
    public string Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string MiddleName { get; set; }
}
```

You want to test that your system returns a specific set of accounts, like this:

```
Then I get back the following accounts
| Id | First Name | Last Name |
| 1  | John      | Galt      |
| 2  | Howard    | Roark     |
```

You can test your results with one call to CompareToSet:

```
[Binding]
public class AccountSteps
{
    ScenarioContext _scenarioContext;

    public AccountSteps(ScenarioContext scenarioContext)
    {
```

(continues on next page)

(continued from previous page)

```

        _scenarioContext = scenarioContext;
    }

    [Then("I get back the following accounts")]
    public void ThenIGetBackTheFollowingAccounts(Table table)
    {
        // (or get the accounts from the database or web service)
        var accounts = _scenarioContext.Get<IEnumerable<Account>>();

        table.CompareToSet<Account>(accounts);
    }
}

```

In this example, `CompareToSet<T>` checks that two accounts are returned, and only tests the properties you defined in the table. **It does not test the order of the objects, only that one was found that matches.** If no record matching the properties in your table is found, an exception is thrown that includes the row number(s) that do not match up.

Comparing Sets When Order Matters

In use cases where the order should match, pass `true` as the second argument to `CompareToSet`:

```

table.CompareToSet<Account>(accounts, true);
//

```

In addition to throwing an exception if property values do not match, SpecFlow will throw an exception if the order of the accounts doesn't match your expectations. This is useful when the order of things is determined by business rules, or in use cases like search results.

2.28.5 Column naming

The SpecFlow Assist helpers use the values in your table to determine what properties to set in your object. However, the names of the columns do not need to match exactly - whitespace and casing is ignored. For example, the following two tables are treated as identical:

```
| FirstName | LastName | DateOfBirth | HappinessRating |
```

```
| First name | Last name | Date of birth | HAPPINESS rating |
```

This allows you to make your tables more readable to others.

2.28.6 Aliasing

Note: Available with version 2.3 and later

If you have properties in your objects that are known by different terms within the business domain, these can be Aliased in your model by applying the attribute `TableAliases`. This attribute takes a collection of aliases as regular expressions that can be used to refer to the property in question.

For example, if you have an object representing an Employee, you might want to alias the Surname property:

```
public class Employee
{
    public string FirstName { get; set; }
    public string MiddleName { get; set; }

    [TableAliases("Last[]?Name", "Family[]?Name")]
    public string Surname { get; set; }
}
```

Test writers can then refer to this property as “Surname”, “Last Name”, “Lastname”, “Family Name” or “FamilyName”, and it will still be mapped to the correct column in your scenario.

The TableAliases attribute can be applied to a field, a property as a single attribute with multiple regular expressions, or as multiple attributes, depending on your preference.

2.28.7 Extensions

Out-of-the-box, the SpecFlow table helpers knows how to handle most C# base types. Types like String, Bool, Enum, Int, Decimal, DateTime, etc. are all covered (see [full list of supported times](#)). If you want to cover more types, including your own custom types, you can do so by registering your own instances of IValueRetriever and IValueComparer.

For example, you have a complex object like this:

```
public class Shirt
{
    public string Name { get; set; }
    public Color Color { get; set; }
}
```

You have a table like this:

Name	Color
XL	Blue
L	Red

If you want to map Blue and Red to the appropriate instance of the Color class, you need to create an instance of IValueRetriever that can convert the strings to the Color instance.

You can register your custom IValueRetriever (and/or an instance of IValueComparer if you want to compare colors) like this:

```
[Binding]
public static class Hooks1
{
    [BeforeTestRun]
    public static void BeforeTestRun()
    {
        Service.Instance.ValueRetrievers.Register(new ColorValueRetriever());
        Service.Instance.ValueComparers.Register(new ColorValueComparer());
    }
}
```

Examples on implementing these interfaces can be found as follows:

- [IValueRetriever](#)
- [IValueComparer](#)

Configuration

Some built in classes support configuration to adjust the default behaviour.

- [DateTimeValueRetriever](#) and [DateTimeOffsetValueRetriever](#) have a static `DateTimeStyles` property to adjust the style used to parse date times.

Example of usage:

```
[Binding]
public static class Hooks1
{
    [BeforeTestRun]
    public static void BeforeTestRun()
    {
        DateTimeValueRetriever.DateTimeStyles = DateTimeStyles.AdjustToUniversal |
        ↪ DateTimeStyles.AssumeUniversal;
    }
}
```

NullValueRetriever (from SpecFlow 3)

By default, non-specified (empty string) values are considered:

- An empty string for `String` and `System.Uri` values
- A null value for `Nullable<>` primitive types
- An error for non-nullable primitive types

To specify null values explicitly, add a `NullValueRetriever` to the set of registered retrievers, specifying the text to be treated as a null value, e.g.:

```
[Binding]
public static class Hooks1
{
    [BeforeTestRun]
    public static void BeforeTestRun()
    {
        Service.Instance.ValueRetrievers.Register(new NullValueRetriever("<null>"));
    }
}
```

Note: The comparison is case-insensitive.

2.28.8 Using ToProjection, ToProjectionOfSet and ToProjectionOfInstance extension methods for LINQ-based instance and set comparison

SpecFlow.Assist.CompareToSet Table extension method only checks for equivalence of collections which is a reasonable default. **SpecFlow.Assist** namespace also contains extension methods for With based operations.

Consider the following steps:

```
Scenario: Matching music collections
  When I have a music collection
    | Artist | Album |
    | Beatles | Rubber Soul |
    | Pink Floyd | Animals |
    | Muse | Absolution |
  Then it should match
    | Artist | Album |
    | Beatles | Rubber Soul |
    | Pink Floyd | Animals |
    | Muse | Absolution |
  And it should match
    | Artist | Album |
    | Beatles | Rubber Soul |
    | Muse | Absolution |
    | Pink Floyd | Animals |
  And it should exactly match
    | Artist | Album |
    | Beatles | Rubber Soul |
    | Pink Floyd | Animals |
    | Muse | Absolution |
  But it should not match
    | Artist | Album |
    | Beatles | Rubber Soul |
    | Queen | Jazz |
    | Muse | Absolution |
  And it should not match
    | Artist | Album |
    | Beatles | Rubber Soul |
    | Muse | Absolution |
  And it should not exactly match
    | Artist | Album |
    | Beatles | Rubber Soul |
    | Muse | Absolution |
    | Pink Floyd | Animals |
```

With LINQ-based operations each of the above comparisons can be expressed using a single line of code:

```
[Binding]
public class MusicCollectionSteps
{
    ScenarioContext _scenarioContext;

    public MusicCollectionSteps(ScenarioContext scenarioContext)
    {
        _scenarioContext = scenarioContext;
    }
}
```

(continues on next page)

(continued from previous page)

```

    }

    [When(@"I have a music collection")]
    public void WhenIHaveAMusicCollection(Table table)
    {
        var collection = table.CreateSet<Item>();

        _scenarioContext.Add("Collection", collection);
    }

    [Then(@"it should match")]
    public void ThenItShouldMatch(Table table)
    {
        var collection = _scenarioContext["Collection"] as IEnumerable<Item>;

        Assert.IsTrue(table.RowCount == collection.Count() && table.ToProjection<Item>().
↪ Except(collection.ToProjection()).Count() == 0);
    }

    [Then(@"it should exactly match")]
    public void ThenItShouldExactlyMatch(Table table)
    {
        var collection = _scenarioContext["Collection"] as IEnumerable<Item>;

        Assert.IsTrue(table.ToProjection<Item>().SequenceEqual(collection.
↪ ToProjection()));
    }

    [Then(@"it should not match")]
    public void ThenItShouldNotMatch(Table table)
    {
        var collection = _scenarioContext["Collection"] as IEnumerable<Item>;

        Assert.IsFalse(table.RowCount == collection.Count() && table.ToProjection<Item>
↪ ().Except(collection.ToProjection()).Count() == 0);
    }

    [Then(@"it should not exactly match")]
    public void ThenItShouldNotExactlyMatch(Table table)
    {
        var collection = _scenarioContext["Collection"] as IEnumerable<Item>;

        Assert.IsFalse(table.ToProjection<Item>().SequenceEqual(collection.
↪ ToProjection()));
    }
}

```

In a similar way we can implement containment validation:

```

Scenario: Containment
    When I have a music collection
        | Artist      | Album      |

```

(continues on next page)

(continued from previous page)

	Beatles		Rubber Soul	
	Pink Floyd		Animals	
	Muse		Absolution	

Then it should contain all items

	Artist		Album	
	Beatles		Rubber Soul	
	Muse		Absolution	

But it should not contain all items

	Artist		Album	
	Beatles		Rubber Soul	
	Muse		Resistance	

And it should not contain any of items

	Artist		Album	
	Beatles		Abbey Road	
	Muse		Resistance	

```
[Binding]
public class MusicCollectionSteps
{
    ScenarioContext _scenarioContext;

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

    [Then(@"it should contain all items")]
    public void ThenItShouldContainAllItems(Table table)
    {
        var collection = _scenarioContext["Collection"] as IEnumerable<Item>;

        Assert.IsTrue(table.ToProjection<Item>().Except(collection.ToProjection()).
↪Count() == 0);
    }

    [Then(@"it should not contain all items")]
    public void ThenItShouldNotContainAllItems(Table table)
    {
        var collection = _scenarioContext["Collection"] as IEnumerable<Item>;

        Assert.IsFalse(table.ToProjection<Item>().Except(collection.ToProjection()).
↪Count() == 0);
    }

    [Then(@"it should not contain any of items")]
    public void ThenItShouldNotContainAnyOfItems(Table table)
    {
        var collection = _scenarioContext["Collection"] as IEnumerable<Item>;

        Assert.IsTrue(table.ToProjection<Item>().Except(collection.ToProjection()).
↪Count() == table.RowCount);
    }
}
```

(continues on next page)

(continued from previous page)

```
}
```

What if Artist and Album are properties of different entities? Look at this piece of code:

```
var collection = from artist in ctx.Artists
                  where artist.Name == "Muse"
                  join album in ctx.Albums
                      on album.ArtistId equals artist.ArtistId
                  select new
                  {
                      Artist = artist.Name,
                      Album = album.Name
                  };
```

SpecFlow.Assist has a generic class named `EnumerableProjection<T>`. If a type `T` is known at compile time, the `ToProjection` method converts a table or a collection into an instance of `EnumerableProjection`:

```
table.ToProjection<Item>();
```

But if we need to compare a table with the collection of anonymous types from the example above, we need to express this type in some way so `ToProjection` will be able to build an instance of specialized `EnumerableProjection`. This is done by sending a collection as an argument to **ToProjection**. And to support both sets and instances and avoid naming ambiguity, corresponding methods are called **ToProjectionOfSet** and **ToProjectionOfInstance**:

```
table.ToProjectionOfSet(collection);
table.ToProjectionOfInstance(instance);
```

Here are the definitions of SpecFlow Table extensions methods that convert tables and collections of `IEnumerables` to `EnumerableProjection`:

```
public static IEnumerable<Projection<T>> ToProjection<T>(this IEnumerable<T> collection,
↪ Table table = null)
{
    return new EnumerableProjection<T>(table, collection);
}

public static IEnumerable<Projection<T>> ToProjection<T>(this Table table)
{
    return new EnumerableProjection<T>(table);
}

public static IEnumerable<Projection<T>> ToProjectionOfSet<T>(this Table table,
↪ IEnumerable<T> collection)
{
    return new EnumerableProjection<T>(table);
}

public static IEnumerable<Projection<T>> ToProjectionOfInstance<T>(this Table table, T
↪ instance)
{
    return new EnumerableProjection<T>(table);
}
```

Note that last arguments of `ToProjectionOfSet` and `ToProjectionOfInstance` methods are not used in method

implementation. Their only purpose is to bring information about T, so the `EnumerableProjection` adapter class can be built properly. Now we can perform the following comparisons with anonymous types collections and instances:

```
[Test]
public void Table_with_subset_of_columns_with_matching_values_should_match_collection()
{
    var table = CreateTableWithSubsetOfColumns();

    table.AddRow(1.ToString(), "a");
    table.AddRow(2.ToString(), "b");

    var query = from x in testCollection
                select new { x.GuidProperty, x.IntProperty, x.StringProperty };

    Assert.AreEqual(0, table.ToProjectionOfSet(query).Except(query.ToProjection()).
    ↪Count());
}

[Test]
public void Table_with_subset_of_columns_should_be_equal_to_matching_instance()
{
    var table = CreateTableWithSubsetOfColumns();

    table.AddRow(1.ToString(), "a");

    var instance = new { IntProperty = testInstance.IntProperty, StringProperty =
    ↪testInstance.StringProperty };

    Assert.AreEqual(table.ToProjectionOfInstance(instance), instance);
}
```

2.29 F# Support

`Bindings` for SpecFlow can be written also in F#. Doing so you can take the advantages of the F# language for writing step definitions: you can define regex-named F# functions for your steps. Simply put the regex between double backticks.

```
let [<Given>] ``I have entered (.*) into the calculator``(number:int) =
    Calculator.Push(number)
```

Although the regex method names are only important for `step definitions` you can also define `hooks` and `step argument conversions` in the F# binding projects.

Note: You need to create a C# or VB project for hosting the feature files and configure your F# project(s) as `external binding assemblies`:

```
<specFlow>
  <stepAssemblies>
    <stepAssembly assembly="MyFSharpBindings" />
  </stepAssemblies>
</specFlow>
```

2.29.1 IDE Support

SpecFlow provides item templates for creating new F# step definitions or hooks in Visual Studio.

Note: The navigation and the binding analysis features of the SpecFlow editor provide only limited support for F# projects.

2.29.2 Examples

An example can be found [here](#).

2.30 Executing SpecFlow Scenarios

In order to execute your SpecFlow tests, you need to define the tests as *Gherkin feature files*, *bind* the steps defined in your feature files to your code, and configure a unit test provider to execute the tests. SpecFlow generates executable unit tests from your Gherkin files.

We recommend that you add a separate project to your solution for your tests.

2.30.1 Configuring the Unit Test Provider

Tests are executed using a *unit test provider*. Add the corresponding NuGet package to your project to define your unit test provider:

- SpecRun.Runner
- SpecFlow.xUnit
- SpecFlow.MsTest
- SpecFlow.NUnit

You can only have one unit test provider!

2.30.2 Configuring the Unit Test Provider with SpecFlow 2 (Legacy)

Configure your unit test provider in your project's app.config file, e.g.:

```
<specFlow>
  <unitTestProvider name="MsTest" />
</specFlow>
```

2.31 Executing specific Scenarios in your Build pipeline

SpecFlow converts the tags in your feature files to test case categories:

- SpecFlow+ Runner: TestCategory
- NUnit: Category or TestCategory
- MSTest: TestCategory
- xUnit: Trait (similar functionality, SpecFlow will insert a Trait attribute with Category name)

This category can be used to filter the test execution in your build pipeline. Note that the incorrect filter can lead to no test getting executed.

You don't have to include the @ prefix in the filter expression.

Learn more about the filters in Microsoft's [official documentation](#).

2.31.1 Examples

All the examples here are using `Category`, but if you are using `MsTest` or `SpecFlow+ Runner` then you should use `TestCategory` instead.

How to use the filters

Below are 2 scenarios where one of them has a tag: @done, and the other one does not have a tag.

```
Feature: Breakfast

@done
Scenario: Eating cucumbers
    Given there are 12 cucumbers
    When I eat 5 cucumbers
    Then I should have 7 cucumbers

Scenario: Use all the sugar
    Given there is some sugar in the cup
    When I put all the sugar to my coffee
    Then the cup is empty
```

If we would like to run only the scenario with @done tag, then the filter should look like:

```
Category=done
```

Below are 2 scenarios where one of them has a tag: @done, and the other one has @automated.

```
Feature: Breakfast

@done
Scenario: Eating cucumbers
    Given there are 12 cucumbers
    When I eat 5 cucumbers
    Then I should have 7 cucumbers

@automated
Scenario: Use all the sugar
    Given there is some sugar in the cup
    When I put all the sugar to my coffee
    Then the cup is empty
```

If we would like to run scenarios which have either @done or @automated:

```
Category=done | Category=automated
```

Below are 2 scenarios where one of them has a tag: @done, and the other one has @automated. There is also a @US123 tag at Feature level.

```
@US123
Feature: Breakfast

@done
Scenario: Eating cucumbers
    Given there are 12 cucumbers
    When I eat 5 cucumbers
    Then I should have 7 cucumbers

@automated
Scenario: Use all the sugar
    Given there is some sugar in the cup
    When I put all the sugar to my coffee
    Then the cup is empty
```

If we would like to run only those scenarios, which have both @US123 and @done:

```
Category=US123 & Category=done
```

Below are 2 scenarios where one of them has two tags: @done and @important. There is another scenario, which has the @automated tag, and there is a @us123 tag at Feature level.

```
@US123
Feature: Breakfast

@done @important
Scenario: Eating cucumbers
    Given there are 12 cucumbers
    When I eat 5 cucumbers
    Then I should have 7 cucumbers

@automated
Scenario: Use all the sugar
    Given there is some sugar in the cup
    When I put all the sugar to my coffee
    Then the cup is empty
```

If we would like to run only those scenarios, which have both @done and @important:

```
Category=done & Category=important
```

dotnet test

Use the `--filter` command-line option:

```
dotnet test --filter Category=done
```

```
dotnet test --filter "Category=us123 & Category=done"
```

```
dotnet test --filter "Category=done | Category=automated"
```

vstest.console.exe

Use the `/TestCaseFilter` command-line option:

```
vstest.console.exe "C:\Temp\BookShop.AcceptanceTests.dll" /TestCaseFilter:"Category=done"
```

```
vstest.console.exe "C:\Temp\BookShop.AcceptanceTests.dll" /TestCaseFilter:  
↪ "Category=us123 & Category=done"
```

```
vstest.console.exe "C:\Temp\BookShop.AcceptanceTests.dll" /TestCaseFilter:"Category=done"  
↪ | Category=automated"
```

Azure DevOps - Visual Studio Test task

>Note: This task is supported only on Windows agents and cannot be used on other platforms.

The filter expression should be provided in the “Test filter criteria” setting in the Visual Studio Test task:

Visual Studio Test ⓘ [View YAML](#) [Remove](#)

Task version

Display name *

Test selection ^

Select tests using * ⓘ

Test files * ⓘ

Search folder * ⓘ

Test results folder ⓘ

Test filter criteria ⓘ

☐ Run only impacted tests ⓘ

☐ Test mix contains UI tests ⓘ

Visual Studio Test ⓘ View YAML Remove

Task version 2.* ▾

Display name *
VsTest - testAssemblies

Test selection ^

Select tests using * ⓘ
Test assemblies ▾

Test files * ⓘ

```

**\Debug\**\*Specs.dll
!*\*TestAdapter.dll
!*\obj\**

```

Search folder * ⓘ
\$(System.DefaultWorkingDirectory)

Test results folder ⓘ
\$(Agent.TempDirectory)\TestResults

Test filter criteria ⓘ
Category=us123 & Category=done

☐ Run only impacted tests ⓘ
 ☐ Test mix contains UI tests ⓘ

Azure DevOps - .NET Core task

Alternatively you could use the dotnet task (DotNetCoreCLI) to run your tests. This works on all kinds of build agents:

```

- task: DotNetCoreCLI@2
  displayName: 'dotnet test'
  inputs:
    command: test
    projects: 'BookShop.AcceptanceTests'
    arguments: '--filter "Category=done"'

```

```

- task: DotNetCoreCLI@2
  displayName: 'dotnet test'
  inputs:
    command: test
    projects: 'BookShop.AcceptanceTests'
    arguments: '--filter "Category=us123 & Category=done"'

```

2.32 Mark Steps as not implemented

To mark a step as not implemented at runtime, you need to throw a `PendingStepException`. The Runtime of SpecFlow will detect this and will report the appropriate test result back to your test runner.

There are multiple ways to throw the exception.

2.32.1 Using `ScenarioContext.Pending` helper method

The `ScenarioContext` class has a static helper method to throw the default `PendingStepException`.

Usage sample:

```
[When("I set the current ScenarioContext to pending")]
public void WhenIHaveAPendingStep()
{
    ScenarioContext.Pending();
}
```

This is the preferred way.

2.32.2 Throwing the `PendingStepException` by your own

You can also throw the Exception manually. In this case you have the possibility to provide a custom message.

Default Message

```
[When("I set the current ScenarioContext to pending")]
public void WhenIHaveAPendingStep()
{
    throw new PendingStepException();
}
```

Custom Message

```
[When("I set the current ScenarioContext to pending")]
public void WhenIHaveAPendingStep()
{
    throw new PendingStepException("custom pendingstep message");
}
```


2.33 Skipping Scenarios

Since SpecFlow 3.1 you can do skip programmatically Scenarios with the `UnitTestRuntimeProvider`.

2.33.1 Example Code

```
[Binding]
public sealed class StepDefinitions
{
    private readonly IUnitTestRuntimeProvider _unitTestRuntimeProvider;

    public CalculatorStepDefinitions(IUnitTestRuntimeProvider
↪unitTestRuntimeProvider)
    {
        _unitTestRuntimeProvider = unitTestRuntimeProvider;
    }

    [When("your binding")]
    public void YourBindingMethod()
    {
        _unitTestRuntimeProvider.TestIgnore("This scenario is always skipped");
    }
}
```

Ignoring is like skipping the scenario. Be careful, as it behaves a little bit different for the different unit test runners (xUnit, NUnit, MSTest, SpecFlow+ Runner).

2.33.2 Limitations

Currently this works only in *step definitions*. It is not possible to use it in *hooks*. See [GitHub Issue #2059](#)

2.34 Test Results

When SpecFlow tests are executed, the execution engine processes the test steps, executing the necessary test logic and either finishing successfully or failing for various reasons.

2.34.1 Test Passes

While executing the tests, the engine outputs information about the execution to the test output. In some cases it makes sense to investigate the test output even if the test passes.

By default, the test output includes the executed test steps, the invoked test logic methods (*bindings*) and the execution time for longer operations. You can *configure* the information displayed in the test output.

2.34.2 Test Fails due to an Error

A test can fail because it causes an error. The test output contains more detailed information, e.g. a stack trace.

2.34.3 Test Fails due to Missing, Pending or Improperly Configured Bindings

A test can fail if the test logic (bindings) have not yet been implemented (or are configured improperly). By default, this is reported as an “inconclusive” result, although you can *configure* how SpecFlow behaves in this case.

Note: Some unit test frameworks do not support inconclusive result. In this case the problem is reported as an error instead.

The test output can be very useful if you are missing bindings, as it contain a step binding method skeleton you can copy to your project and extend with the test logic.

2.34.4 Ignored Tests

Just like with normal unit tests, you can also ignore SpecFlow tests. To do so, tag the feature or scenario with the `@ignore` tag. Don't forget that ignoring a test will not solve any problems with your implementation... ;-)

2.35 Parallel Execution

SpecFlow scenarios are often automated as integration or system level tests. The system under test (SUT) might have several external dependencies and a more complex internal architecture. The key design question when running the tests in parallel is how the parallel test executions can be isolated from each other.

2.35.1 Test Isolation Levels

Determining the ideal level of isolation for your automated tests is a tradeoff. The higher the isolation of the parallel tests the smaller the likelihood of conflicts on shared state and dependencies, but at the same time the higher the execution time and amount of resources needed to maintain the isolated environments.

2.35.2 Parallel Scheduling Unit

Depending on the test isolation level and the used test runner tools you can consider different “units of scheduling” that can run in parallel with each other. When using SpecFlow we can consider the parallel scheduling on the level of scenarios, features and test assemblies.

2.35.3 Running SpecFlow features in parallel with thread-level isolation

Test	Duration	Traits
SpecsNUnit (9)	9,2 sec	
SpecsNUnit.Features (9)	9,2 sec	
F1Feature (3)	3,1 sec	
F1_S1	1 sec	
F1_S2	1 sec	
F1_S3	1 sec	
F2Feature (3)	3,1 sec	
F2_S1	1 sec	
F2_S2	1 sec	
F2_S3	1 sec	
F3Feature (3)	3,1 sec	
F3_S1	1 sec	
F3_S2	1 sec	
F3_S3	1 sec	

Properties

- Tests are running in multiple threads within the same process and the same application domain.
- Only the thread-local state is isolated.
- Smaller initialization footprint and lower memory requirements.
- The SpecFlow binding registry (step definitions, hooks, etc.) and some other core services are shared across test threads.

Requirements

- You have to use a test runner that supports in-process parallel execution (currently NUnit v3, xUnit v2, MSTest and SpecFlow+ Runner)
- You have to ensure that your code does not conflict on static state.
- You must not use the static context properties of SpecFlow `ScenarioContext.Current`, `FeatureContext.Current` or `ScenarioStepContext.Current` (see further information below).
- You have to configure the test runner to execute the SpecFlow features in parallel with each other (see configuration details below).

Execution Behavior

- [BeforeTestRun] and [AfterTestRun] hooks (events) are executed only once on the first thread that initializes the framework. Executing tests in the other threads is blocked until the hooks have been fully executed on the first thread.
- All scenarios in a feature must be executed on the **same thread**. See the configuration of the test runners below. This ensures that the [BeforeFeature] and [AfterFeature] hooks are executed only once for each feature and that the thread has a separate (and isolated) FeatureContext.
- Scenarios and their related hooks (Before/After scenario, scenario block, step) are isolated in the different threads during execution and do not block each other. Each thread has a separate (and isolated) ScenarioContext.
- The test trace listener (that outputs the scenario execution trace to the console by default) is invoked asynchronously from the multiple threads and the trace messages are queued and passed to the listener in serialized form. If the test trace listener implements TechTalk.SpecFlow.Tracing.IThreadSafeTraceListener, the messages are sent directly from the threads.

NUnit Configuration

By default, **NUnit** does not run the tests in parallel. Parallelisation must be configured by setting an assembly-level attribute in the SpecFlow project.

```
using NUnit.Framework;  
[assembly: Parallelizable(ParallelScope.Fixtures)]
```

>Note: SpecFlow does not support scenario level parallelization with NUnit (when scenarios from the same feature execute in parallel). If you configure a higher level NUnit parallelization than “Fixtures” your tests will fail with runtime errors.

MSTest Configuration

By default, **MsTest** does not run the tests in parallel. Parallelisation must be configured by setting an assembly-level attribute in the SpecFlow project.

```
using Microsoft.VisualStudio.TestTools.UnitTesting;  
[assembly: Parallelize(Scope = ExecutionScope.ClassLevel)]
```

>Note: SpecFlow does not support scenario level parallelization with MsTest (when scenarios from the same feature execute in parallel). If you configure a higher level MsTest parallelization than “ClassLevel” your tests will fail with runtime errors.

xUnit Configuration

By default xUnit runs all SpecFlow features in parallel with each other. No additional configuration is necessary.

Thread-safe ScenarioContext, FeatureContext and ScenarioStepContext

When using parallel execution accessing the obsolete `ScenarioContext.Current`, `FeatureContext.Current` or `ScenarioStepContext.Current` static properties is not allowed. Accessing these static properties during parallel execution throws a `SpecFlowException`.

To access the context classes in a thread-safe way you can either use context injection or the instance properties of the Steps base class. For further details please see the [FeatureContext](#) and [ScenarioContext](#) documentation.

Excluding SpecFlow features from parallel execution

To exclude specific features from running in parallel with any other features, see the `addNonParallelizableMarkerForTags` [configuration](#) option.

Please note that xUnit requires additional configuration to ensure that non parallelizable features do not run in parallel with any other feature. This configuration is automatically provided for users via the xUnit plugin (so no additional effort is required). The following class will be defined within your test assembly for you:

```
[CollectionDefinition("SpecFlowNonParallelizableFeatures", DisableParallelization =   
↪true)]  
public class SpecFlowNonParallelizableFeaturesCollectionDefinition  
{  
}
```

2.35.4 Running SpecFlow scenarios in parallel with AppDomain or Process isolation

If there are no external dependencies or they can be cloned for parallel execution, but the application architecture depends on static state (e.g. static caches etc.), the best way is to execute tests in parallel isolated by AppDomain or Process. This ensures that every test execution thread is hosted in a separate AppDomain and hence static state is not accessed in parallel. In such scenarios, SpecFlow+Runner can be used to execute tests in parallel without any extra considerations. [SpecFlow+ Runner supports parallel execution](#) with AppDomain, SharedAppDomain and Process isolation.

Properties

- Tests threads are separated by an AppDomain or process boundary.
- Also the static memory state is isolated. Conflicts might be expected on external dependencies only.
- Bigger initialization footprint and higher memory requirements.

Requirements

- You have to use SpecFlow+ Runner with AppDomain or Process isolation.

Execution Behavior

- [BeforeTestRun] and [AfterTestRun] hooks are executed for each individual test execution thread (App-Domain or process), so you can use them to initialize/reset shared memory.
- Each test thread manages its own enter/exit feature execution workflow. The [BeforeFeature] and [AfterFeature] hooks may be executed multiple times in different test threads if they run scenarios from the same feature file. The execution of these hooks do not block one another, but the Before/After feature hooks are called in pairs within a single thread (the [BeforeFeature] hook of the next scenario is only executed after the [AfterFeature] hook of the previous one). Each test thread has a separate (and isolated) `FeatureContext`.

2.36 Debugging

SpecFlow Visual Studio integration also supports debugging the execution of your tests. Just like in the source code files of your project, you can place breakpoints in the SpecFlow feature files. Whenever you execute the generated tests in debug mode, the execution will stop at the specified breakpoints and you can execute the steps one-by-one using “Step Over” (F10), or follow the detailed execution of the bindings using “Step Into” (F11).

If the execution of a SpecFlow test is stopped at a certain point of the binding (e.g. because of an exception), you can navigate to the current step in the feature file from the “Call Stack” tool window in Visual Studio.

By default, you cannot debug inside the generated .feature.cs files. You can enable debugging for these files by setting *generator allowDebugGeneratedFiles="true"*.

2.37 Output API

The SpecFlow Output API allows you to display texts and attachments in your IDE's test explorer output window and also in SpecFlow+LivingDoc.

To use the SpecFlow output API interface you must inject the `ISpecFlowOutputHelper` interface via *Context Injection*:

```
private readonly ISpecFlowOutputHelper _specFlowOutputHelper;

public CalculatorStepDefinitions(ISpecFlowOutputHelper outputHelper)
{
    _outputHelper = outputHelper;
}
```

There are two methods available:

2.37.1 WriteLine(string text)

This method adds text:

```
_specFlowOutputHelper.WriteLine("TEXT");
```

2.37.2 AddAttachment(string filePath)

This method adds an attachment and requires the file path:

```
_specFlowOutputHelper.AddAttachment("filePath");
```

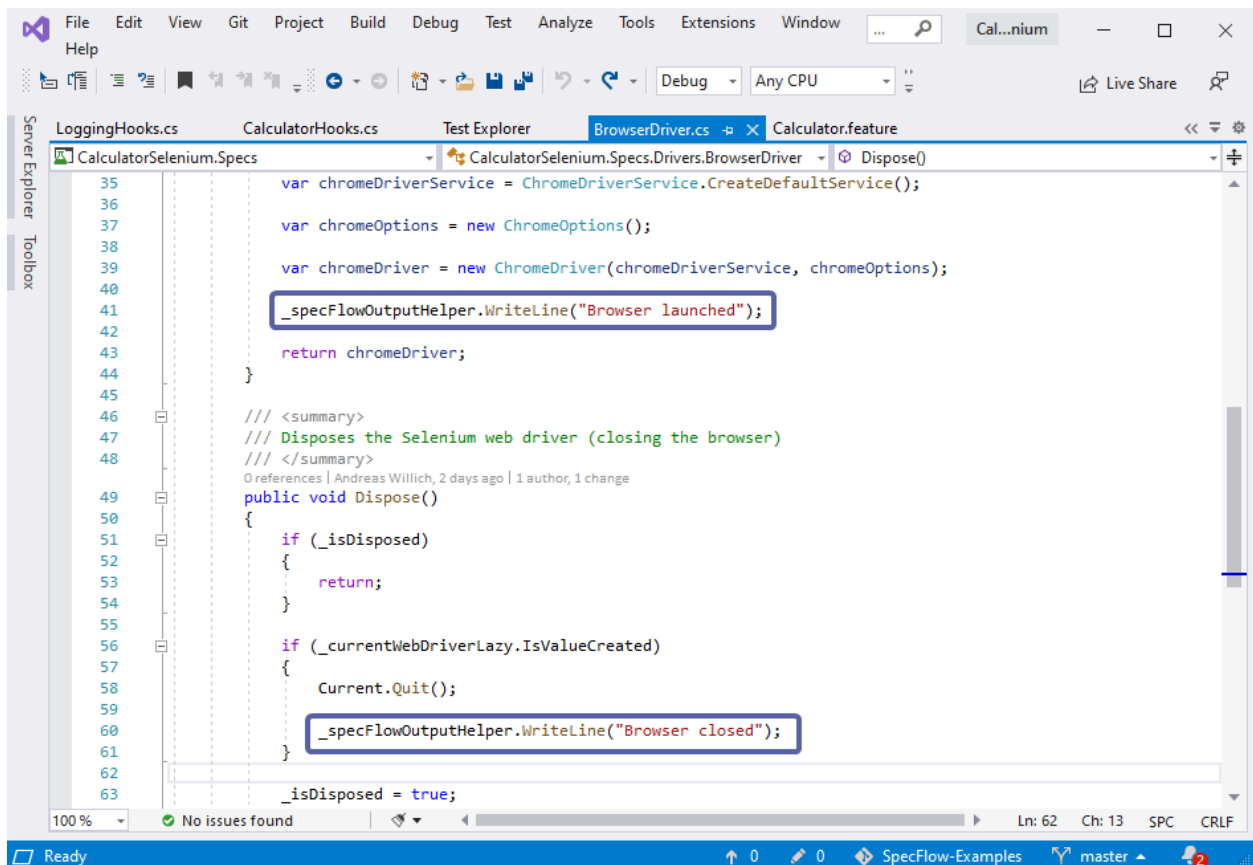
> Note: The attachment file can be stored anywhere. But it is important to keep mind that if a local file is added, it will only work on your machine and not accessible when shared with others.

> Note: Handling of attachments depends on your runner. MStest and NUnit currently support this feature but xUnit and SpecFlow+ Runner do **not**.

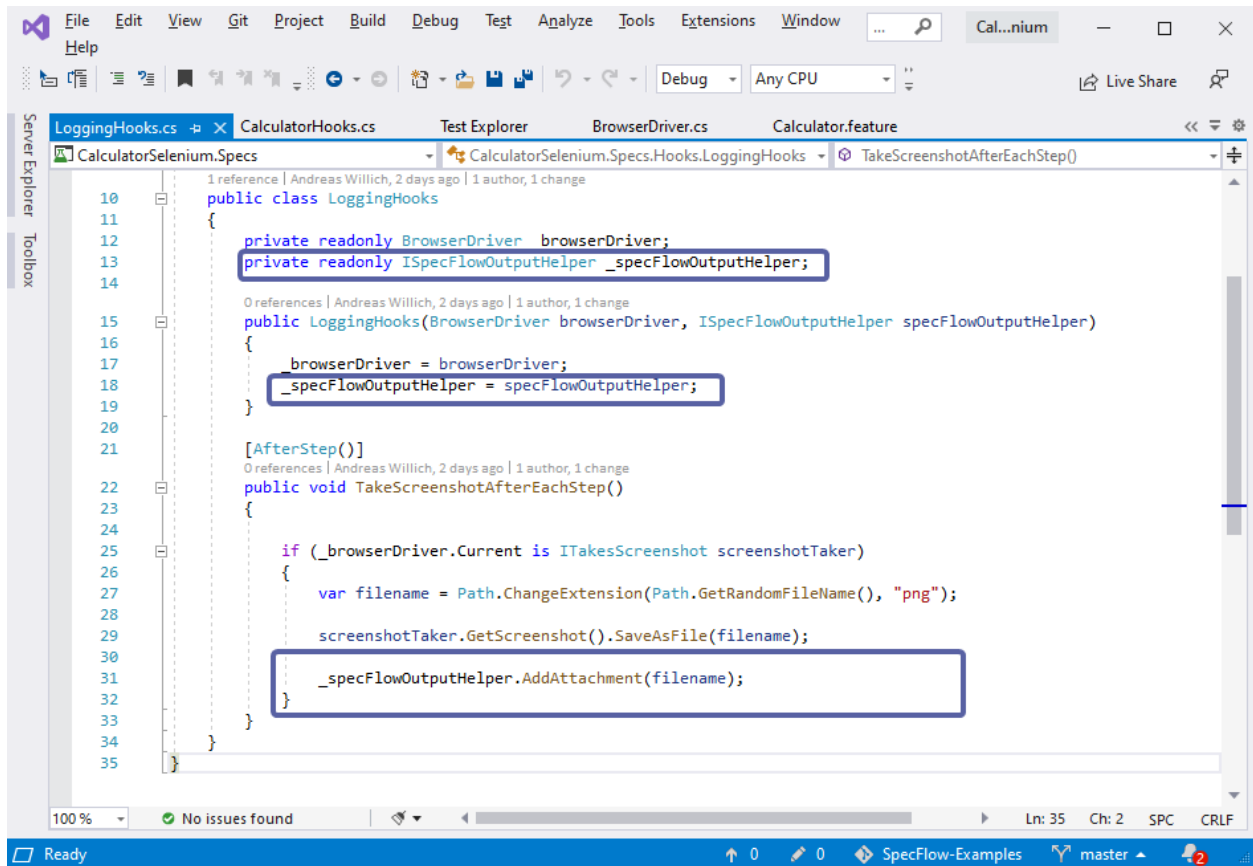
2.37.3 Example

This example is based on the *Selenium with Page Object Model Pattern* project which tests a simple calculator web application. You can download the repo for this example [here](#).

The `_specFlowOutputHelper.WriteLine` is used to indicate when the browser launches and closes:

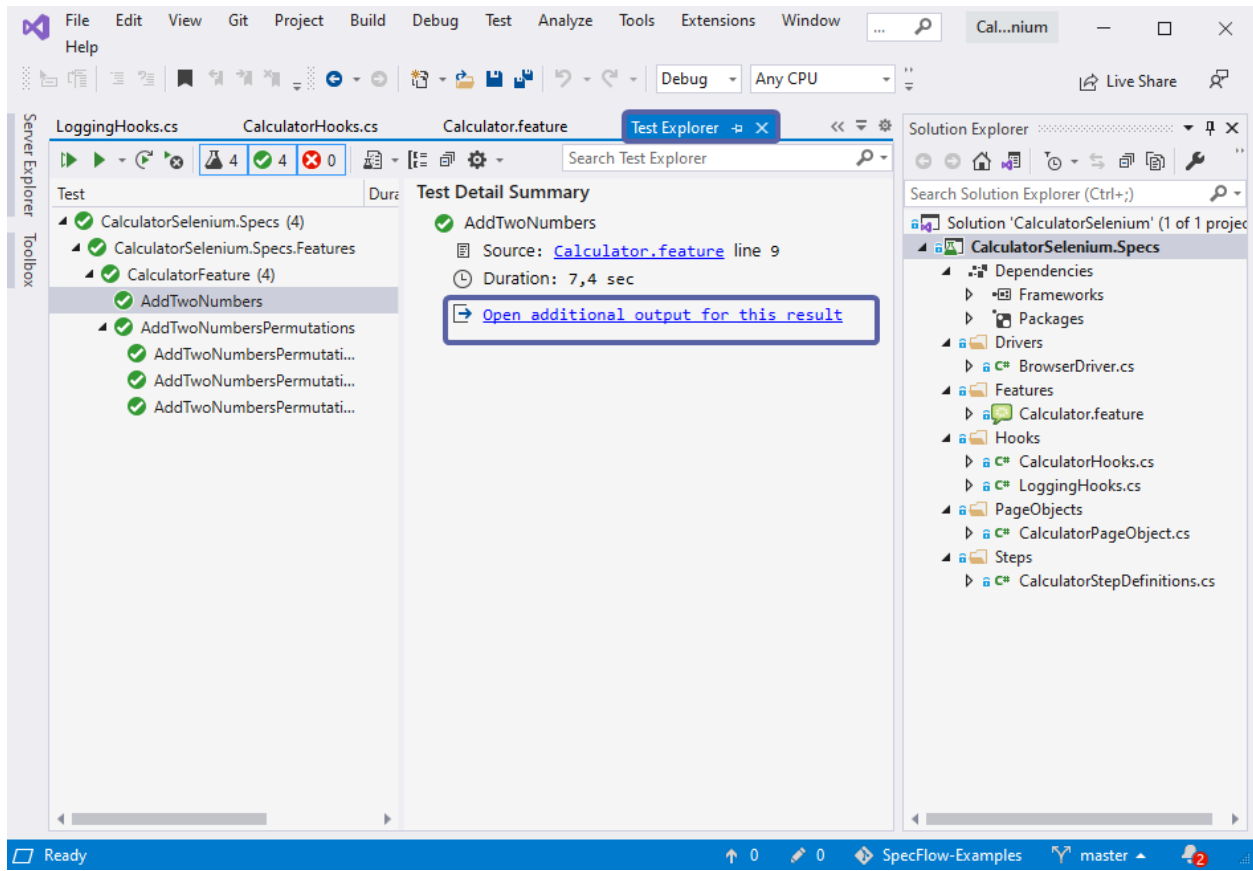


Since this project tests a web application using a browser, the `_specFlowOutputHelper.AddAttachment` method has been used in the logging *Hooks* to display the saved screen shots taken during testing:

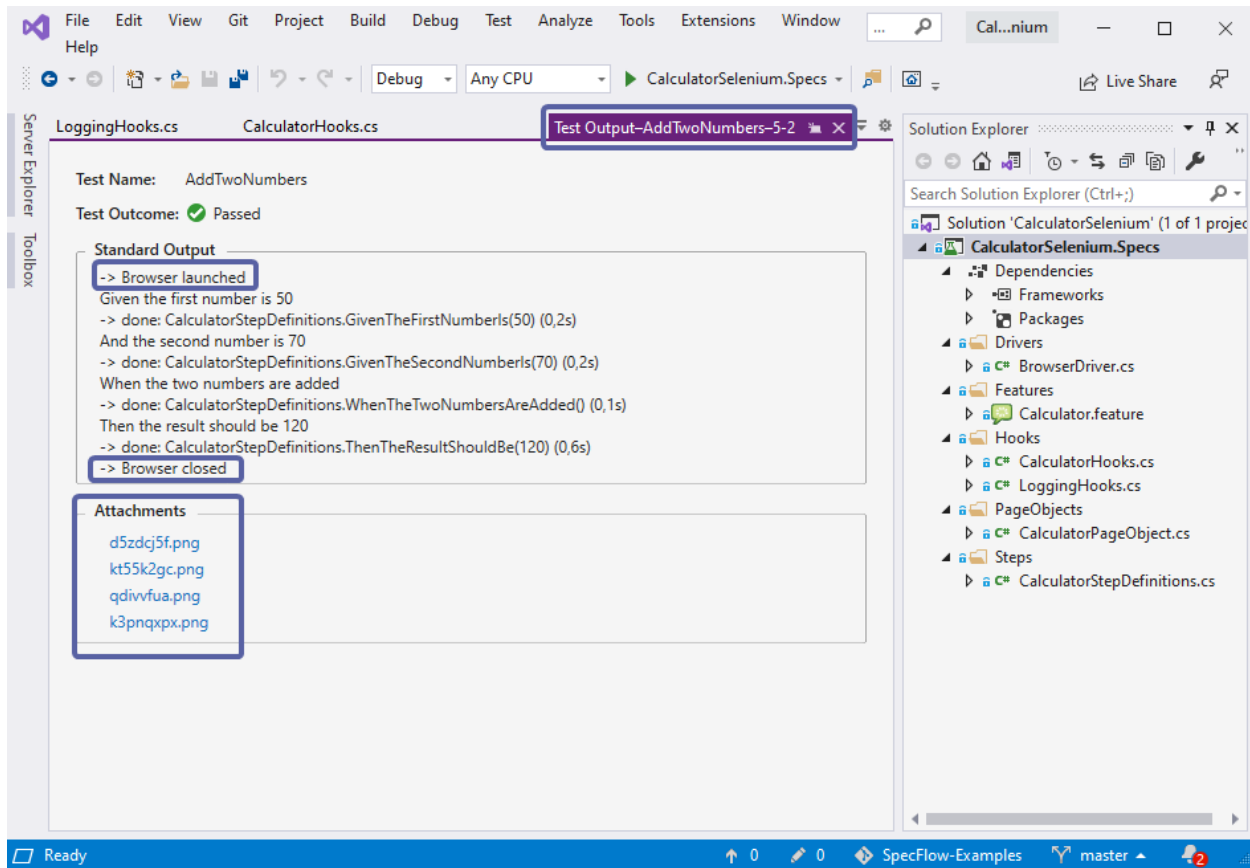


Results

To view the output window in Visual Studio, go to the text explorer and click on *Open Additional output for this result*:



The resulting output for the example project used above would look like this:



The added text lines on browser launch and termination can help you easily identify the exact point at which the action takes place. The screen shots taken during testing are all also listed as image files under *Attachments*.

If an attachment fails, the output explorer will display this message:

```
-> Attachment 'I' added (not forwarded to the test runner).
```

In SpecFlow+LivingDoc, no additional setup is required, simply [generate LivingDoc](#) as you normally do. You can then view the output texts and attachments by toggling the *Show/Hide Test Output* :

CalculatorSelenium.Specs

generated May 14, 2021, 11:09 AM GMT+2

Living Documentation Analytics

Filter by Keyword Filter by Scenario Result X

Test results ☒

CalculatorSelenium.Specs 1 Passed 0 Failed 0 Others

Features 1 Passed 0 Failed 0 Others

Calculator

- Add two numbers
- Add two numbers permutations

@Calculator

Feature: Calculator

Simple calculator for adding **two** numbers

Link to a feature: [Calculator](#)


Further read: [Learn more about how to generate Living Documentation](#)

@Calculator


Scenario: Add two numbers 791ms

Browser launched BeforeScenario

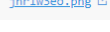
Given the first number is 50

AfterStep 

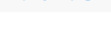
And the second number is 70

AfterStep 

When the two numbers are added

AfterStep 

Then the result should be 120

AfterStep 

> Note: If the test output toggle is missing, it may be that you are on an older version of SpecFlow+LivingDoc, click [here](#) to update to the latest version.

> Note: The Output API in SpecFlow+LivingDoc supports the following four Hooks :

- BeforeScenario,
- AfterScenario,
- BeforeStep,
- AfterStep

2.38 Color Test Result Output

2.38.1 Configuration

To enable the colorization of the test result output, you can turn the `trace.coloredOutput` to true in the [configuration](#)

The color will only be visible in supported place, like in Rider test runner or in the console when running test using `dotnet test`.

You can turn off the color by setting `NO_COLOR=1` environment variable. This can be useful when you run the tests on a build server that does not support colors.

2.38.2 Customization

You can customize the colors by configuring a Hook and injecting `IColorOutputTheme` like in the following example.

```
[Binding]
public class Hooks
{
    [BeforeTestRun]
    public static void ConfigureColor(IColorOutputTheme colorOutputTheme)
    {
        colorOutputTheme.Keyword = AnsiColor.Reset;
        colorOutputTheme.Error = AnsiColor.Composite(AnsiColor.Bold, AnsiColor.
↪Foreground(TerminalRgbColor.FromHex("FF8EF3")));
        colorOutputTheme.Done = AnsiColor.Foreground(TerminalRgbColor.FromHex(
↪"3A86FF"));
    }
}
```

2.39 SpecFlow+ Runner

SpecFlow+ Runner (formerly “SpecRun”) is a dedicated test execution framework for SpecFlow. SpecFlow+ Runner integrates more tightly with Visual Studio's testing infrastructure and Team Foundation Server (TFS) Build. The documentation for SpecFlow+ can be found [here](#).

2.39.1 Installation

SpecFlow+ Runner is provided as a NuGet package ([SpecRun.SpecFlow](#)). Detailed setup instructions can be found [here](#).

2.39.2 Visual Studio Test Explorer Support

SpecFlow+ Runner allows you to run and debug your scenarios as first class citizens:

- Run/debug individual scenarios or scenario outline examples from the feature file editor (choose “Run/Debug SpecFlow Scenarios” from the context menu)
- View scenarios in the Visual Studio Test Explorer window with the scenario title
- Use the Test Explorer to:
 - Group scenarios by tags (choose “Traits” grouping) and features (choose “Class”)
 - Filter scenarios by different criteria
 - Run/debug selected/all scenarios
 - Jump to the corresponding scenario in the feature file
 - View test execution results
- You can specify [processor architecture \(x86/x64\)](#), [.NET platform](#) and many other details for the test execution, including special [config file transformations](#) used for the test execution only.

2.39.3 Team Foundation Server Support

The SpecRun NuGet package contains all necessary integration components for Team Foundation Server Build, and you do not need to make any additional configuration or build process template modifications for TFS Build to execute your scenarios. You can also:

- Display scenario titles in the execution result
- Generate detailed and customizable HTML report
- Filter scenarios in the TFS build definition

More information on using SpecFlow+ Runner with build servers can be found [here](#).

2.39.4 Test Execution Features

SpecFlow+ Runner is a smarter integration test runner for SpecFlow:

- Faster feedback: parallel test execution and smart execution order
- More information: advanced metrics, detecting random failures, historical execution statistics
- Not limited to SpecFlow: execute integration tests written with other unit testing frameworks

See the short introduction video about the [configurable test execution environments](#) and about [parallel test execution](#).

2.40 MSTest

SpecFlow supports MsTest V2.

Documentation for MSTest can be found [here](#).

2.40.1 Needed NuGet Packages

For SpecFlow: `SpecFlow.MSTest`

For MSTest: `MSTest.TestFramework`

For Test Discovery & Execution:

- `MSTest.TestAdapter`
- `Microsoft.NET.Test.Sdk`

2.40.2 Accessing TestContext

You can access the MsTest TestContext instance in your step definition or hook classes by constructor injection:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

[Binding]
public class MyStepDefs
{
    private readonly TestContext _testContext;
    public MyStepDefs(TestContext testContext) // use it as ctor parameter
    {
```

(continues on next page)

(continued from previous page)

```
{
    _testContext = testContext;
}

[Given("a step")]
public void GivenAStep()
{
    //you can access the TestContext injected in the ctor
    _testContext.WriteLine(_testContext.TestRunDirectory);
}

[BeforeScenario()]
public void BeforeScenario()
{
    //you can access the TestContext injected in the ctor
    _testContext.WriteLine(_testContext.TestRunDirectory);
}
}
```

In the static BeforeTestRun/AfterTestRun hooks you can use parameter injection:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

[Binding]
public class Hooks
{
    [BeforeTestRun]
    public static void BeforeTestRun(TestContext testContext)
    {
        //you can access the TestContext injected as parameter
        testContext.WriteLine(testContext.TestRunDirectory);
    }

    [AfterTestRun]
    public static void AfterTestRun(TestContext testContext)
    {
        //you can access the TestContext injected as parameter
        testContext.WriteLine(testContext.DeploymentDirectory);
    }
}
```

2.40.3 Tags for TestClass Attributes

The MsTest Generator can generate test class attributes from tags specified on a **feature**.

Owner

Tag:

```
@Owner: John
```

Output:

```
[Microsoft.VisualStudio.TestTools.UnitTesting.OwnerAttribute("John")]
```

WorkItem

Tag:

```
@WorkItem: 123
```

Output:

```
[Microsoft.VisualStudio.TestTools.UnitTesting.WorkItemAttribute(123)]
```

DeploymentItem

Example 1 : Copy a file to the same directory as the deployed test assemblies

Tag:

```
@MsTest:DeploymentItem:test.txt
```

Output:

```
[Microsoft.VisualStudio.TestTools.UnitTesting.DeploymentItemAttribute("test.txt")]
```

Example 2 : Copy a file to a sub-directory relative to the deployment directory

Tag:

```
@MsTest:DeploymentItem:Resources\DeploymentItemTestFile.txt:Data
```

Output:

```
[Microsoft.VisualStudio.TestTools.UnitTesting.DeploymentItemAttribute("Resources\\  
↪DeploymentItemTestFile.txt", "Data")]
```

2.41 NUnit

SpecFlow supports NUnit 3.13.1 or later.

Documentation for NUnit can be found [here](#).

2.41.1 Needed NuGet Packages

For SpecFlow: [SpecFlow.NUnit](#)

For NUnit: [NUnit](#)

For Test Discovery & Execution:

- [NUnit3TestAdapter](#)
- [Microsoft.NET.Test.Sdk](#)

2.42 xUnit

SpecFlow supports xUnit 2.4 or later.

Documentation for xUnit can be found [here](#).

2.42.1 Needed NuGet Packages

For SpecFlow: [SpecFlow.xUnit](#)

For xUnit: [xUnit](#)

For Test Discovery & Execution:

- [xunit.runner.visualstudio](#)
- [Microsoft.NET.Test.Sdk](#)

2.42.2 Access ITestOutputHelper

The xUnit ITestOutputHelper is registered in the ScenarioContainer. You can get access to simply via getting it via *Context-Injection*.

Example

```
using System;
using TechTalk.SpecFlow;

[Binding]
public class BindingClass
{
    private Xunit.Abstractions.ITestOutputHelper _testOutputHelper;
    public BindingClass(Xunit.Abstractions.ITestOutputHelper testOutputHelper)
    {
```

(continues on next page)

(continued from previous page)

```
        _testOutputHelper = testOutputHelper;
    }

    [When(@"I do something")]
    public void WhenIDoSomething()
    {
        _testOutputHelper.WriteLine("EB7C1291-2C44-417F-ABB7-A5154843BC7B");
    }
}
```

2.43 Azure DevOps (Server)

2.43.1 Build Integration

The easiest way to execute SpecFlow scenarios on Azure DevOps (Team Foundation Server (TFS) Build is to use [SpecFlow+ Runner](#) as unit test provider (see [\[SpecFlow+ Runner Integration\]](#)). The SpecRun NuGet package contains all necessary integration components, and you don't need to do any additional configuration or build process template modification to let TFS build execute your scenarios, and even more:

- Display scenario titles in the execution result
- Generate detailed and customizable HTML report
- Allows filtering scenarios in the TFS build definition
- The integration also works with the hosted [Azure DevOps Server](#)

Legacy Integration

As SpecFlow generates unit test code from the scenarios, the tests can be executed on any build server (as unit tests). The configuration depends on the unit test provider used.

2.44 TeamCity Integration

- Automate via PowerShell runner and NuGet. Get more information [here](#)

2.45 Browserstack

If you want to perform web testing on multiple browsers and operating systems, it can be quite complicated to maintain machines for each of the target environments. BrowserStack provides “remote web browsers as a service”, making it easy to do this sort of matrix testing without having to maintain the multiple browser installations yourself. Specflow provides easy integration with BrowserStack.

To help you use Browserstack and Selenium together with SpecFlow we have put together [this plugin](#). Check it out for more details.

2.46 Autofac

2.46.1 Introduction

SpecFlow plugin for using Autofac as a dependency injection framework for step definitions.

Currently supports

Autofac v4.0.0 or above

2.46.2 Step by step walkthrough of using SpecFlow.Autofac

1. Install plugin from NuGet into your SpecFlow project.

```
PM> Install-Package SpecFlow.Autofac
```

2. Create static methods somewhere in the SpecFlow project

Plugin supports both registration of dependencies globally and per scenario:

2.1 Optionally configure dependencies that need to be shared globally for all scenarios:

Create a static method somewhere in the SpecFlow project to register scenario dependencies: (Recommended to put it into the Support folder) that returns void and has one parameter of Autofac ContainerBuilder, tag it with the [GlobalDependencies] attribute.

When registering global dependencies, it is also a requirement to configure scenario dependencies as well in order to register classes marked with the [Binding] attribute as shown below.

Globally registered dependencies may be resolved in the [BeforeTestRun] and [AfterTestRun] methods.

2.2 Configure dependencies to be resolved each time for a scenario:

Create a static method somewhere in the SpecFlow project to register scenario dependencies: (Recommended to put it into the Support folder) that returns void and has one parameter of Autofac ContainerBuilder, tag it with the [ScenarioDependencies] attribute.

2.3 Configure your dependencies for the scenario execution within either the two methods [GlobalDependencies] and [ScenarioDependencies] or the single [ScenarioDependencies] method.

2.4 You also have to register the step definition classes in the [ScenarioDependencies] method, that you can do by either registering all public types from the SpecFlow project:

```
builder.RegisterAssemblyTypes(typeof(YourClassInTheSpecFlowProject).Assembly).  
    .SingleInstance();
```

2.5 or by registering all classes marked with the [Binding] attribute:

You may use a provided extension method to do this, but importing:

```
using SpecFlow.Autofac.SpecFlowPlugin;
```

Then

```
containerBuilder.AddSpecFlowBindings(typeof(YourClassInTheSpecFlowProject))
```

Or overload

```
containerBuilder.AddSpecFlowBindings<YourClassInTheSpecFlowProject>()
```

Or manually register like so:

```
builder
    .RegisterAssemblyTypes(typeof(TestDependencies).Assembly)
    .Where(t => Attribute.IsDefined(t, typeof(BindingAttribute)))
    .SingleInstance();
```

3. A typical dependency builder method for [GlobalDependencies] with [ScenarioDependencies] probably looks like this:

```
[GlobalDependencies]
public static void CreateGlobalContainer(ContainerBuilder containerBuilder)
{
    // Register globally scoped runtime dependencies
    Dependencies.RegisterGlobalDependencies(containerBuilder);

    //TODO: add Services that are shared globally.
}

[ScenarioDependencies]
public static void CreateContainerBuilder(ContainerBuilder containerBuilder)
{
    // Register scenario scoped runtime dependencies
    Dependencies.RegisterScenarioDependencies(containerBuilder);

    //TODO: add customizations, stubs required for testing

    containerBuilder.AddSpecFlowBindings<TestDependencies>()
}
```

4. It is also possible to continue to use the legacy method as well, however this method is not compatible with global dependency registration and can only be used on it's own like so:

Create a static method somewhere in the SpecFlow project to register scenario dependencies: (Recommended to put it into the Support folder) that returns an Autofac ContainerBuilder and tag it with the [ScenarioDependencies] attribute.

2.47 Castle Windsor

2.47.1 Introduction

SpecFlow plugin for using Castle Windsor as a dependency injection framework for step definitions.

Currently supports

Castle Windsor v5.0.1 or above

2.47.2 Step by step walkthrough of using SpecFlow.Windsor

1. Install plugin

- Install plugin from NuGet into your SpecFlow project.

```
PM> Install-Package SpecFlow.Windsor
```

2. Create static method

- Create a static method somewhere in the SpecFlow project

(Recommended to put it into the Support folder) that returns a Windsor IWindsorContainer and tag it with the [ScenarioDependencies] attribute.

- Configure your dependencies for the scenario execution within the method.

- All your binding classes are automatically registered, including ScenarioContext etc.

3. Sample dependency builder method

- A typical dependency builder method probably looks like this:

```
[ScenarioDependencies]
public static IWindsorContainer CreateContainer()
{
    var container = new WindsorContainer();

    //TODO: add customizations, stubs required for testing

    return container;
}
```

4. Reusing a container

- To re-use a container between scenarios, try the following:

Your shared services will be resolved from the root container, while scoped objects such as `ScenarioContext` will be resolved from the new container.

```
[ScenarioDependencies]
public static IWindsorContainer CreateContainer()
{
    var container = new WindsorContainer();
    container.Parent = sharedRootContainer;

    return container;
}
```

5. Customize binding behavior

- To customize binding behavior, use the following:

Default behavior is to auto-register bindings. To manually register these during `CreateContainer` you can use the following attribute:

```
[ScenarioDependencies(AutoRegisterBindings = false)]
public static IWindsorContainer CreateContainer()
{
    // Register your bindings here
}
```

2.48 Integration

The Visual Studio integration includes a number of features that make it easier to edit Gherkin files and navigate to and from bindings in Visual Studio. You can also generate skeleton code including step definition methods from feature files. The Visual Studio integration also allows you to execute tests from Visual Studio's Test Explorer.

You can install the integration from Visual Studio Gallery (Marketplace) or directly in Visual Studio. Detailed instructions can be found [here](#).

The integration provides the following features:

- Editor
 - Gherkin syntax highlighting in feature files, highlight unbound steps and parameters
 - IntelliSense (auto-completion) for keywords and steps
 - Outlining (folding) sections of the feature file
 - Comment/uncomment feature file lines
 - Automatic Gherkin table formatting
 - Document formatting
- Navigation
 - Navigate from steps in scenarios to binding methods and vice versa

- Detect bindings from the SpecFlow project, from project references and from assembly references
- Cached step analysis for faster solution startup
- Generic Test Runner Support
 - You can execute tests using the following test runners: Visual Studio, ReSharper and SpecRun. You can execute SpecFlow scenarios on all supported unit testing platforms (e.g. NUnit, xUnit, MSTest).
- Visual Studio Test Explorer Support
 - Run/debug (including from feature files)
 - Scenario title displayed in Test Explorer
 - Full access to Test Explorer functions
- Other
 - Generate [skeleton step definition methods](#) from feature files
 - Re-generate feature files (from project node context menu and automatically on configuration change)
 - Configurable options
 - Support for ReSharper command shortcuts (when ReSharper is installed): commenting, navigation, test execution

2.48.1 Troubleshooting

If you are having trouble with the Visual Studio integration, refer to the [Troubleshooting page](#) first.

2.49 Installation

The integration packages can also be downloaded and installed separately from the Visual Studio Gallery:

- [VS2019 integration](#)
- [VS2017 integration](#)
- [VS2022 integration](#)

or through the Visual Studio extensions:

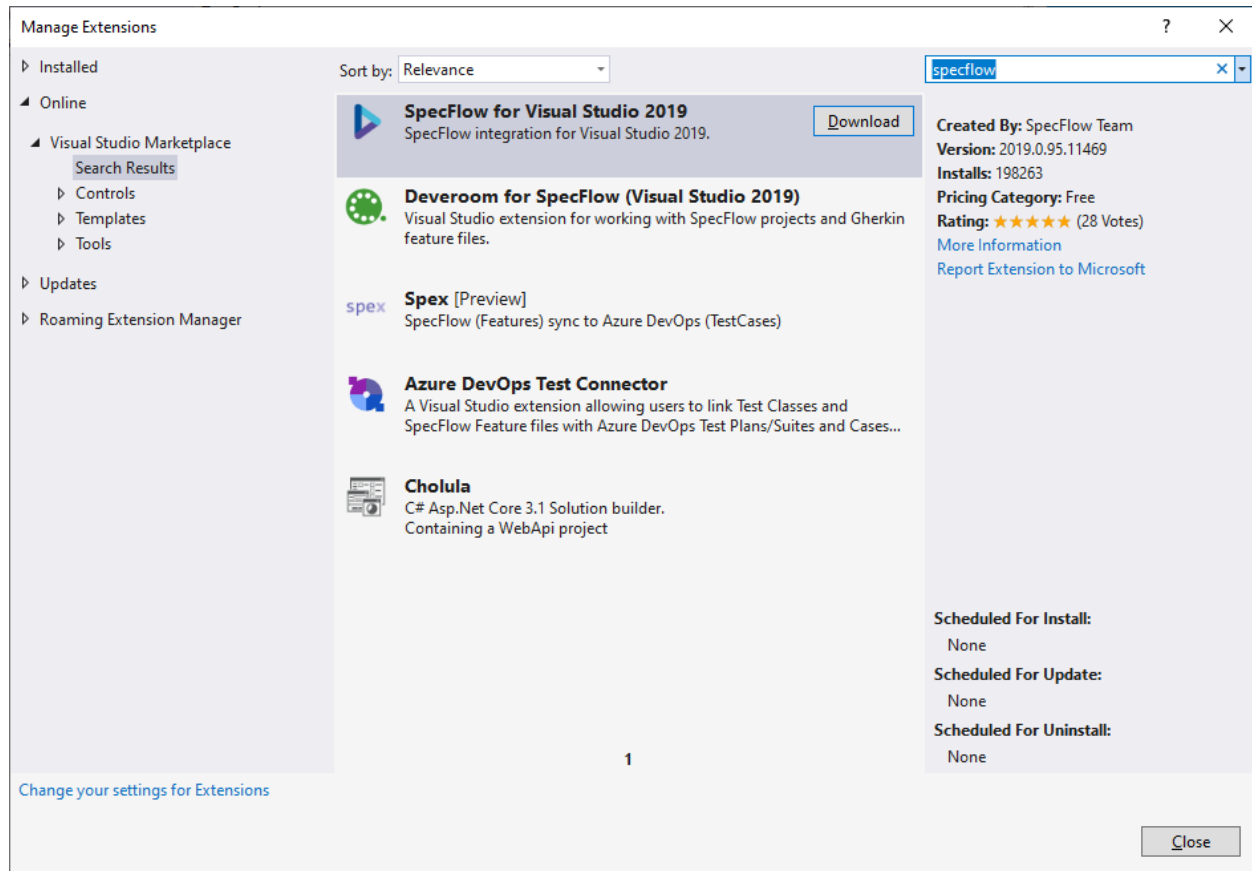
1- Open Visual Studio.

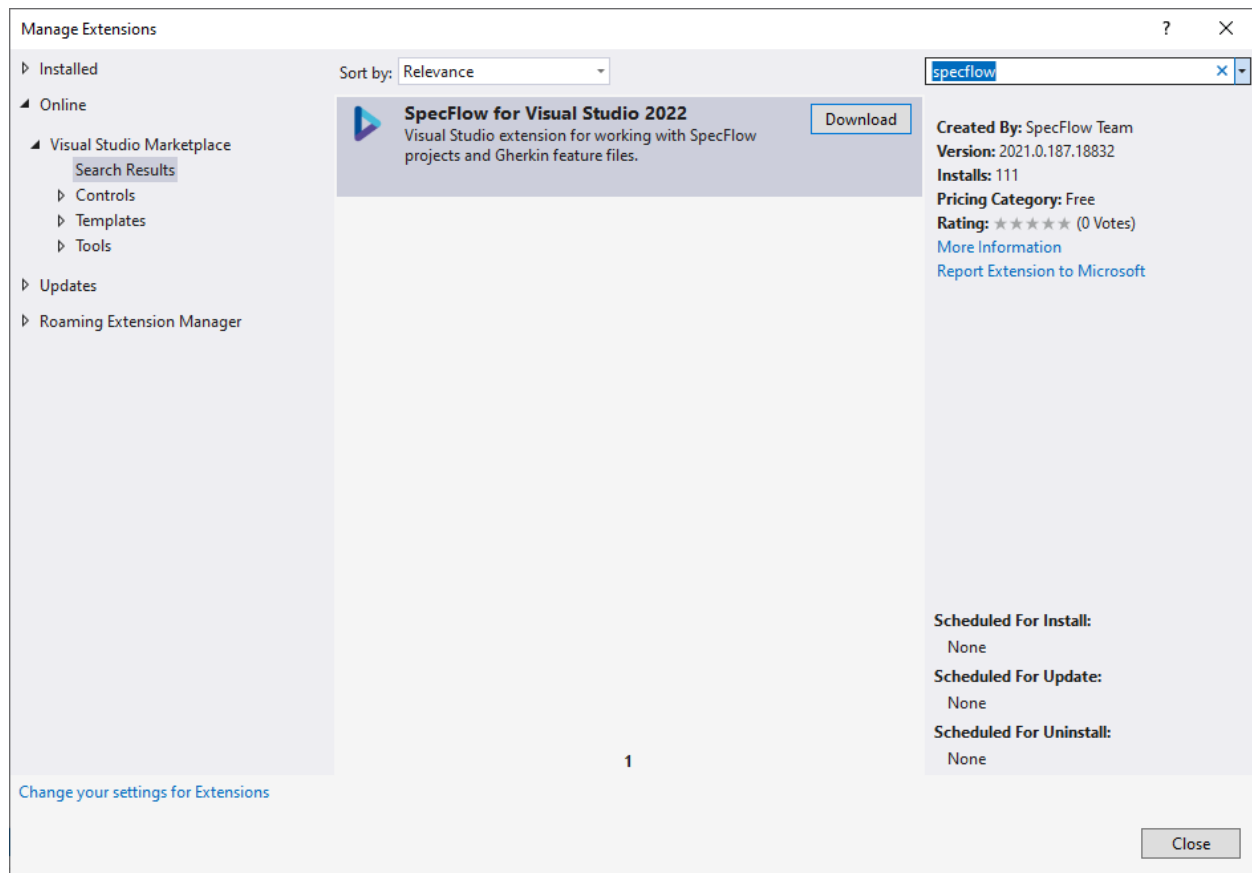
2- Navigate to ***Extensions Manage Extensions Online*** and search for “SpecFlow” in the search bar.

3- Hit ***Download*** to begin the installation. You will need to restart Visual Studio for the installation to complete

VS2019

VS2022





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

The below video is from our getting started guide which covers the installation of SpecFlow’s Visual Studio extension. If you are new to SpecFlow we highly recommend you to go through this [getting started guide](#)

2.50 Extension Settings/Options

The extension settings can be configured as per below:

VS2019

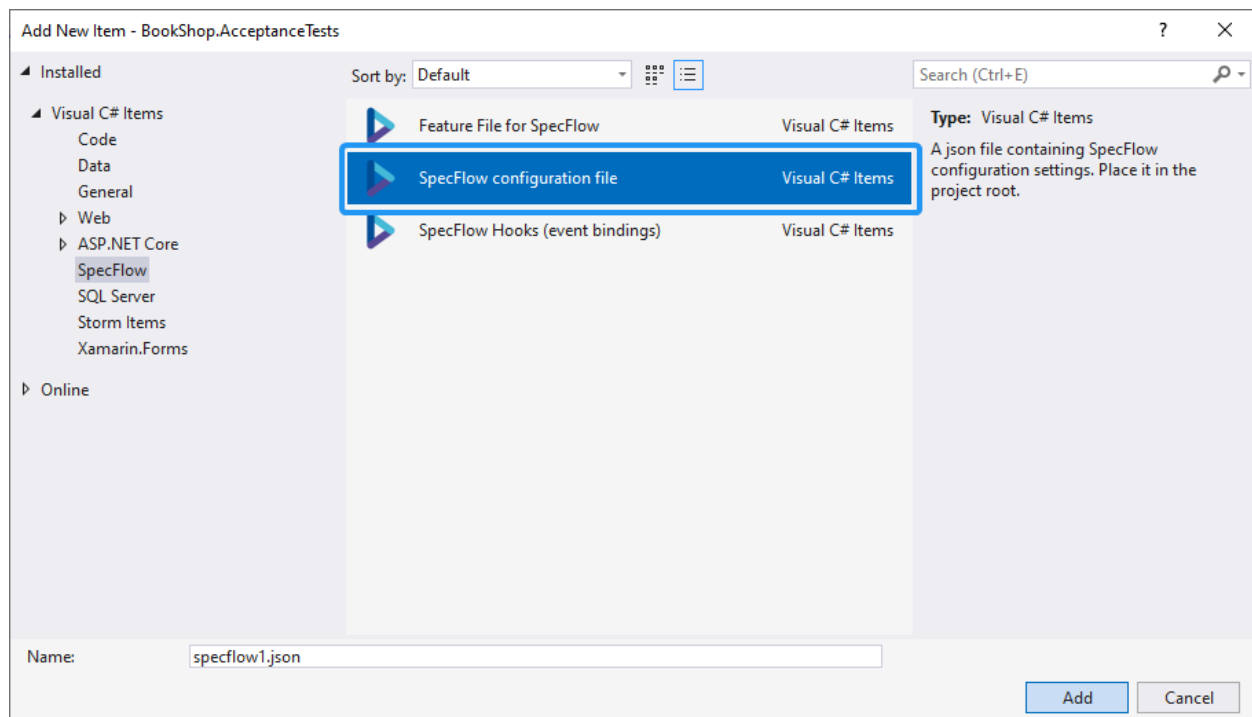
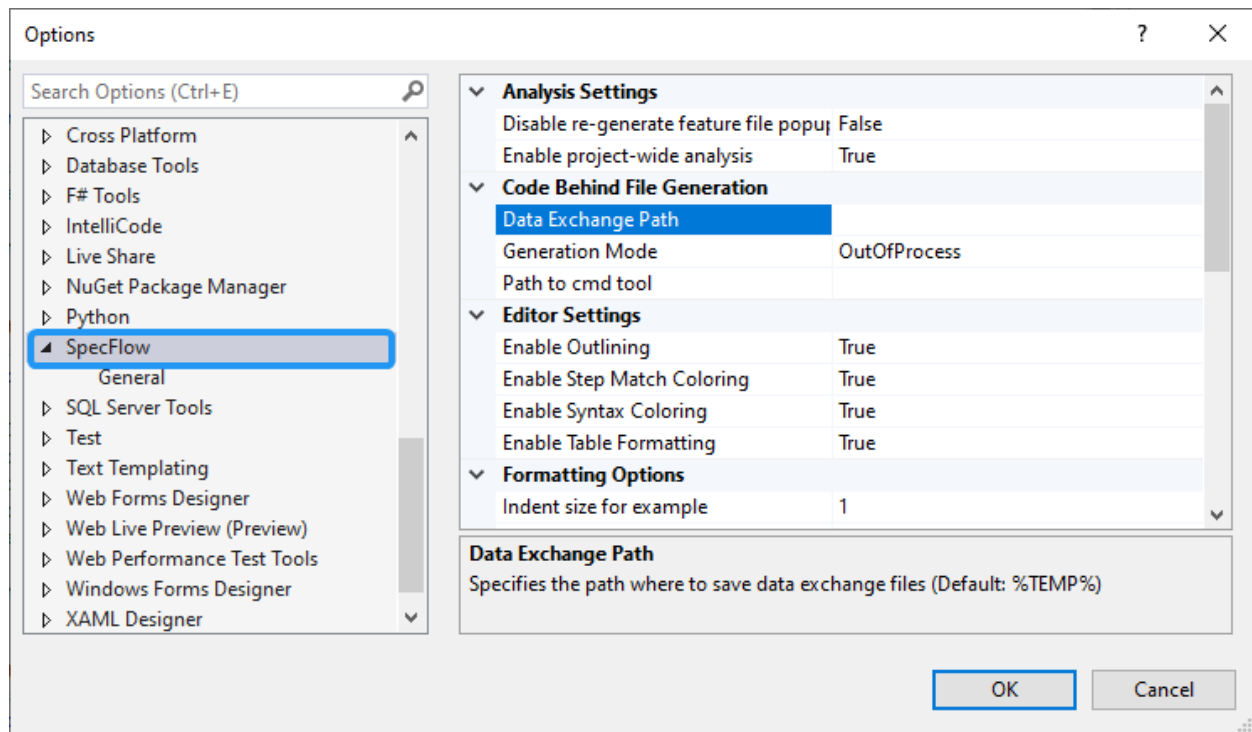
VS2022

Navigate to **Tools | Options | SpecFlow | General** | to access the extension settings.

You must edit the `specflow.json` config file to access the extension settings. If you don’t have the `specflow.json` file you can add it by right clicking on the SpecFlow project -> Add -> New item... -> Add SpecFlow configuration file.

The configuration file has a JSON [schema](#) , therefore you will see all available properties as you start typing.

Important: You must build your project for the changes in `specflow.json` to take effect.



2.51 Editing Features

The Visual Studio integration includes the following features to make it easier to edit feature files and identify which steps have already been bound.

2.51.1 Gherkin Syntax Highlighting

Various default styles have been defined for the Gherkin syntax. You can customize these colours in Visual Studio's settings (**Tools | Options | Environment | Fonts and Colors**). The names of the corresponding **Display items** in the list begin with "Gherkin".

In addition to highlighting keywords, comments, tags etc., unbound steps and parameters in feature files are highlighted when editing the file in Visual Studio. The following syntax highlighting is used by default:

- Purple: unbound steps
- Black: bound steps
- Grey italics: parameters in bound steps

You can thus tell immediately which steps in a feature file have been bound.

Note: In SpecFlow Visual Studio 2022 plugin a project must be **built** for syntax highlighting to update.

2.51.2 IntelliSense (auto-completion) for Keywords and Steps

IntelliSense makes SpecFlow easy to use when integrated with Visual Studio. IntelliSense uses find-as-you-type to restrict the list of suggested entries.

Gherkin Files

IntelliSense is available in feature files for the following:

- Gherkin keywords (e.g. Scenario, Given etc.)
- Existing steps are listed after a **Given**, **When** or **Then** statement, providing quick access to your current steps definitions. Bound steps are indicated with "->".

Note: Note that all the steps in all "*.feature" files that match the current type (Given, When, Then) are displayed

VS2019

VS2022

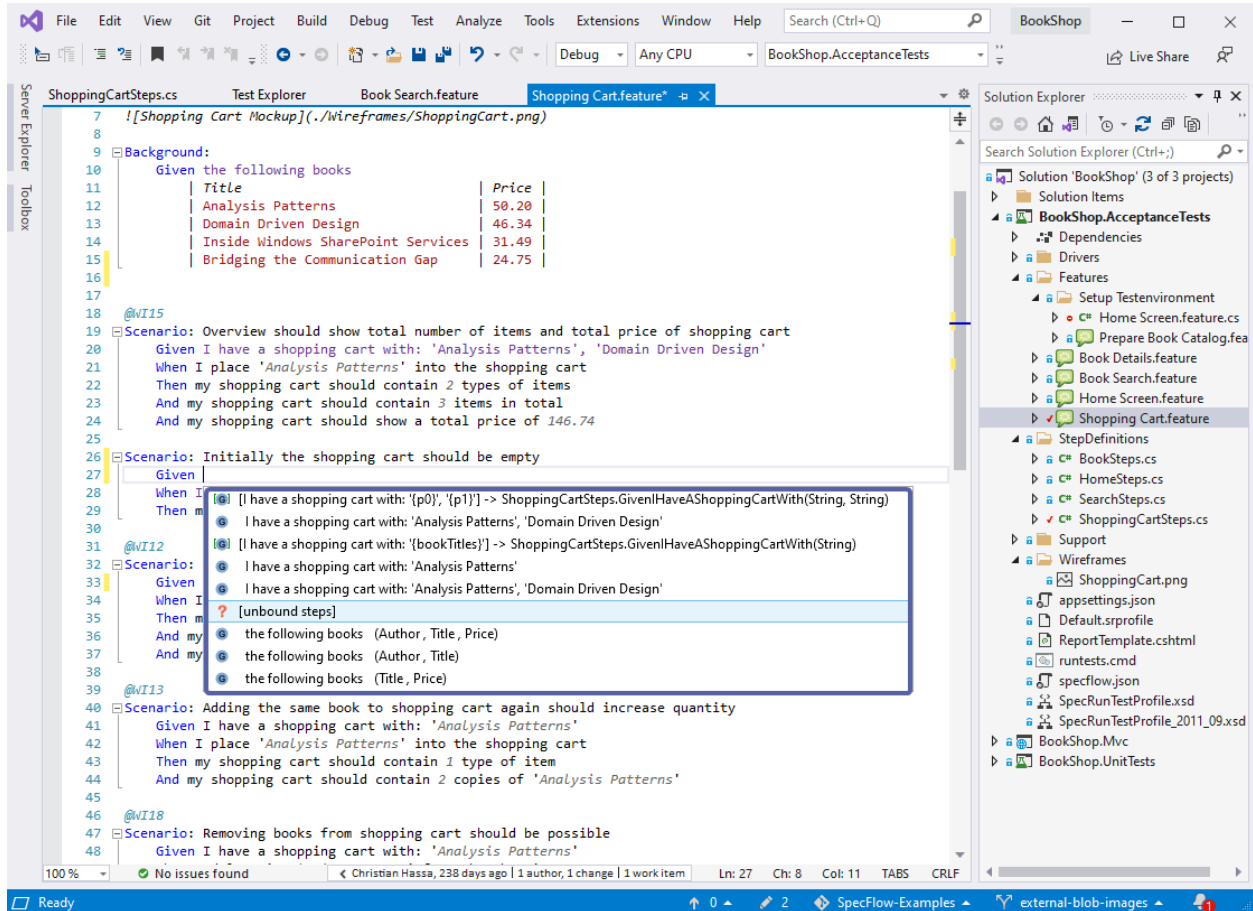
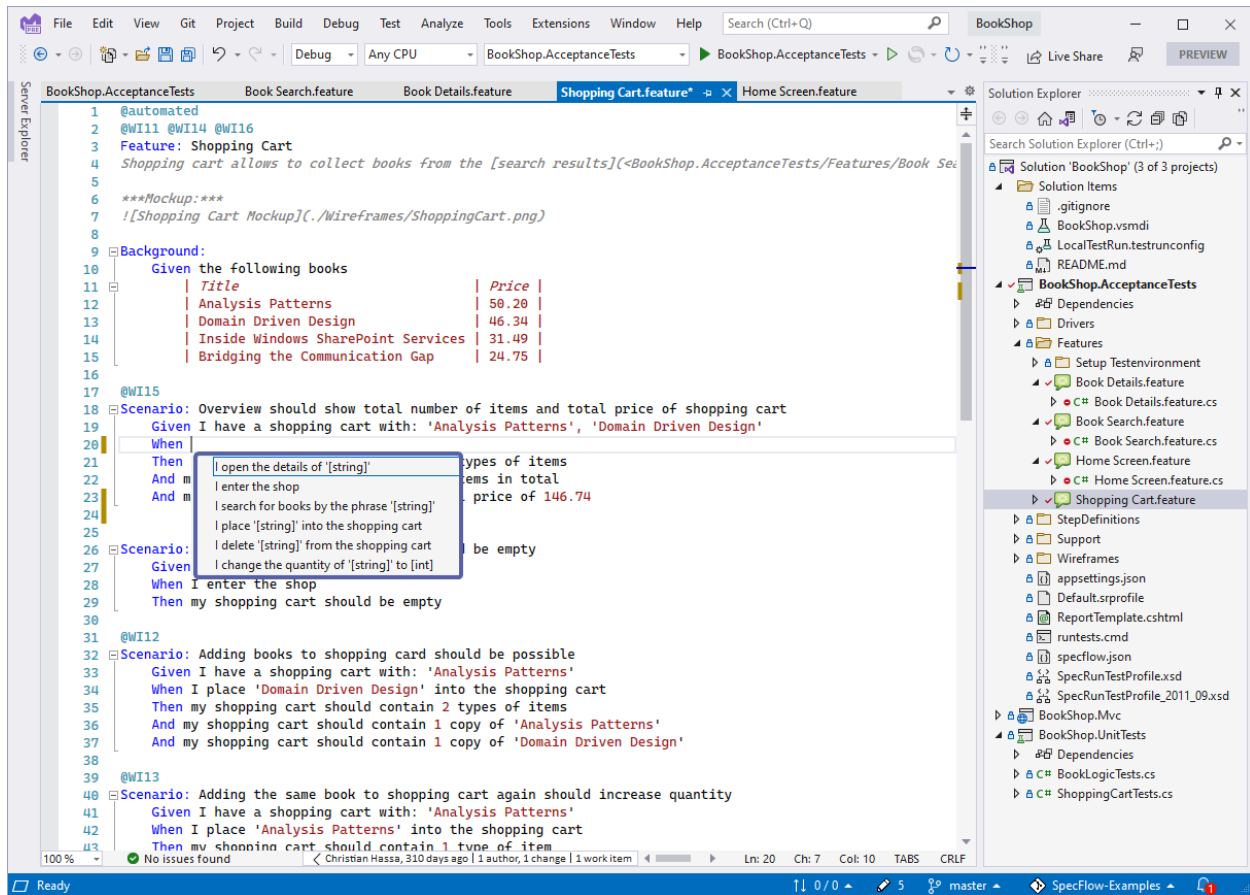


Fig. 1: The entries in the list that are marked with “->” indicate that the step is bound.



Code Files

IntelliSense is also available for the Gherkin keywords in your code files.

IntelliSense settings

As much as having a suggested list of previous entries could speed up your work, the list may become too long in a big project and not really usable. To fix this problem, SpecFlow gives you the option to limit the number of IntelliSense step instances suggestions for each step template.

To do this, simply navigate to **Tools | Options | SpecFlow | General | IntelliSense** and set the desired number of suggestions you wish to see against the **Max Step Instances Suggestions** value:

VS2019

VS2022

Navigate to **Tools | Options | SpecFlow | General | IntelliSense** and set the desired number of suggestions you wish to see against the **Max Step Instances Suggestions** value:

You must edit the `specflow.json` config file to turn inteli on or off and also access other settings as per below:

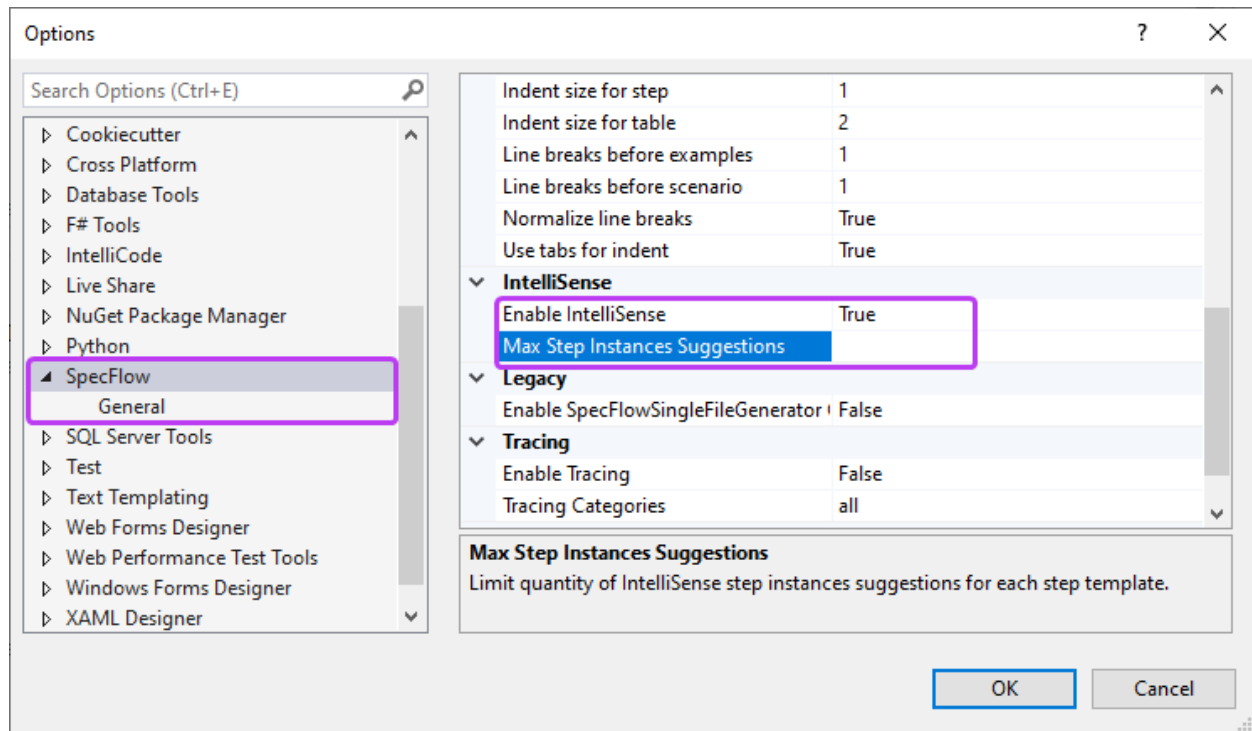
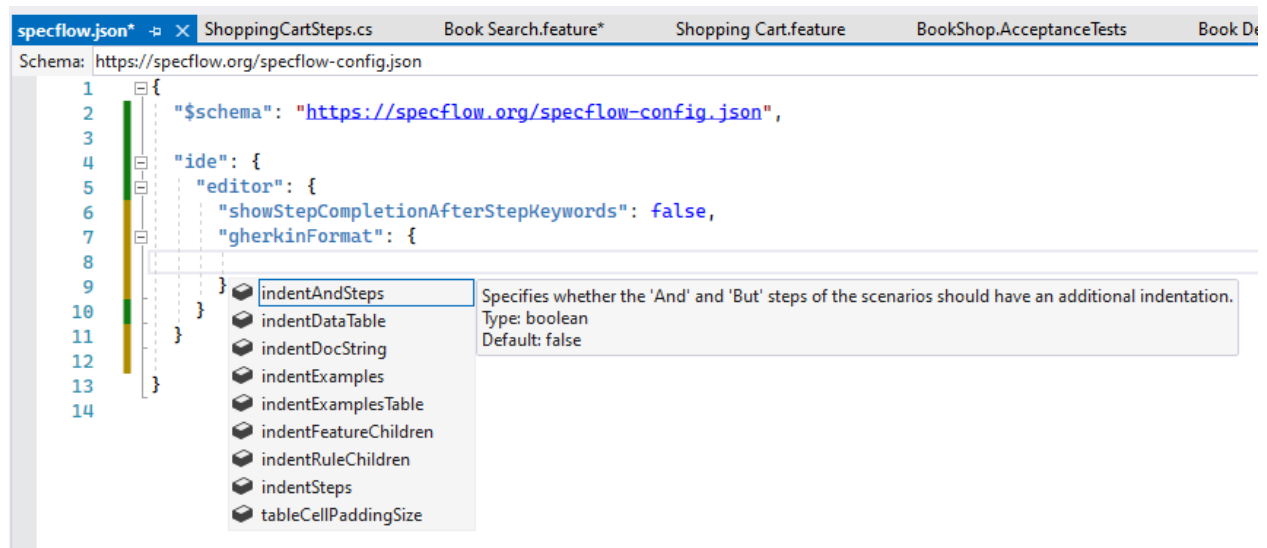


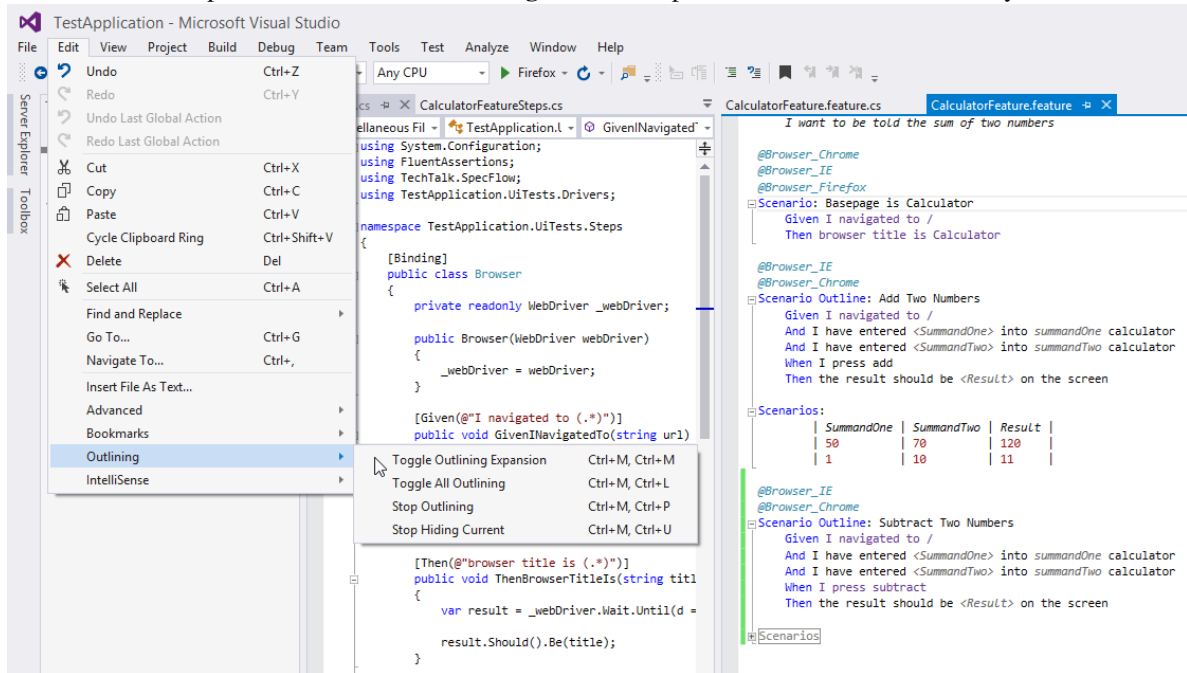
Fig. 2: > Note: Setting this value to 0 will only show the template.



2.51.3 Outlining and Comments in Feature Files

Most of the items in the **Edit** menu work well with SpecFlow feature files, for example:

- You can comment and uncomment selected lines ('#' character) with the default shortcut for comments (Ctrl+K Ctrl+C/Ctrl+K Ctrl+U) or from the menu
- You can use the options in the **Edit | Outlining** menu to expand and contract sections of your feature files



2.51.4 Table Formatting

Tables in SpecFlow are also expanded and formatted automatically as you enter column names and values:

Fig. 3: Formatted table

2.51.5 Document Formatting

Document formatting is also available. It automatically re-indents code and fixes blank lines, comments, etc.

You can find this option under **Edit->Advanced->Format document** or alternatively use the **Ctrl+K, Ctrl+D** shortcut:

Below is a feature file document which is not indented correctly:

After the **Format Document** command:

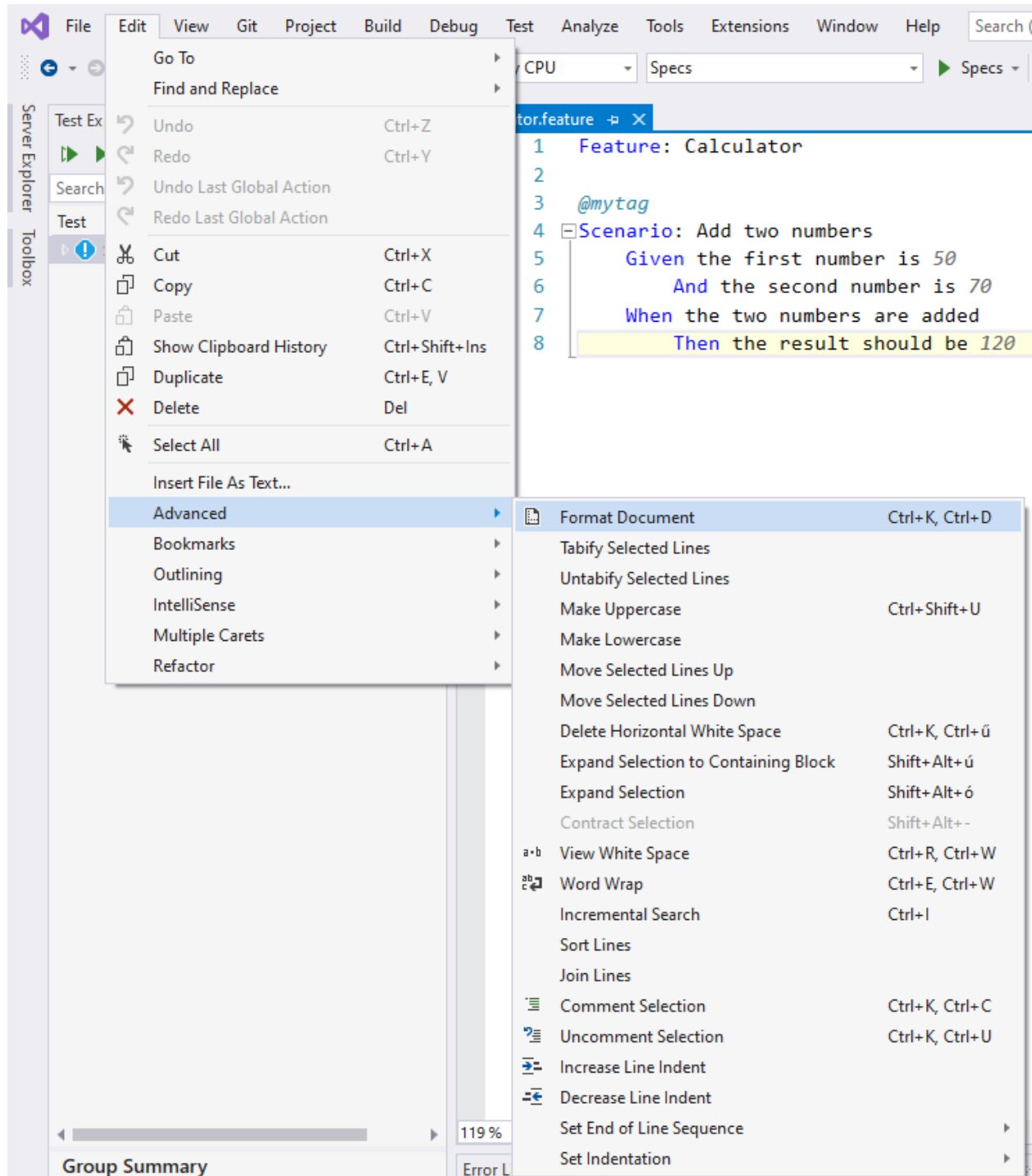
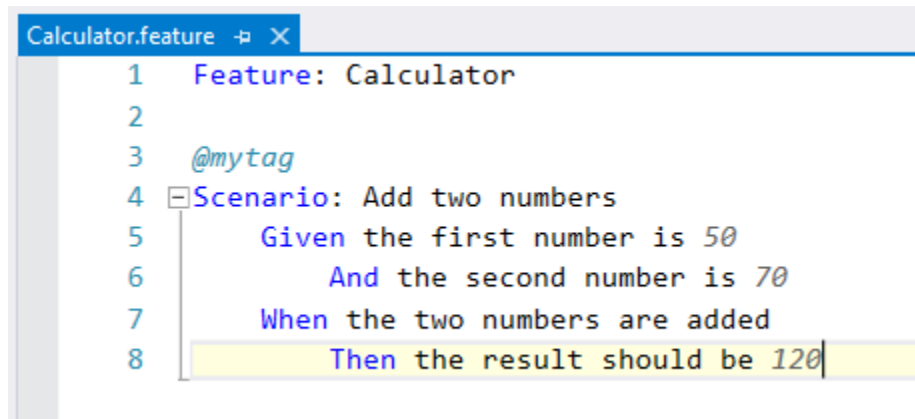


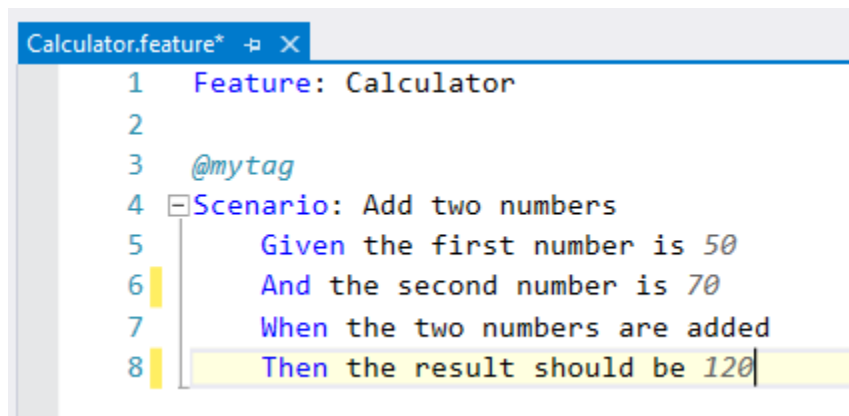
Fig. 4: Format document



The screenshot shows a code editor window titled "Calculator.feature" with a standard text editor interface. The content is a Gherkin document for a calculator feature. The text is not formatted, with no indentation or color-coding. The lines are numbered 1 through 8 on the left margin.

```
1 Feature: Calculator
2
3 @mytag
4 Scenario: Add two numbers
5     Given the first number is 50
6     And the second number is 70
7     When the two numbers are added
8     Then the result should be 120
```

Fig. 5: Unformatted document



The screenshot shows the same code editor window, but now the document is formatted. The text is color-coded: "Feature:" is purple, "@mytag" is green, "Scenario:" is blue, and the step keywords "Given", "And", "When", and "Then" are blue. The step descriptions are indented. The lines are numbered 1 through 8 on the left margin.

```
1 Feature: Calculator
2
3 @mytag
4 Scenario: Add two numbers
5     Given the first number is 50
6     And the second number is 70
7     When the two numbers are added
8     Then the result should be 120
```

Fig. 6: Formatted document

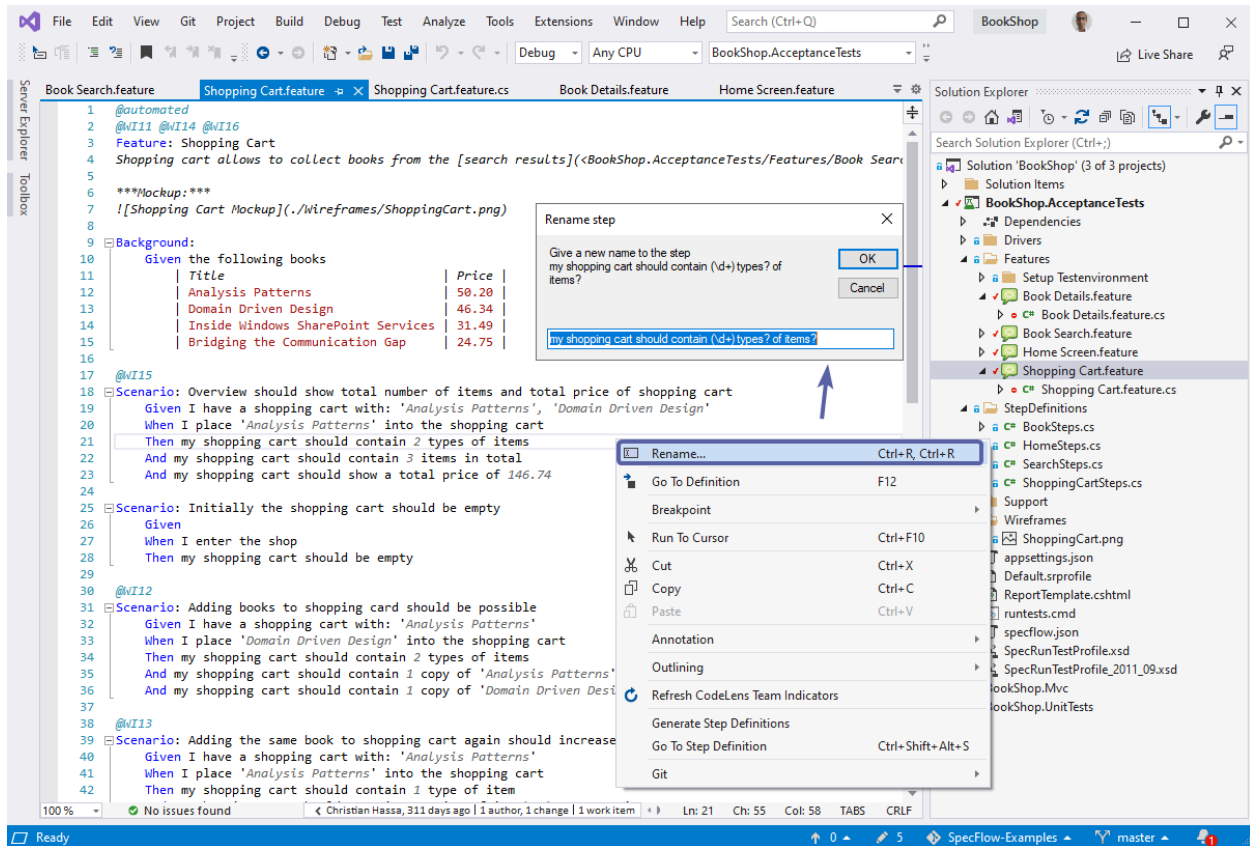
2.51.6 Renaming Steps

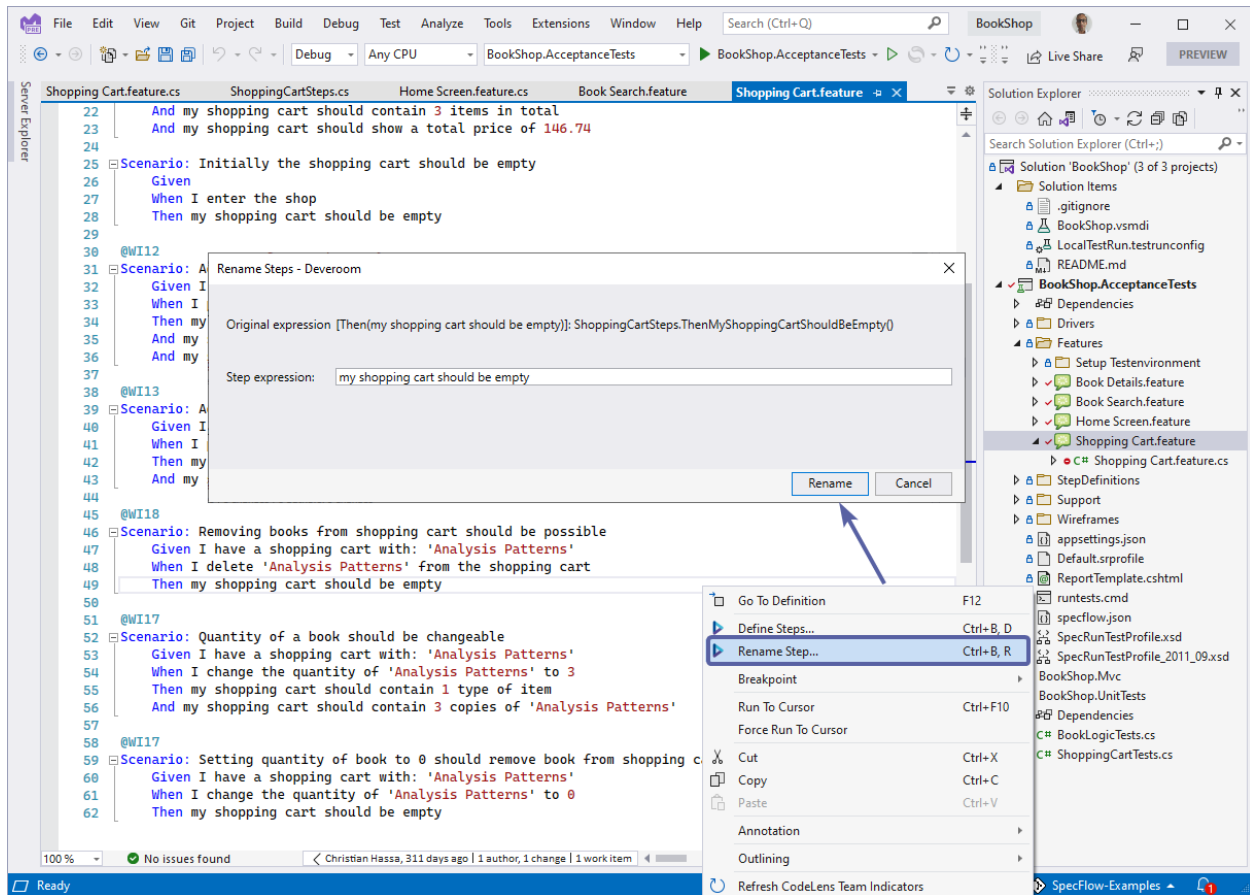
You can globally rename steps and update the associated bindings automatically. To do so:

1. Open the feature file containing the step.
2. Right-click on the step you want to rename and select Rename from the context menu.
3. Enter the new text for the step in the dialog and confirm with OK.
4. Your bindings and all feature files containing the step are updated.

VS2019

VS2022





Note: If the rename function is not affecting your feature files, you may need to restart Visual Studio to flush the cache.

2.52 Navigation Features

You can navigate between the methods in your bindings and the associated steps in your Gherkin feature files.

2.52.1 Navigating from a Binding to Steps in Gherkin Files

You can navigate from a step definition method to the matching step(s) in your Gherkin feature file(s):

VS2019

VS2022

Right click the binding and select *'Go to SpecFlow step definition usages'*

Right click the binding and select *'Find step definition usages'*

2.52.2 Navigating from a Gherkin Step to a Binding

To navigate from a step in a Gherkin feature file to the corresponding step definition method:

1. Place your cursor on the step in your feature file.
2. Right-click and select **Go To Step Definition** from the menu (F12).
3. The file containing the binding is opened at the appropriate step definition method.

2.53 Test Explorer Support

The Visual Studio integration supports executing SpecFlow scenarios from the Visual Studio Test Explorer. The basic Test Explorer features work with all unit test providers, although you may need to install additional Visual Studio connectors, depending on the unit test framework. Full integration is provided for [SpecFlow+ Runner](#), meaning you can run and debug your scenarios as first class citizens:

- View scenarios listed by title in the Test Explorer
- Group scenarios in the Test Explorer by tag (choose “Traits” grouping) or feature (choose “Class”)
- Filter scenarios by various criteria using the search field
- Run/debug selected or all scenarios
- Double-click an entry in the list to switch to the scenario in the feature file
- View test execution results

2.54 Generating Skeleton Code

You can automatically create a suitable class with skeleton bindings and methods in Visual Studio:

VS2019

VS2022

- Open your feature file.
- Right-click in the editor and select **Generate Step Definitions** from the menu.
- A dialog is displayed with a list of the steps in your feature file. Use the check boxes to determine which steps to generate skeleton code for.
- Enter a name for your class in the **Class name** field.
- Choose your desired [Step Definition Style](#), which include formats without regular expressions. Click on **Preview** to preview the output.
- Click on **Generate** to add a new .cs file with your class to your project. This file will contain the skeleton code for your class and the selected steps.

Tip: If your feature file already contains bound steps, i.e the binding file already exists and you like to add a new one, click on **Copy methods to clipboard** to copy the generated skeleton code to the clipboard. You can then paste it to the file of your choosing.

The most common parameter usage patterns (quotes, apostrophes, numbers) are detected automatically when creating the code and are used by SpecFlow to generate methods and regular expressions. For more information on the available options and custom templates, refer to the [Step Definition Style](#) page.

- Open your feature file.
- Right-click in the editor and select **Define steps...** from the menu.
- Enter a name for your class in the **Class name** field.
- Click on **create** to generate a new step definition file or use the **Copy methods to clipboard** method to paste the skeleton code in an existing step definition file.

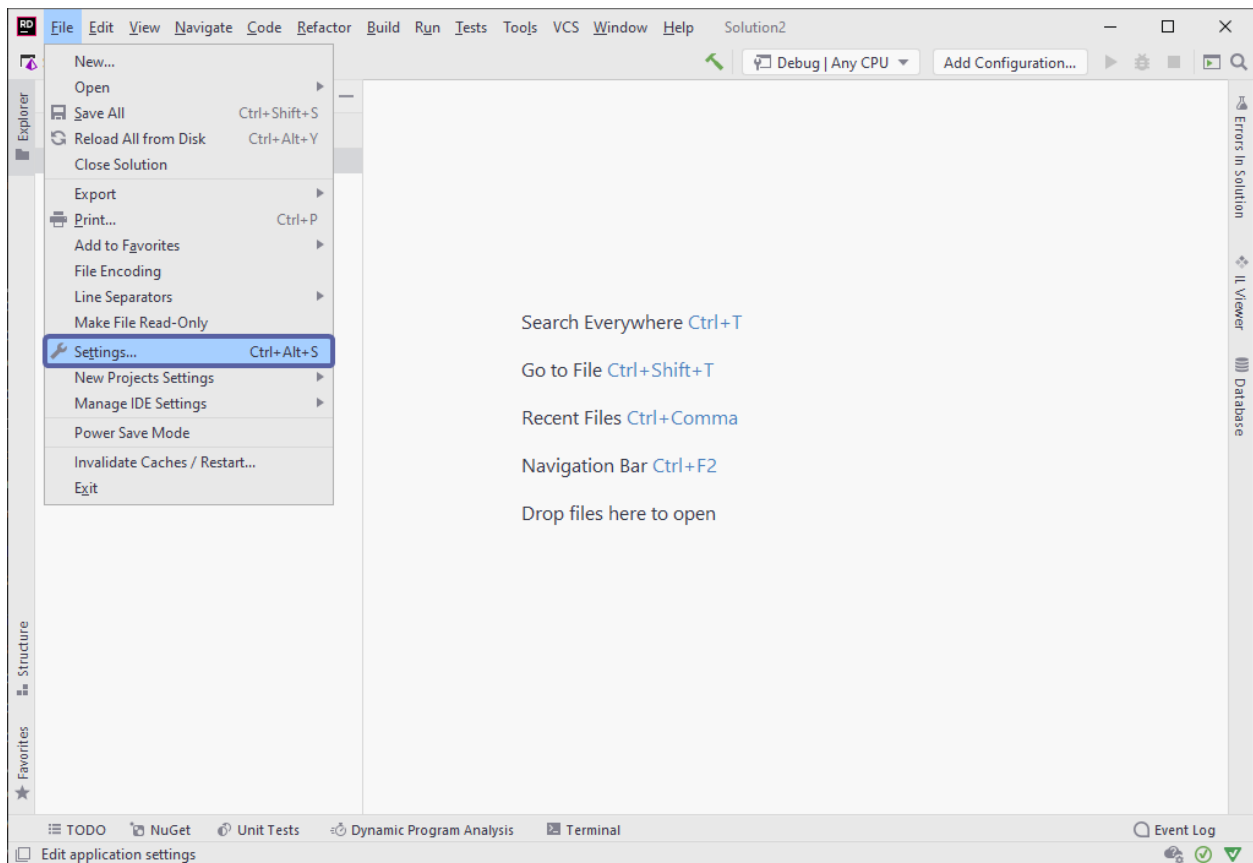
2.55 Installation

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

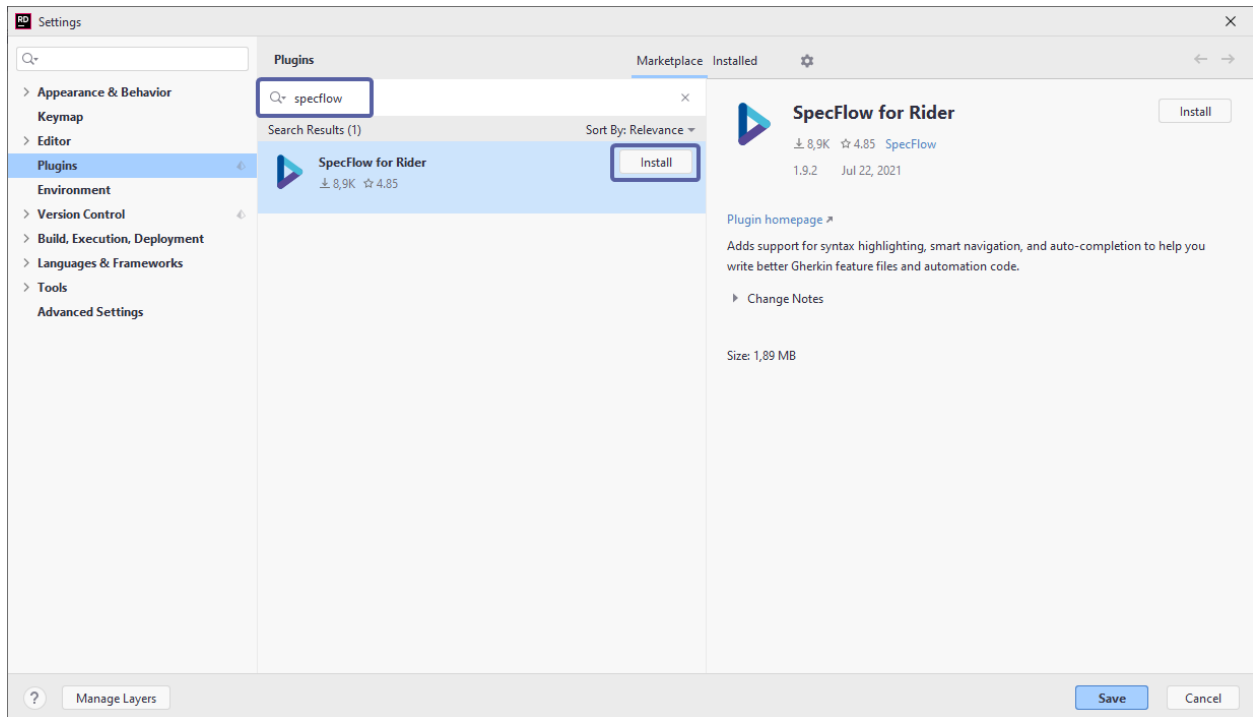
To install the plugin directly from JetBrains Rider:

1- Open JetBrains Rider

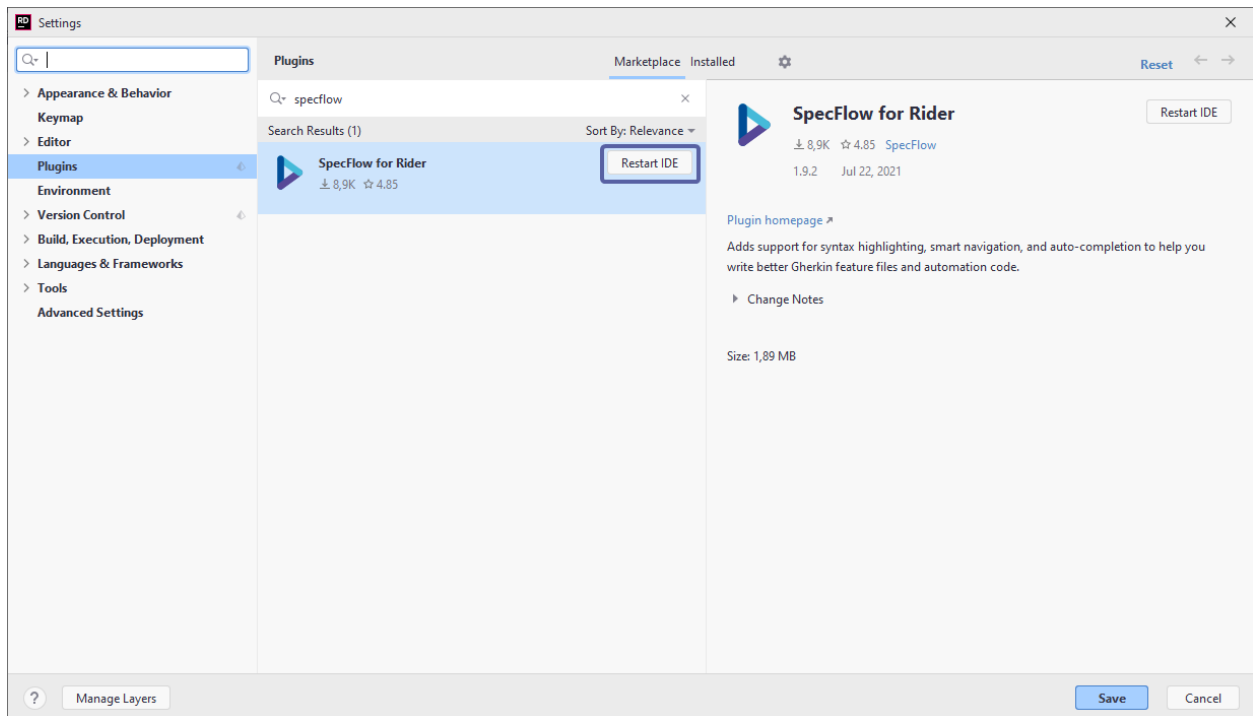
2- Navigate to **File Settings Plugins** (**Ctrl+Alt+S**) and search for “SpecFlow” in the search bar:



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**:



The installation is now finished. Check out the [SpecFlow for Rider features](#) to see a list of all the handy features in this plugin.

2.56 Features

The SpecFlow for Rider plugin is bundled with a handful of features, see below for more details.

2.56.1 Navigation

The SpecFlow for Rider plugin allows you to quickly navigate between a Gherkin Step, its relevant bindings, and vice versa.

There are multiple ways to achieve this:

>Note The keyboard shortcuts documented below may vary depending on how they were setup during the Rider installation. This documentation is based off the Visual Studio shortcut schema. More info [here](#).

1- The quickest way to navigate from a Gherkin step to its relevant binding is to use keyboard shortcuts (**Ctrl + F12**). Alternatively, you can right click on a Gherkin step and navigate to **Go to Implementations** :

2- To navigate from a binding to its corresponding Gherkin step, the keyboard shortcuts (**Shift + F12**) can be used :

3- You can also see all Gherkin steps currently using a particular binding by pressing **F12** or by navigating to it **Go to Go to Declarations or Usages** :

2.56.2 Creating Steps

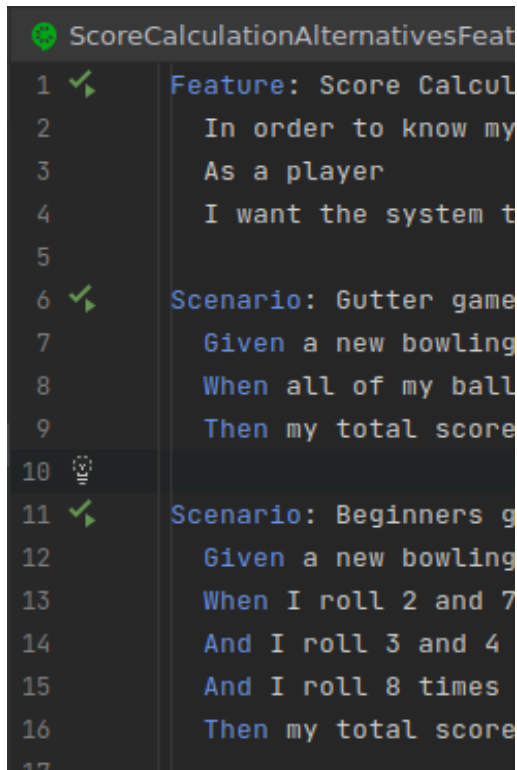
The plugin allows you to quickly create a step and also highlights when a step is missing. To do this, click on an unbound step, click on view action list, and then click on *Create step*:

Create a New Binding Class

You also have the option to create a new binding class straight from the *Create step* menu. To do this, click on an unbound step, click on view action list, and then click on *Create step* and then navigate to *Create new binding class*:

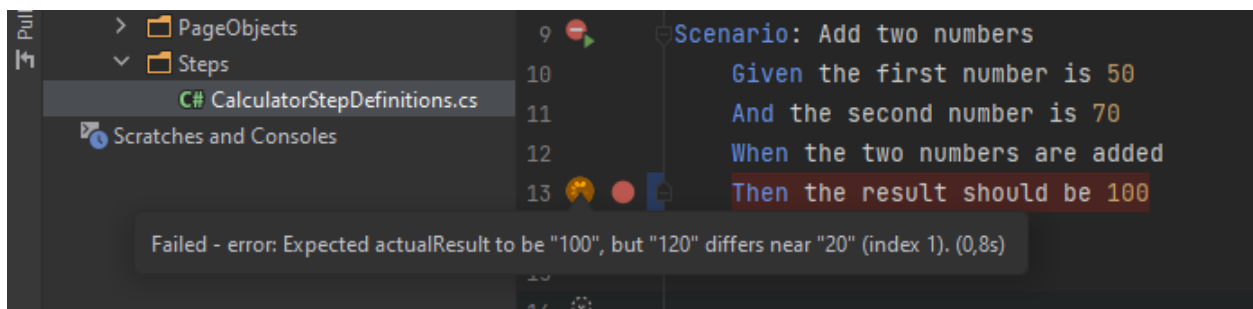
2.56.3 Test Results

The SpecFlow plugin also displays the test results for a specific Gherkin step. You can see whether a test has passed or failed and also execute it by pressing the button:



Failed Tests Results

If a test fails, the plugin shows you the exact step at which the step failed by highlighting it and also conveniently displays the error message at feature file level. You can view this by hovering over the SpecFlow logo on the failed step:



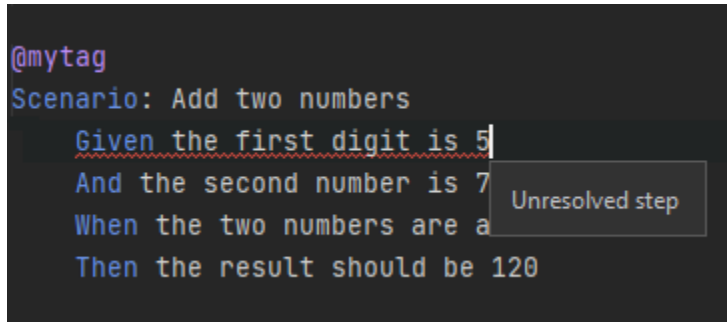
2.56.4 Auto Renaming

The plugin also automatically detects any mismatch in step definition names and displays suggestions to match it to its pattern:

2.56.5 Syntax Highlighting

Syntax highlighting helps you quickly identify unbound steps in your feature files by underlining them:

```
@mytag
Scenario: Add two numbers
    Given the first digit is 5
    And the second number is 7
    When the two numbers are a
    Then the result should be 120
```



The absence of a red underline indicates that the step is bound. Notice the parameters in bound steps are also colored differently, in this example light grey.

```
@mytag
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
```

> Note: Syntax highlighting colors may vary depending on your Rider theme.

2.56.6 Table Formatting

Tables are expanded and formatted automatically in feature files. You can use the keyboard shortcuts (**Ctrl + Alt + Enter**) or go to **Code Reformat Code** to apply correct formatting on the entire feature file.

Please note we are always working to improve and introduce new features to make the plugin more versatile and easy to use.

2.57 Installation

Although there is **no official** SpecFlow for Visual Studio Code extension available right now, we have put together this guide to help you have an experience as close as possible to a SpecFlow Extension.

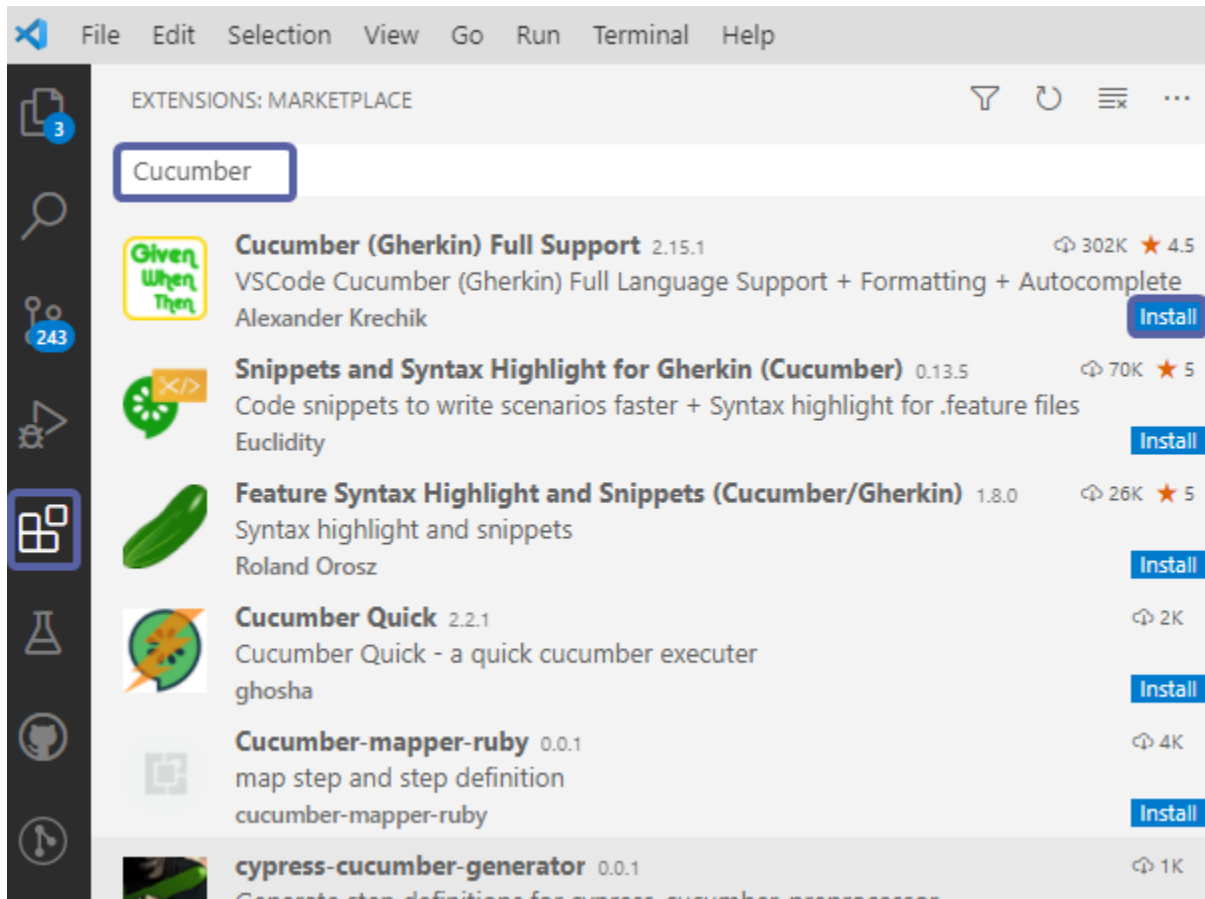
If you like to see an official SpecFlow extension for VS Code please vote for it on our community feature request page [here](#). You can also vote on other existing requests by other community members.

Note we do have an official extension for Visual Studio, more details [here](#)

We recommend installing the [Cucumber \(Gherkin\) Full Support](#) extension for Visual Studio Code to enable key features such as syntax highlighting, Auto-completion of steps, and table formatting.

You can install this extension through the [Visual Studio Marketplace](#) or through VS code IDE itself:

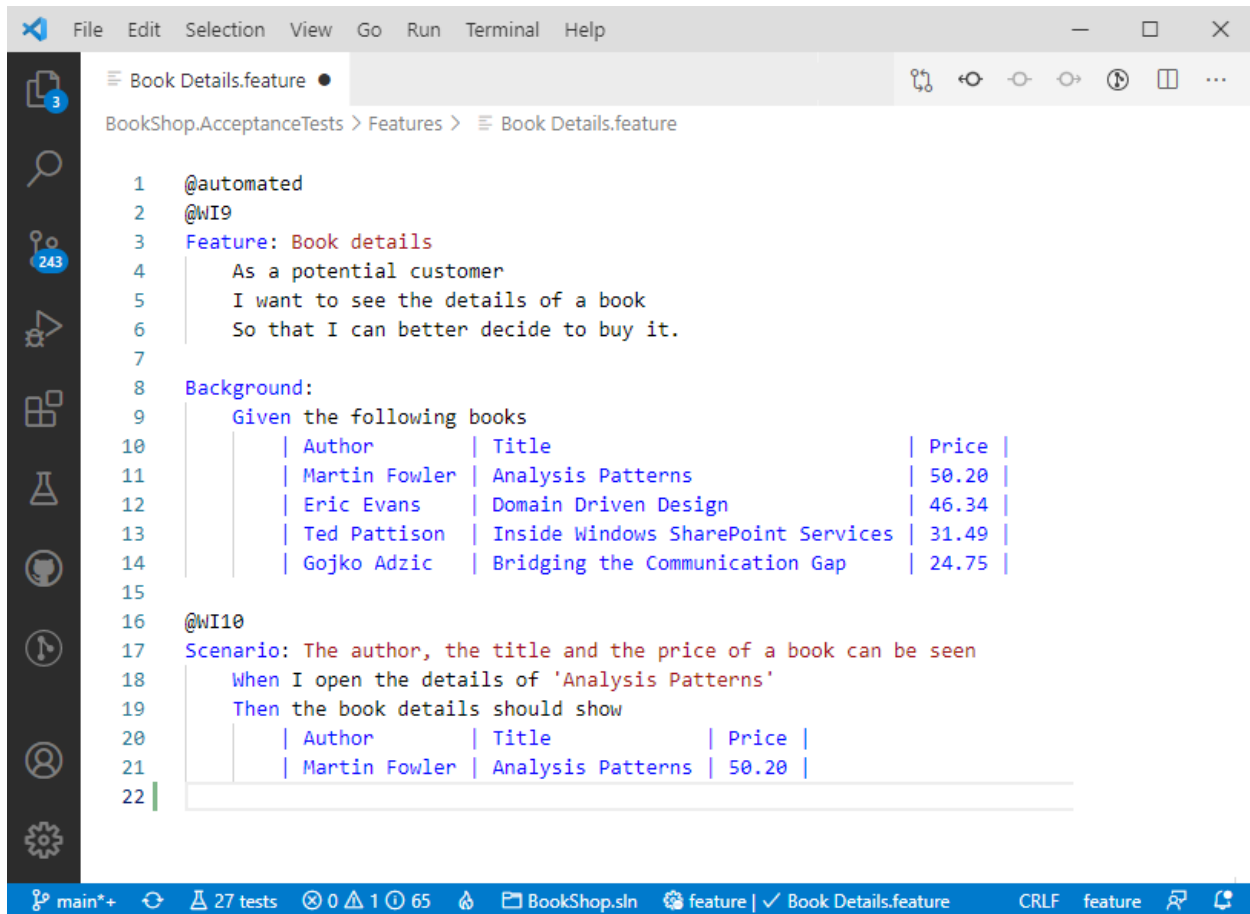
1- Open VS Code and navigate to **Extensions** and search for **Cucumber** in the search box and hit **Install**:



The extension enables the following features:

- Syntax highlight
- Basic Snippets support
- Auto-parsing of feature steps from paths, provided in settings.json
- Autocompletion of steps
- Definitions support for all the steps parts
- Document format support, including tables formatting
- Supporting of many spoken languages
- Gherkin page objects native support
- Multiple programming languages, C#, JS, TS, Ruby, Kotlin etc.

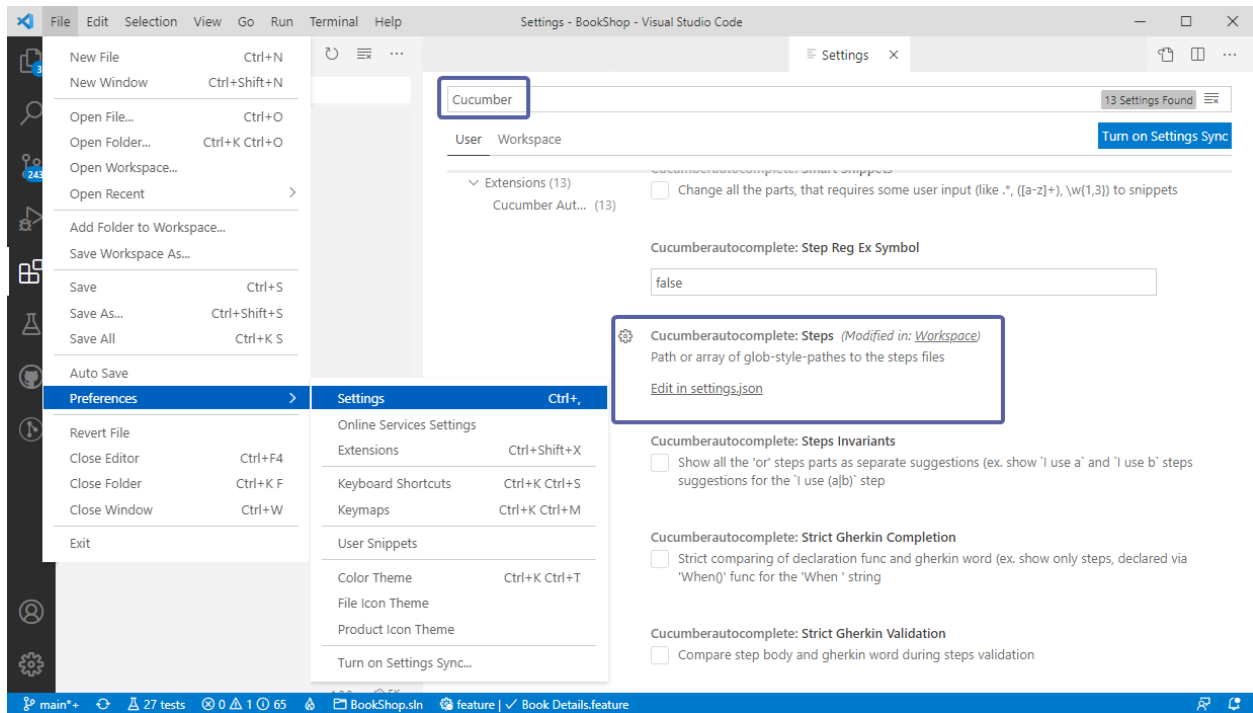
These features will nicely format and highlight your Gherkin files:



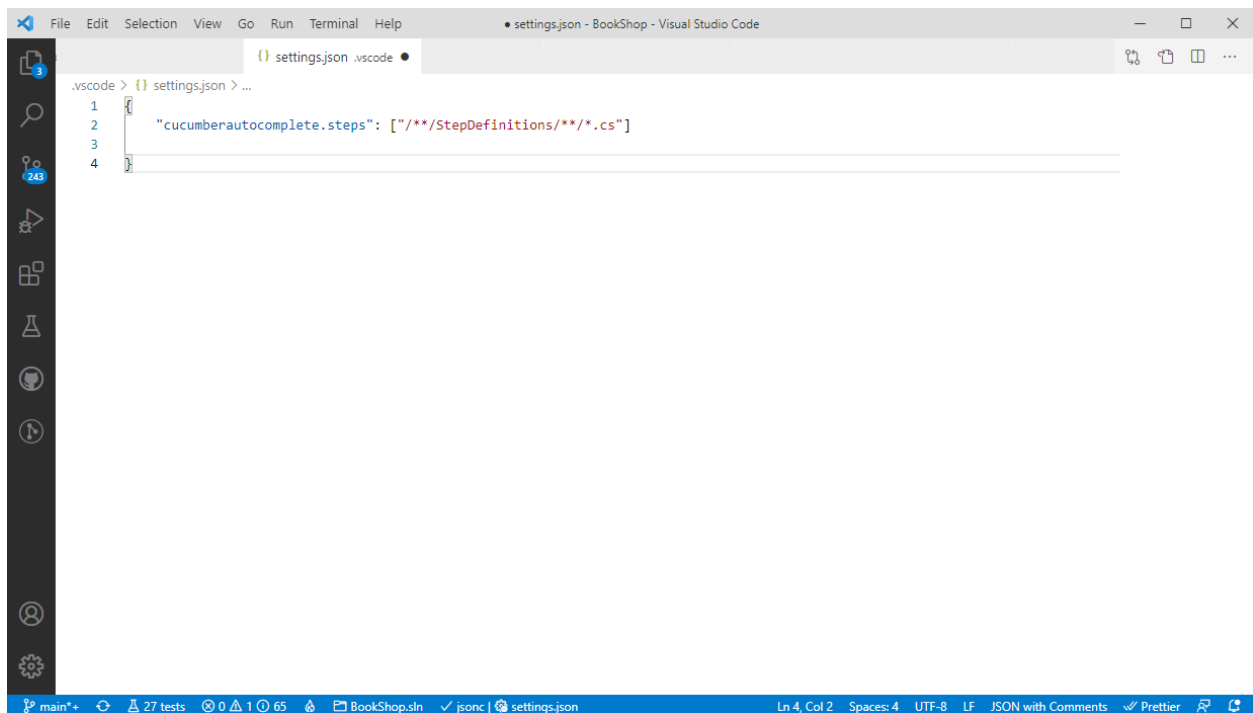
```
1 @automated
2 @WI9
3 Feature: Book details
4     As a potential customer
5     I want to see the details of a book
6     So that I can better decide to buy it.
7
8 Background:
9     Given the following books
10         | Author | Title | Price |
11         | Martin Fowler | Analysis Patterns | 50.20 |
12         | Eric Evans | Domain Driven Design | 46.34 |
13         | Ted Pattison | Inside Windows SharePoint Services | 31.49 |
14         | Gojko Adzic | Bridging the Communication Gap | 24.75 |
15
16 @WI10
17 Scenario: The author, the title and the price of a book can be seen
18     When I open the details of 'Analysis Patterns'
19     Then the book details should show
20         | Author | Title | Price |
21         | Martin Fowler | Analysis Patterns | 50.20 |
22
```

There is an important and handy feature missing here which is the navigation between feature files and their respective step definitions (bindings). Fortunately, we can enable this with a simple tweak in the extension's settings:

- 2- Navigate to **File Preferences Settings** and look for **Cucumber** in the settings search box
- 3- Click on **Edit in setting.json** under the **Steps** settings:



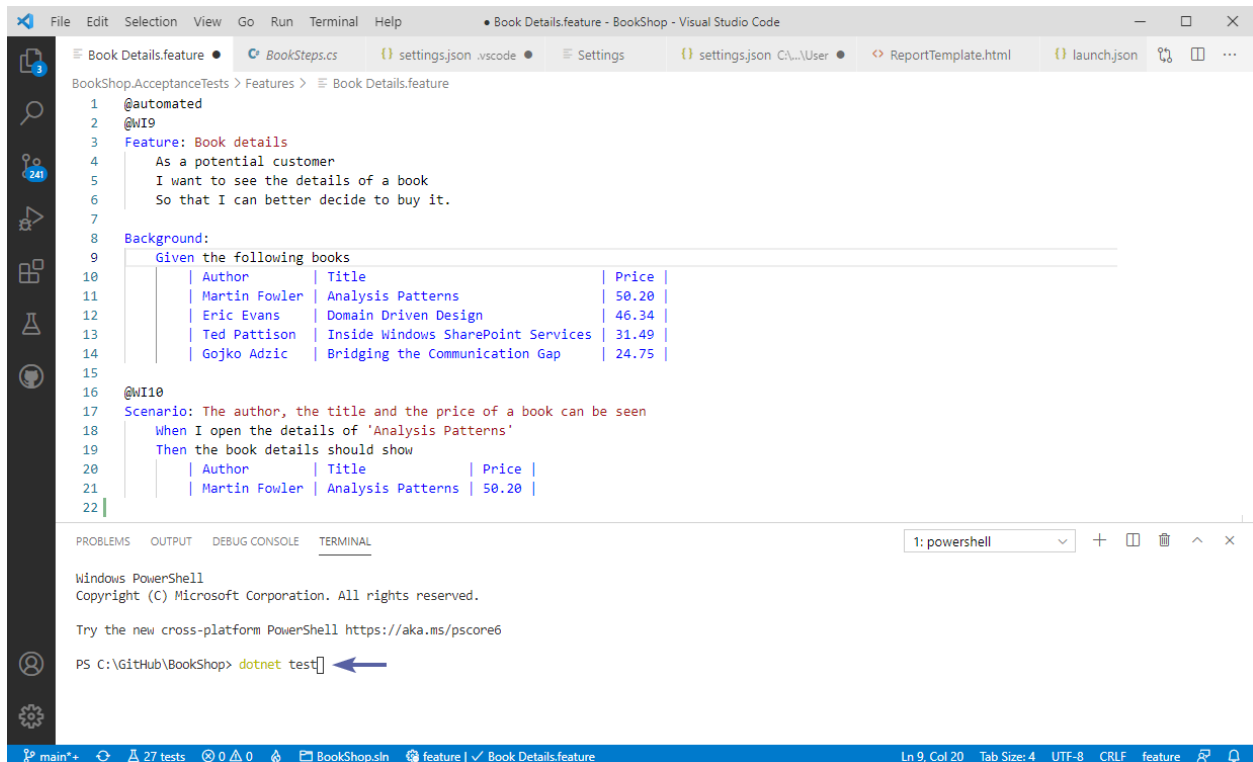
As the settings describes, you have to insert the path or array of glob-style-paths to the step definition files in your project. If you have been using a SpecFlow template project your step definition files would be in a folder called *StepDefinitions* so you would enter the path as per below:



4- Save your settings and go back to your feature files. You will now be able to navigate from your feature files steps to their corresponding bindings by pressing **F12** or by holding **Ctrl** and clicking on a step:

2.58 Test Execution

The best way to execute your tests is to use the VS Code terminal, simply open the terminal type in `dotnet test` and hit enter:



The screenshot shows the Visual Studio Code interface. The editor displays a file named `BookDetails.feature` with the following content:

```
1 @automated
2 @W10
3 Feature: Book details
4   As a potential customer
5     I want to see the details of a book
6     So that I can better decide to buy it.
7
8 Background:
9   Given the following books
10    | Author | Title | Price |
11    | Martin Fowler | Analysis Patterns | 50.20 |
12    | Eric Evans | Domain Driven Design | 46.34 |
13    | Ted Pattison | Inside Windows SharePoint Services | 31.49 |
14    | Gojko Adzic | Bridging the Communication Gap | 24.75 |
15
16 @W110
17 Scenario: The author, the title and the price of a book can be seen
18   When I open the details of 'Analysis Patterns'
19   Then the book details should show
20    | Author | Title | Price |
21    | Martin Fowler | Analysis Patterns | 50.20 |
22
```

The terminal window at the bottom shows the command `dotnet test` being executed. The terminal output includes the Windows PowerShell prompt and the command execution:

```
PS C:\GitHub\BookShop> dotnet test
```

You can then view the attachments, log file, and an overall results:

The screenshot shows the Visual Studio Code interface with a SpecFlow feature file open. The feature file contains a scenario with a data table. The terminal window shows the execution results, including a summary of failed, passed, and skipped tests, and a list of attachments.

```

1 @automated
2 @WI9
3 Feature: Book details
4   As a potential customer
5   I want to see the details of a book
6   So that I can better decide to buy it.
7
8 Background:
9   Given the following books
10  | Author | Title | Price |
11  | Martin Fowler | Analysis Patterns | 50.20 |
12  | Eric Evans | Domain Driven Design | 46.34 |
13  | Ted Pattison | Inside Windows SharePoint Services | 31.49 |
14  | Gojko Adzic | Bridging the Communication Gap | 24.75 |
15
16 @WI10
17 Scenario: The author, the title and the price of a book can be seen
18   When I open the details of 'Analysis Patterns'
19   Then the book details should show
20   | Author | Title | Price |
21   | Martin Fowler | Analysis Patterns | 50.20 |
22
  
```

Terminal Output:

```

1: powershell
at TechTalk.SpecRun.Framework.TaskExecutors.StaticOrInstanceMethodExecutor.ExecuteInternal(ITestThreadExecutionContext testThreadExecutionContext)
at TechTalk.SpecRun.Framework.TaskExecutors.StaticOrInstanceMethodExecutor.Execute(ITestThreadExecutionContext testThreadExecutionContext)
at TechTalk.SpecRun.Framework.TestNodeTask.Execute()
Skipped Calculating the number of books on a bookshelf [184 ms]

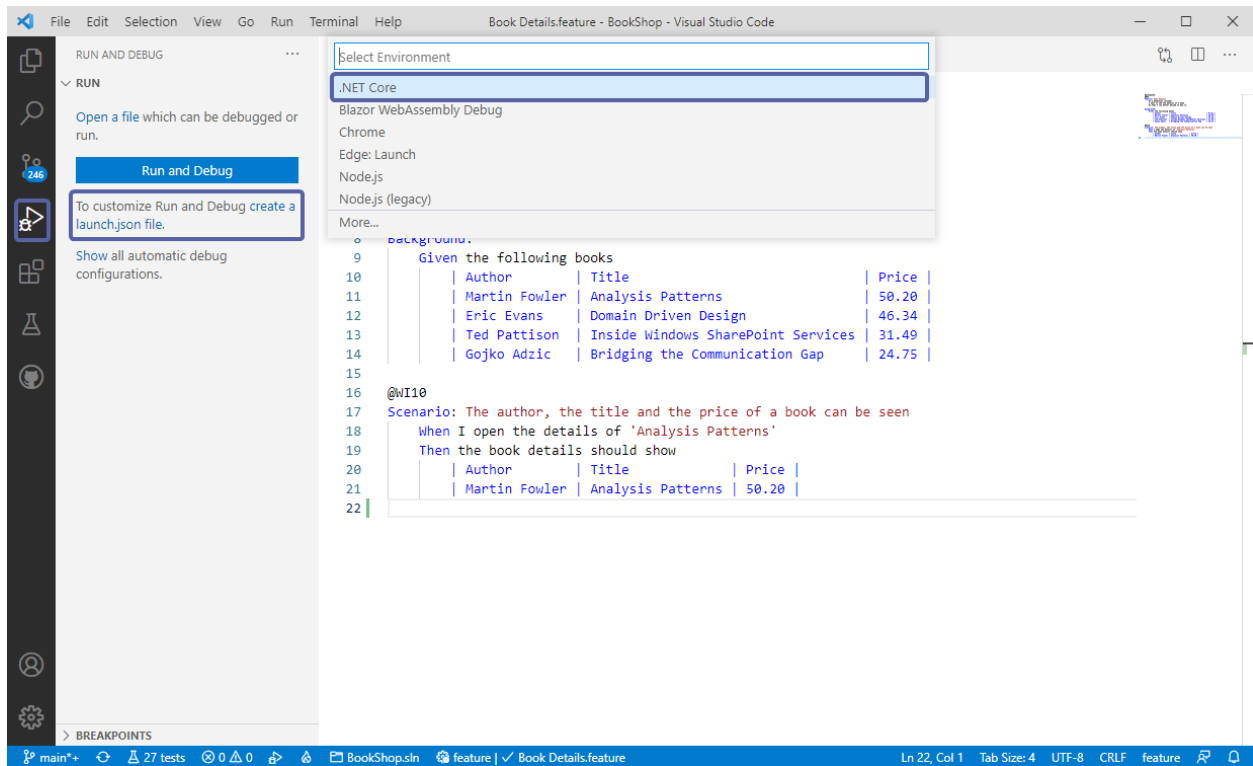
Attachments:
C:\Users\alimo\AppData\Local\Temp\tjl5yxor.ecg\BookShop.AcceptanceTests_BookShop.AcceptanceTests_2021-03-17T153418.log
C:\Users\alimo\AppData\Local\Temp\tjl5yxor.ecg\ReportTemplate.html
Failed! - Failed: 12, Passed: 6, Skipped: 1, Total: 19, Duration: 4 s - BookShop.AcceptanceTests.dll (netcoreapp3.1)
  
```

> **Note:** The attachments above would vary depending on the runner you are using. In our example we are using the *SpecFlow+Runner*.

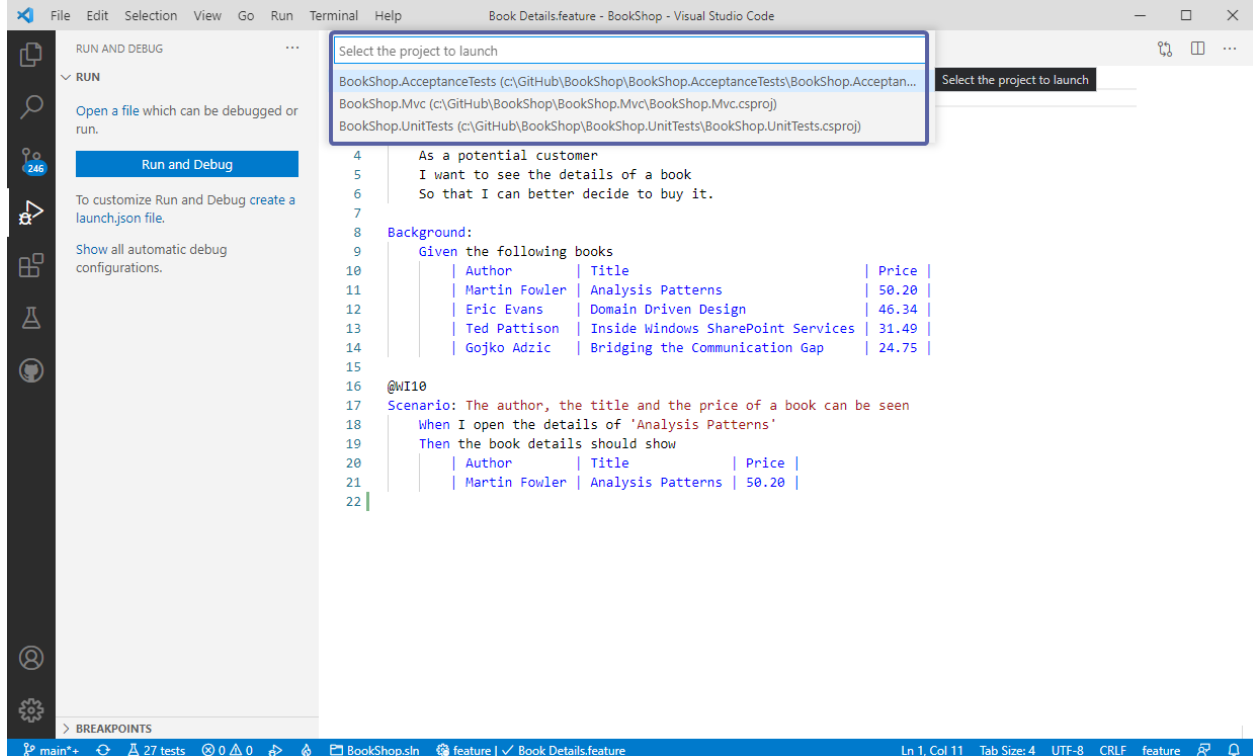
2.59 Debugging

Follow the below steps to setup your VS Code environment for debugging:

- 1- Click on the **Run and Debug** (Ctrl + Shift + D) button on the left pane and then click on **Create a launch.json file**. VS Code will then ask you what kind of environment you want to debug, select **.NET Core**:

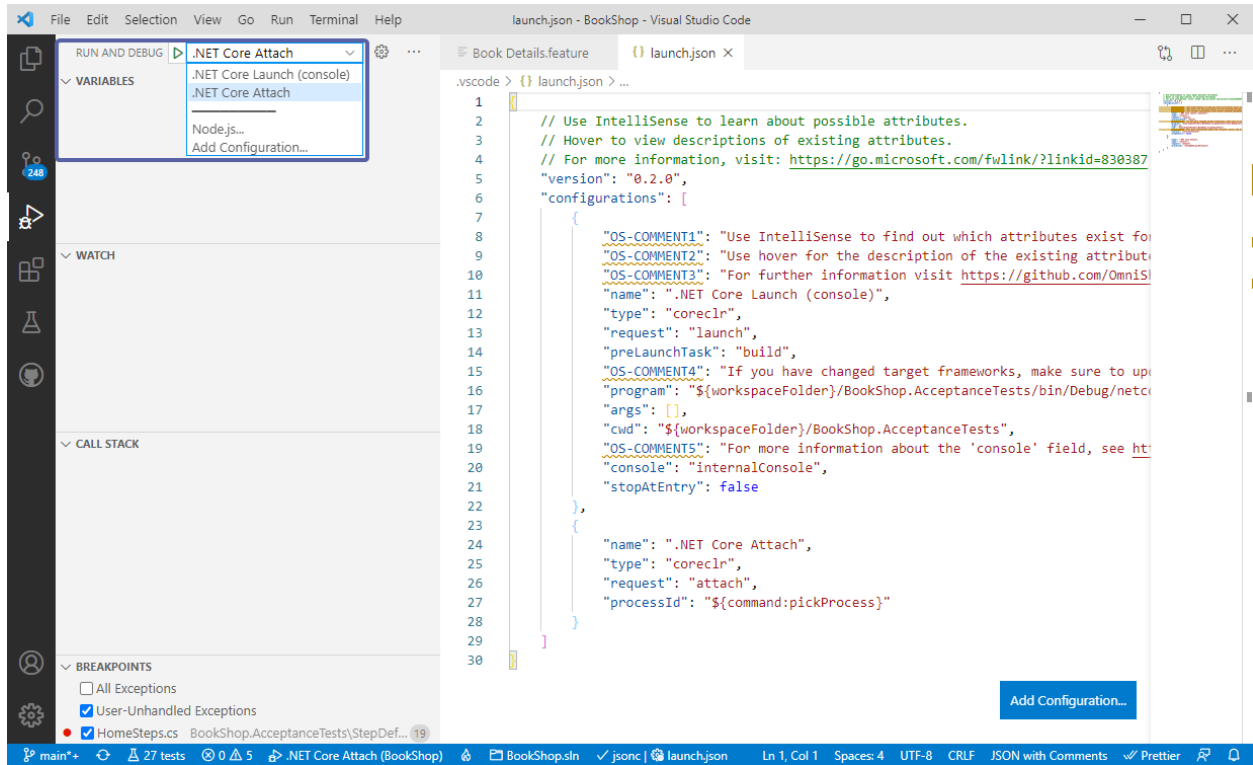


Next, you will have to choose the project you want to debug. Select the project and VS Code will then create a **launch.json**



2- Once the **launch.json** file is created by VS Code you will see the debug options on the left pane. There are two debug operations to choose from here, **.NET Core Launch (Consohle)** which is for console applications and **.NET**

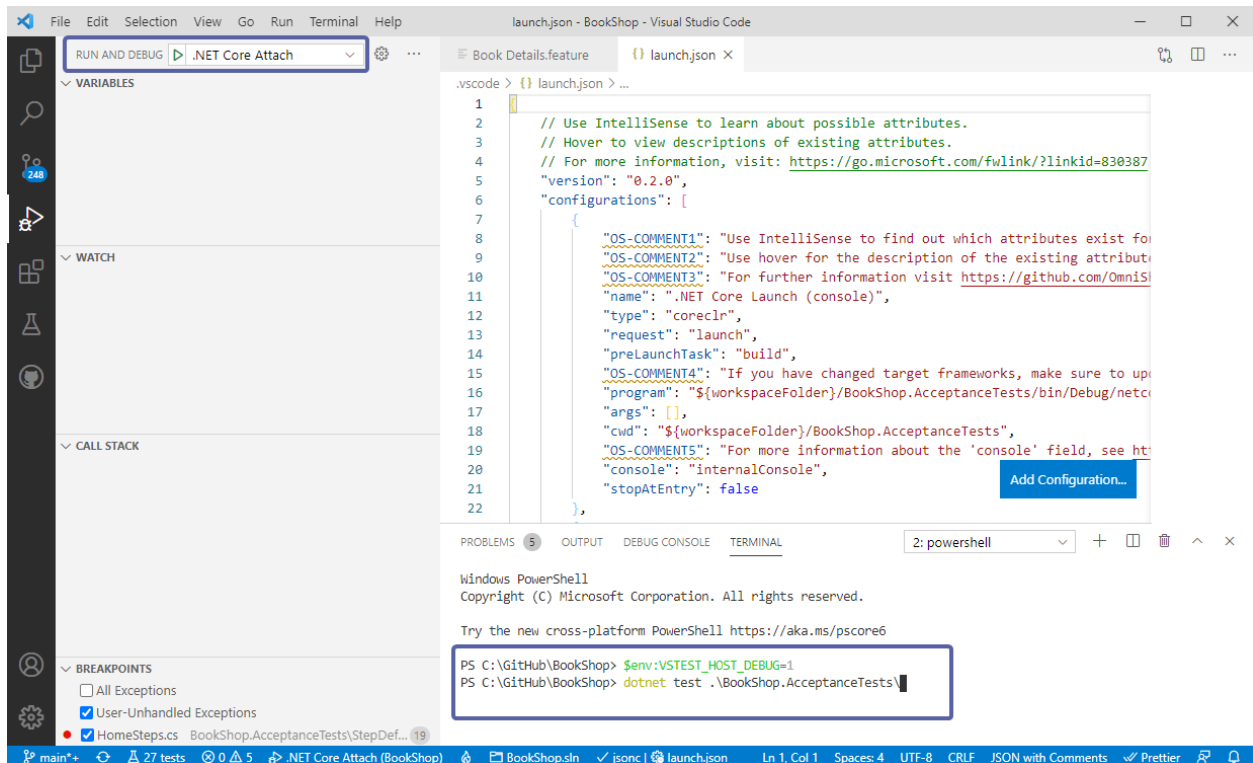
Core Attach which is what we are going for in this example, Select **.NET Core Attach**.



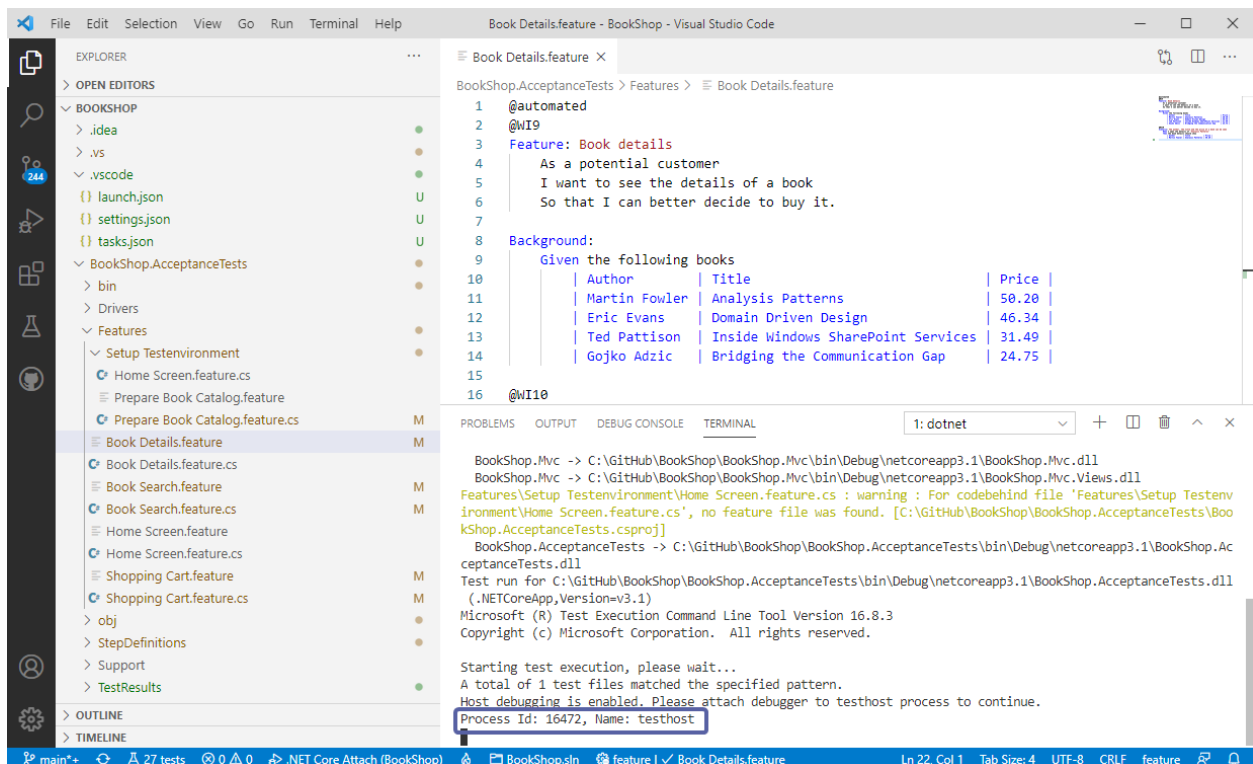
3- Open a new Terminal and enter the below into the terminal and hit Enter to set it up for debugging:

```
$env:VSTEST_HOST_DEBUG=1
```

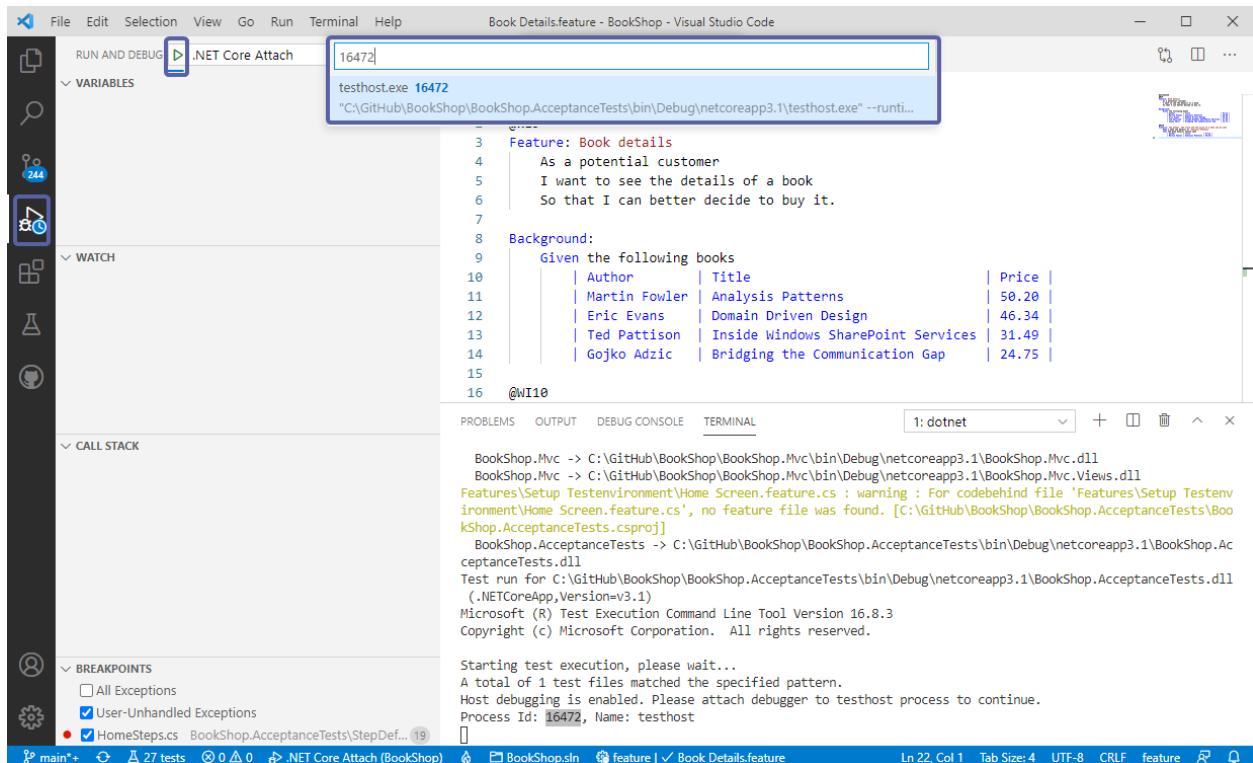
The terminal is now ready for debugging, Enter `dotnet test` followed by your project and hit Enter:



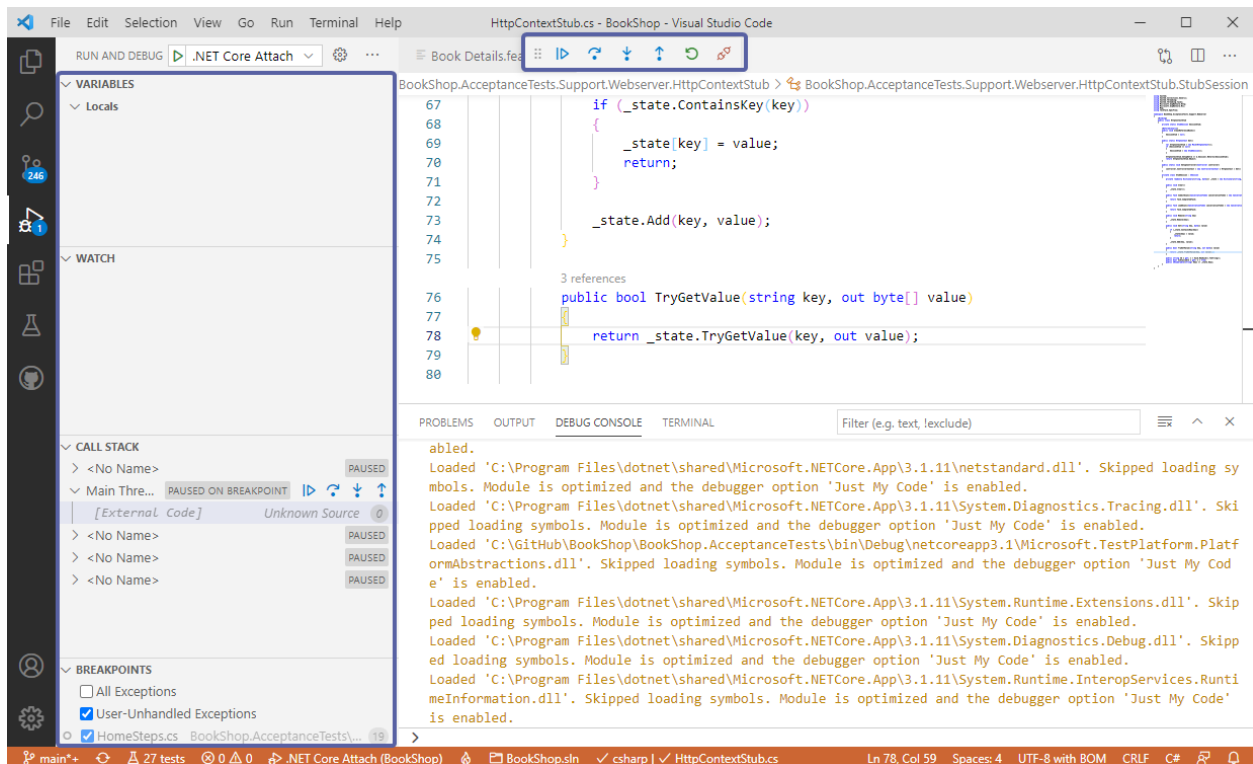
4- `dotnet test` will then start in debug mode and display the test host process ID on which you can attach the debugger to:



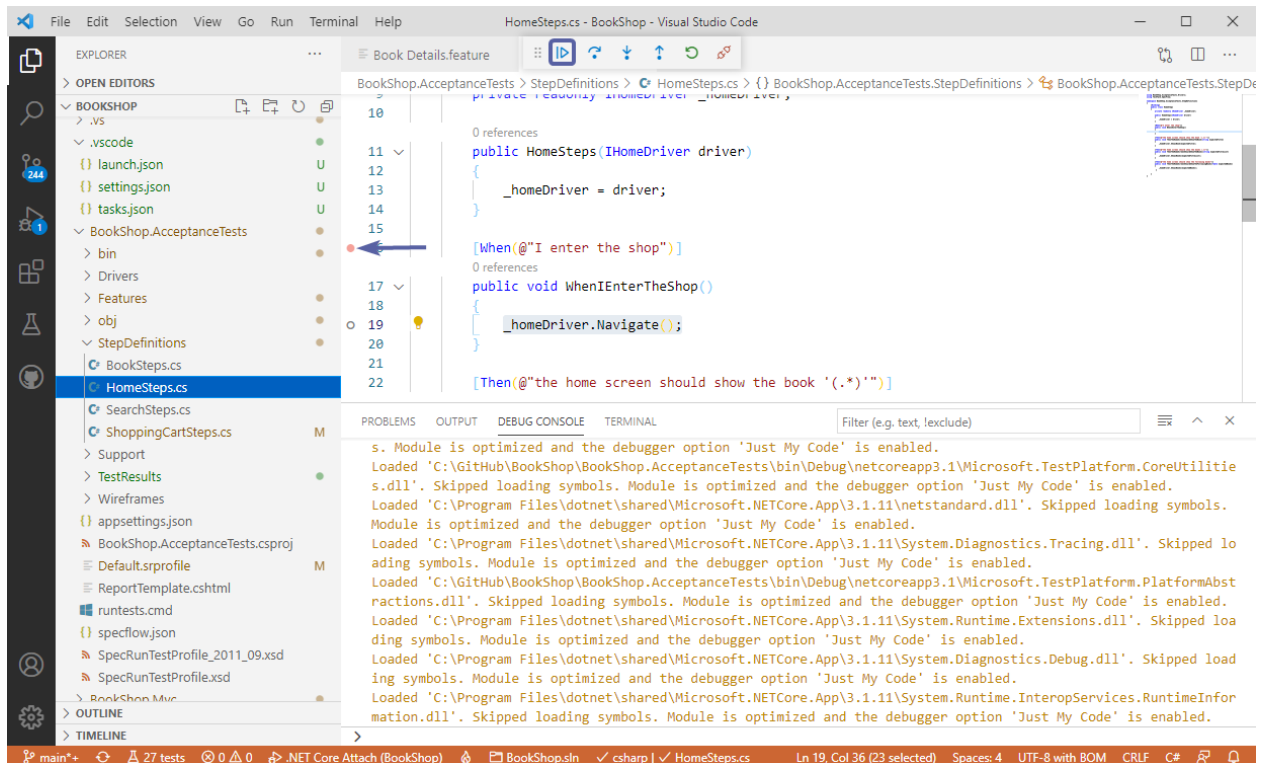
5- Copy the process ID number and navigate back to debug menu on the left pane and click on the icon next to **.NET Core Attach** and paste the process ID number in the displayed search box and select **testhost.exe**:



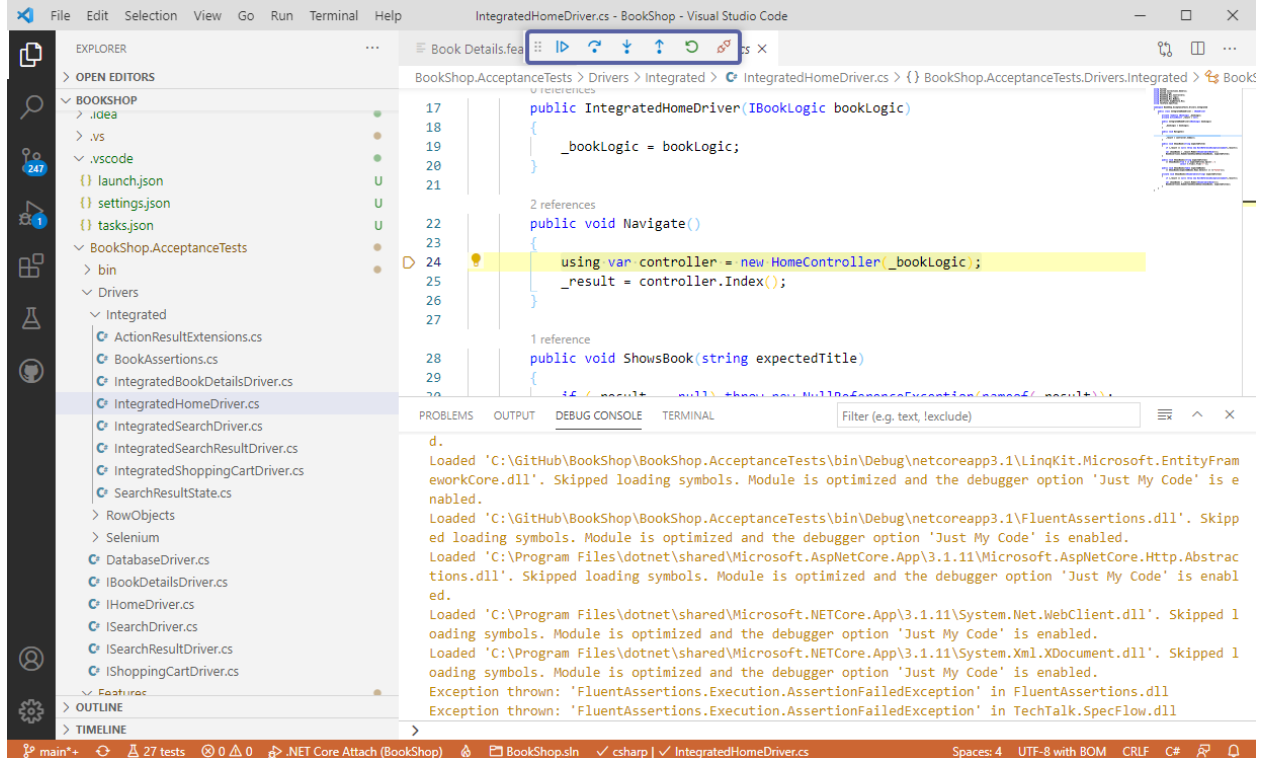
6- The debugger menu will then open up and you can see additional debugging options:



7- Now would be a good time to navigate to your files and set your break points before hitting the start button the menu:



8- The debugger menu buttons give you the options to navigate around and once finished you can disconnect by using the red unplugged button:



2.60 Value Retrievers

SpecFlow can turn properties in a table like this:

```
Given I have the following people
| First Name | Last Name | Age | IsAdmin |
| John      | Guppy    | 40  | true    |
```

Into an object like this:

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; }
    public bool IsAdmin { get; set; }
}
```

With commands like these:

```
[Given(@"I have the following people")]
public void x(Table table)
{
    var person = table.CreateInstance<Person>();
    // OR
    var people = table.CreateSet<Person>();
}
```

But how does SpecFlow match the values in the table with the values in the object? It does so with Value Retrievers. There are value retrievers defined for almost every C# base type, so mapping most basic POCOs can be done with SpecFlow without any modification.

2.60.1 Extending with your own value retrievers

Often you might have a more complicated POCO type, one that is not comprised solely of C# base types. Like this one:

```
public class Shirt
{
    public Guid Id { get; set; }
    public string Name { get; set; }
    public Color Color { get; set; }
}

public class Color
{
    public string Name { get; set; }
}
```

Simple example how to process the human readable color 'red' to the Hex value:

```
| First Name | ShirtColor |
| Scott     | Red        |
```

The table will be processed, and the following code can be used to capture the table translation and customize it:

```
public class ShirtColorValueRetriever : IValueRetriever
{
    public bool CanRetrieve(KeyValuePair<string, string> keyValuePair, Type targetType, Type propertyType)
    {
        if (!keyValuePair.Key.Equals("ShirtColor"))
        {
            return false;
        }

        bool value;
        if (Color.TryParse(keyValuePair.Value, out value))
        {
            return true;
        }
    }

    public object Retrieve(KeyValuePair<string, string> keyValuePair, Type targetType, Type propertyType)
    {
        return Color.Parse(keyValuePair.Value).HexCode;
    }
}
```

2.60.2 Registering Custom ValueRetrievers

Before you can utilize a custom ValueRetriever, you'll need to register it. We recommend doing this prior to a test run using the [BeforeTestRun] attribute and Service.Instance.ValueRetrievers.Register(). For example:

```
[Binding]
public static class Hooks
{
    [BeforeTestRun]
    public static void BeforeTestRun()
    {
        Service.Instance.ValueRetrievers.Register(new MyCustomValueRetriever());
    }
}
```

2.61 Plugins

SpecFlow supports the following types of plugins:

- Runtime
- Generator

All types of plugins are created in a similar way.

This information only applies to SpecFlow 3. For legacy information on plugins for previous versions, see [Plugins \(Legacy\)](#). With SpecFlow 3, we have changed how you configure which plugins are used. They are no longer configured

in your `app.config` (or `specflow.json`).

2.61.1 Runtime plugins

Runtime plugins need to target .NET Framework 4.6.1 and .NET Standard 2.0. SpecFlow searches for files that end with `.SpecFlowPlugin.dll` in the following locations:

- The folder containing your `TechTalk.SpecFlow.dll` file
- Your working directory

SpecFlow loads plugins in the order they are found in the folder.

Create a runtime plugin

You can create your `RuntimePlugin` in a separate project, or in the same project where your tests are.

Optional:

1. Create a new class library for your plugin.

Mandatory:

1. Add the SpecFlow NuGet package to your project.
2. Define a class that implements the `IRuntimePlugin` interface (defined in `TechTalk.SpecFlow.Plugins`).
3. Flag your assembly with the `RuntimePlugin` attribute for the plugin to be identified by SpecFlow plugin loader. The following example demonstrates a `MyNewPlugin` class that implements the `IRuntimePlugin` interface: `[assembly: RuntimePlugin(typeof(MyNewPlugin))]`
4. Implement the `Initialize` method of the `IRuntimePlugin` interface to access the `RuntimePluginEvents` and `RuntimePluginParameters`.

RuntimePluginsEvents

- `RegisterGlobalDependencies` - registers a new interface in the global container, see [Available Containers & Registrations](#)
- `CustomizeGlobalDependencies` - overrides registrations in the global container, see [Available Containers & Registrations](#)
- `ConfigurationDefaults` - adjust configuration values
- `CustomizeTestThreadDependencies` - overrides or registers a new interface in the test thread container, see [Available Containers & Registrations](#)
- `CustomizeFeatureDependencies` - overrides or registers a new interface in the feature container, see [Available Containers & Registrations](#)
- `CustomizeScenarioDependencies` - overrides or registers a new interface in the scenario container, see [Available Containers & Registrations](#)

Sample runtime plugin

A complete example of a Runtime plugin can be found [here](#). It packages a Runtime plugin as a NuGet package.

SampleRuntimePlugin.csproj

The sample project is [here](#).

This project targets multiple frameworks, so the project file uses `<TargetFrameworks>` instead of `<TargetFramework>`. Our target frameworks are .NET 4.6.1 and .NET Standard 2.0.

```
<TargetFrameworks>net461;netstandard2.0</TargetFrameworks>
```

We set a different `<AssemblyName>` to add the required `.SpecFlowPlugin` suffix to the assembly name. You can also simply name your project with `.SpecFlowPlugin` at the end.

```
<AssemblyName>SampleRuntimePlugin.SpecFlowPlugin</AssemblyName>
```

`<GeneratePackageOnBuild>` is set to true so that the NuGet package is generated on build. We use a NuSpec file (`SamplePlugin.nuspec`) to provide all information for the NuGet package. This is set with the `<NuspecFile>` property.

```
<GeneratePackageOnBuild>true</GeneratePackageOnBuild>
<NuspecFile>$(MSBuildThisFileDirectory)SamplePlugin.nuspec</NuspecFile>
```

Runtime plugins only need a reference to the SpecFlow NuGet package.

```
<ItemGroup>
  <PackageReference Include="SpecFlow" Version="3.0.199" />
</ItemGroup>
```

build/SpecFlow.SamplePlugin.targets

The sample targets file is [here](#).

We need to copy a different assembly to the output folder depending on the target framework (.NET Core vs .NET Framework) of the project using the runtime plugin package. Because the `$(TargetFrameworkIdentifier)` property is only available in imported targets files, we have to work out the path to the assembly here.

```
<_SampleRuntimePluginFramework Condition=" '$(TargetFrameworkIdentifier)' == '.NETCoreApp'
→ ' " ">netstandard2.0</_SampleRuntimePluginFramework>
<_SampleRuntimePluginFramework Condition=" '$(TargetFrameworkIdentifier)' == '.
→ NETFramework' ">net461</_SampleRuntimePluginFramework>
<_SampleRuntimePluginPath>$(MSBuildThisFileDirectory)\..\lib\$(
→ SampleRuntimePluginFramework)\SampleRuntimePlugin.SpecFlowPlugin.dll</_
→ SampleRuntimePluginPath>
```

build/SpecFlow.SamplePlugin.props

The sample props file is [here](#).

To copy the plugin assembly to the output folder, we include it in the None ItemGroup and set CopyToOutputDirectory to PreserveNewest. This ensures that it is still copied to the output directory even if you change it.

```
<ItemGroup>
  <None Include="$(_SampleRuntimePluginPath)" >
    <Link>%(Filename)%(Extension)</Link>
    <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
    <Visible>False</Visible>
  </None>
</ItemGroup>
```

Directory.Build.targets

The sample file is [here](#).

To specify the path to the assemblies which should be included in the NuGet package, we have to set various NuSpec properties. This is done in the Directory.Build.targets, so it defined for all projects in subfolders. We add the value of the current configuration to the available properties.

```
<Target Name="SetNuspecProperties" BeforeTargets="GenerateNuspec" >
  <PropertyGroup>
    <NuspecProperties>$(NuspecProperties);config=$(Configuration)</NuspecProperties>
  </PropertyGroup>
</Target>
```

SamplePlugin.nuspec

The sample file is [here](#).

This is the NuSpec file that provides information on the NuGet package. To get the files from the build directory into the NuGet package, we have to specify them in the file list. The runtime plugin assemblies are also specified here, using the additional \$config\$ property we added in the Directory.Build.targets.

```
<files>
  <file src="build\**\*" target="build"/>
  <file src="bin\${config}\net461\SampleRuntimePlugin.SpecFlowPlugin.*" target="lib\
↪net461"/>
  <file src="bin\${config}\netstandard2.0\SampleRuntimePlugin.SpecFlowPlugin.dll"
↪target="lib\netstandard2.0"/>
  <file src="bin\${config}\netstandard2.0\SampleRuntimePlugin.SpecFlowPlugin.pdb"
↪target="lib\netstandard2.0"/>
</files>
```

2.61.2 Generator plugins

Generator plugins need to target .NET Framework 4.7.1 and .NET Core 3.1. The MSBuild task needs to know which generator plugins it should use. You therefore have to add your generator plugin to the `SpecFlowGeneratorPlugins` ItemGroup. This is passed to the MSBuild task as a parameter and later used to load the plugins.

Create a generator plugin

1. Create a new class library for your plugin.
2. Add the `SpecFlow.CustomPlugin` NuGet package to your project.
3. Define a class that implements the `IGeneratorPlugin` interface (defined in `TechTalk.SpecFlow.Generator.Plugins` namespace).
4. Flag your assembly with the `GeneratorPlugin` attribute for the plugin to be identified by SpecFlow plugin loader. The following example demonstrates a `MyNewPlugin` class that implements the `IGeneratorPlugin` interface: `[assembly: GeneratorPlugin(typeof(MyNewPlugin))]`
5. Implement the `Initialize` method of the `IGeneratorPlugin` interface to access `GeneratorPluginEvents` and `GeneratorPluginParameters` parameters.

GeneratorPluginsEvents

- `RegisterDependencies` - registers a new interface in the Generator container
- `CustomizeDependencies` - overrides registrations in the Generator container
- `ConfigurationDefaults` - adjust configuration values

Sample generator plugin

A complete example of a generator plugin can be found [here](#). It packages a Generator plugin into a NuGet package.

SampleGeneratorPlugin.csproj

The sample project is [here](#).

The project targets multiple frameworks, so it uses `<TargetFrameworks>` and not `<TargetFramework>`. We set a different `<AssemblyName>` to add the required `.SpecFlowPlugin` at the end. You can also name your project with `.SpecFlowPlugin` at the end. `<GeneratePackageOnBuild>` is set to true, so that the NuGet package is generated on build. We use a NuSpec file (`SamplePlugin.nuspec`) to provide all information for the NuGet package. This is set with the `<NuspecFile>` property.

```
<PropertyGroup>
  <TargetFrameworks>net471;netcoreapp3.1</TargetFrameworks>
  <GeneratePackageOnBuild>true</GeneratePackageOnBuild>
  <NuspecFile>$(MSBuildThisFileDirectory)SamplePlugin.nuspec</NuspecFile>
  <AssemblyName>SampleGeneratorPlugin.SpecFlowPlugin</AssemblyName>
</PropertyGroup>
```

For a generator plugin you need a reference to the `SpecFlow.CustomPlugin`- NuGet package.


```
<ItemGroup>
  <PackageReference Include="SpecFlow.CustomPlugin" Version="3.0.199" />
</ItemGroup>
```

build/SpecFlow.SamplePlugin.targets

The sample targets file is [here](#).

We have to add a different assembly to the ItemGroup depending on the MSBuild version in use (.NET Full Framework or .NET Core). Because the \$(MSBuildRuntimeType) property is only available in imported target files, we have to work out the path to the assembly here.

```
<PropertyGroup>
  <_SampleGeneratorPluginFramework Condition=" '$(MSBuildRuntimeType)' == 'Core'">
    ↪netcoreapp3.1</_SampleGeneratorPluginFramework>
  <_SampleGeneratorPluginFramework Condition=" '$(MSBuildRuntimeType)' != 'Core'">
    ↪net471</_SampleGeneratorPluginFramework>
  <_SampleGeneratorPluginPath>$(MSBuildThisFileDirectory)\$(
    ↪SampleGeneratorPluginFramework)\SampleGeneratorPlugin.SpecFlowPlugin.dll</_
    ↪SampleGeneratorPluginPath>
</PropertyGroup>
```

build/SpecFlow.SamplePlugin.props

The sample props file is [here](#).

As we have now a property containing the path to the assembly, we can add it to the SpecFlowGeneratorPlugins ItemGroup here.

```
<ItemGroup>
  <SpecFlowGeneratorPlugins Include="$_SampleGeneratorPluginPath" />
</ItemGroup>
```

Directory.Build.targets

The sample file is [here](#).

To specify the path to the assemblies which should be included in the NuGet package, we have to set various Nuspec properties. This is done in Directory.Build.targets, so it defined for all projects in subfolders. We add the value of the current configuration to the available properties.

```
<Target Name="SetNuspecProperties" BeforeTargets="GenerateNuspec" >
  <PropertyGroup>
    <NuspecProperties>$(NuspecProperties);config=$(Configuration)</NuspecProperties>
  </PropertyGroup>
</Target>
```

SamplePlugin.nuspec

The sample file is [here](#).

This is the NuSpec file that provides information for the NuGet package. To get the files from the build directory into the NuGet package, we have to specify them in the file list. The generator plugin assemblies are also specified here, using the additional \$config\$ property we added in the Directory.Build.targets. It is important to ensure that they are not added to the lib folder. If this were the case, they would be referenced by the project where you add the NuGet package. This is something we don't want to happen!

```
<files>
  <file src="build\**\*" target="build"/>
  <file src="bin\${config}\net471\SampleGeneratorPlugin.SpecFlowPlugin.*" target="build\
↳net471"/>
  <file src="bin\${config}\netcoreapp3.1\SampleGeneratorPlugin.SpecFlowPlugin.dll"
↳target="build\netcoreapp3.1"/>
  <file src="bin\${config}\netcoreapp3.1\SampleGeneratorPlugin.SpecFlowPlugin.pdb"
↳target="build\netcoreapp3.1"/>
</files>
```

2.61.3 Combined Package with both plugins

If you need to update generator and runtime plugins with a single NuGet package (as we are doing with the SpecFlow.xUnit, SpecFlow.NUnit and SpecFlow.xUnit packages), you can do so.

As with the separate plugins, you need two projects. One for the runtime plugin, and one for the generator plugin. As you only want one NuGet package, the **NuSpec files must only be present in the generator project**. This is because the generator plugin is built with a higher .NET Framework version (.NET 4.7.1), meaning you can add a dependency on the Runtime plugin (which is only .NET 4.6.1). This will not working the other way around.

You can simply combine the contents of the .targets and .props file to a single one.

Example

A complete example of a NuGet package that contains a runtime and generator plugin can be found [here](#).

2.61.4 Tips & Tricks

Building Plugins on non-Windows machines

For building .NET 4.6.1 projects on non- Windows machines, the .NET Framework reference assemblies are needed.

You can add them with following PackageReference to your project:

```
<ItemGroup>
  <PackageReference Include="Microsoft.NETFramework.ReferenceAssemblies" Version="1.0.0
↳">
    <PrivateAssets>all</PrivateAssets>
    <IncludeAssets>runtime; build; native; contentfiles; analyzers</IncludeAssets>
  </PackageReference>
</ItemGroup>
```

2.61.5 Plugin Developer Channel

We have set up a Discord channel for plugin developers [here](#). If you have any questions regarding the development of plugins for SpecFlow, this is the place to ask them.

2.62 Available Plugins

2.62.1 Plugins for DI Container

2.62.2 Other Plugins

2.62.3 Outdated Plugins

2.63 Available Containers & Registrations

2.63.1 Global Container

The global container captures global services for test execution and the step definition, hook and transformation discovery result (i.e. what step definitions you have).

- `IRuntimeConfigurationProvider`
- `ITestRunnerManager`
- `ISStepFormatter`
- `ITestTracer`
- `ITraceListener`
- `ITraceListenerQueue`
- `IErrorProvider`
- `IRuntimeBindingSourceProcessor`
- `IRuntimeBindingRegistryBuilder`
- `IBindingRegistry`
- `IBindingFactory`
- `ISStepDefinitionRegexCalculator`
- `IBindingInvoker`
- `ISStepDefinitionSkeletonProvider`
- `ISkeletonTemplateProvider`
- `ISStepTextAnalyzer`
- `IRuntimePluginLoader`
- `IBindingAssemblyLoader`
- `IBindingInstanceResolver`
- [RuntimePlugins](#)
 - `RegisterGlobalDependencies`- Event

- CustomizeGlobalDependencies- Event

2.63.2 Test Thread Container

> **Note:** *Parent Container is the Global Container*

The test thread container captures the services and state for executing scenarios on a particular test thread. For parallel test execution, multiple test runner containers are created, one for each thread.

- ITestRunner
- IContextManager
- ITestExecutionEngine
- IStepArgumentTypeConverter
- IStepDefinitionMatchService
- ITraceListener
- ITestTracer
- RuntimePlugins
 - CustomizeTestThreadDependencies- Event

2.63.3 Feature Container

> **Note:** *Parent Container is the Test Thread Container*

The feature container captures a feature's execution state. It is disposed after the feature is executed.

- FeatureContext (also available from the *test thread container* through IContextManager)
- [RuntimePlugins]
 - CustomizeFeatureDependencies- Event

2.63.4 Scenario Container

> **Note:** *Parent Container is the Test Thread Container*

The scenario container captures the state of a scenario execution. It is disposed after the scenario is executed.

- (step definition classes)
- (dependencies of the step definition classes, aka context injection)
- ScenarioContext (also available from the *Test Thread Container* through IContextManager)
- [RuntimePlugins]
 - CustomizeScenarioDependencies- Event

2.64 Decorators

SpecFlow supports decorators which can be used in feature files. Decorators can be used to convert a tag in a feature file to an attribute in the generated code behind file.

2.64.1 Example decorator

Say we want to add an NUnit `Apartment` attribute to a test method in the generated code behind file (a file with extension `.feature.cs`) to specify that the test should be running in a particular apartment, either the STA or the MTA. For this, we can use a decorator which we need to register in a generator plugin so that the decorator can have its effect during the code behind file generation.

Steps to follow:

1. Create a SpecFlow project with test framework NUnit using the project template provided by the SpecFlow Visual Studio extension. [Learn more](#)
2. Create a GeneratorPlugin. You can follow the steps [here](#) or you can use the [sample generator plugin project](#) as a basis
3. Create a Decorator (which is a class which implements interfaces like `ITestMethodTagDecorator`, `ITestMethodDecorator`, etc.):
 - `ITestMethodDecorator` is called always
 - `ITestMethodTagDecorator` is called only if the scenario has at least one tag

```
public class MyMethodTagDecorator : ITestMethodTagDecorator
{
    public static readonly string TAG_NAME = "myMethodTagDecorator";
    private readonly ITagFilterMatcher _tagFilterMatcher;

    public MyMethodTagDecorator(ITagFilterMatcher tagFilterMatcher)
    {
        _tagFilterMatcher = tagFilterMatcher;
    }

    public bool CanDecorateFrom(string tagName, TestClassGenerationContext _
    ↪ generationContext, CodeMemberMethod testMethod)
    {
        return _tagFilterMatcher.Match(TAG_NAME, tagName);
    }

    public void DecorateFrom(string tagName, TestClassGenerationContext _
    ↪ generationContext, CodeMemberMethod testMethod)
    {
        var attribute = new CodeAttributeDeclaration(
            "NUnit.Framework.ApartmentAttribute",
            new CodeAttributeArgument(
                new CodeFieldReferenceExpression(
                    new CodeTypeReferenceExpression(typeof(System.Threading.
    ↪ ApartmentState)),
                    "STA"))));

        testMethod.CustomAttributes.Add(attribute);
    }
}
```

(continues on next page)

(continued from previous page)

```
}

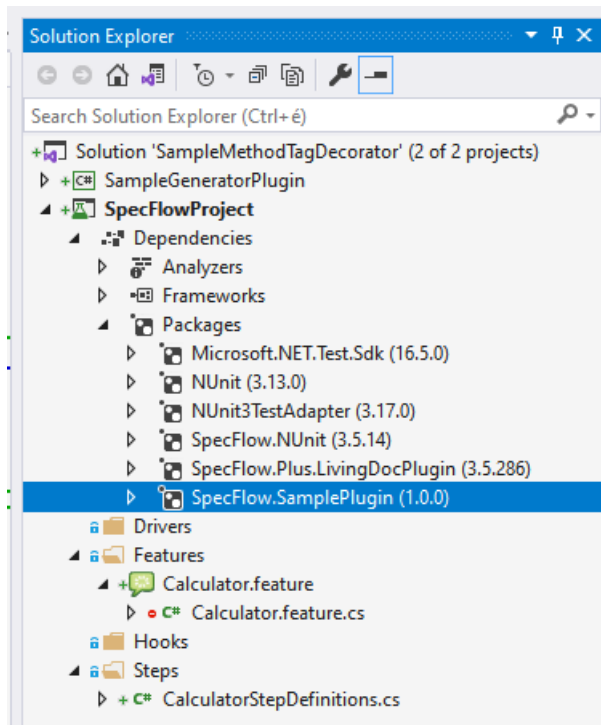
public int Priority { get; }
public bool RemoveProcessedTags { get; }
public bool ApplyOtherDecoratorsForProcessedTags { get; }
}
```

4. Register the Decorator in the Initialize method of the GeneratorPlugin:

```
public void Initialize(GeneratorPluginEvents generatorPluginEvents,
    GeneratorPluginParameters generatorPluginParameters,
    UnitTestProviderConfiguration unitTestProviderConfiguration)
{
    // Register the decorator
    generatorPluginEvents.RegisterDependencies += RegisterDependencies;
}

private void RegisterDependencies(object sender, RegisterDependenciesEventArgs
    eventArgs)
{
    eventArgs.ObjectContainer.RegisterTypeAs<MyMethodTagDecorator,
    ITestMethodTagDecorator>(MyMethodTagDecorator.TAG_NAME);
}
```

5. Install the GeneratorPlugin NuGet package to the SpecFlow project. Once the installation finishes, it should look like this:



6. Add tag @myMethodTagDecorator to the feature file:

```

Calculator.feature
1 Feature: Calculator
2 ![[Calculator]](https://specflow.org/wp-content
3 Simple calculator for adding **two** numbe
4
5 Link to a feature: [Calculator](SpecFlowPr
6 ***Further read***: **[Learn more about hc
7
8 @myMethodTagDecorator
9 Scenario: Add two numbers
10     Given the first number is 50
11     And the second number is 70
12     When the two numbers are added
13     Then the result should be 120

```

7. Build the solution

8. Check the generated code behind file (.feature.cs) if it contains the NUnit Apartment attribute:

```

Calculator.feature.cs
SpecFlowProject
78
79
80 [NUnit.Framework.TestAttribute()]
81 [NUnit.Framework.DescriptionAttribute("Add two numbers")]
82 [NUnit.Framework.ApartmentAttribute(System.Threading.ApartmentState.STA)]
83 [NUnit.Framework.CategoryAttribute(name: "myMethodTagDecorator")]
84 0 references | 0 changes | 0 authors, 0 changes
85 public virtual void AddTwoNumbers()
{

```

You can find the complete Decorator example on GitHub.

2.64.2 Further read

- NUnit Apartment attribute: <https://docs.nunit.org/articles/nunit/writing-tests/attributes/apartment.html>
- Apartments: <https://docs.microsoft.com/en-us/windows/win32/com/processes-threads-and-apartments>

2.65 Generate Tests From MsBuild

2.65.1 General

You need to use the MSBuild code behind file generation for SpecFlow 3.0.

After version SpecFlow 3.3.30 don't need to add the SpecFlow.Tools.MSBuild.Generation package anymore to your project, if you are using one of our *Unit-Test-Provider* NuGet packages.

Note: You will need at least VS2017/MSBuild 15 to use this package.

Configuration

1. Add the NuGet package `SpecFlow.Tools.MsBuild.Generation` with the same version as SpecFlow to your project.
2. Remove all `SpecFlowSingleFileGenerator` custom tool entries from your feature files.
3. Select Tools | Options | SpecFlow from the menu in Visual Studio, and set Enable `SpecFlowSingleFileGenerator CustomTool` to "false".

SDK Style project system

Please use the SpecFlow 2.4.1 NuGet package or higher, as this version fixes an issue with previous versions (see *Known Issues* below)

2.65.2 Additional Legacy Options (Prior to SpecFlow 3)

The `TechTalk.SpecFlow.targets` file defines a number of default options in the following section:

```
<PropertyGroup>
  <ShowTrace Condition="'$(ShowTrace)'==''">false</ShowTrace>
  <OverwriteReadOnlyFiles Condition="'$(OverwriteReadOnlyFiles)'==''">false</
  <OverwriteReadOnlyFiles>
  <ForceGeneration Condition="'$(ForceGeneration)'==''">false</ForceGeneration>
  <VerboseOutput Condition="'$(VerboseOutput)'==''">false</VerboseOutput>
</PropertyGroup>
```

- `ShowTrace`: Set this to true to output trace information.
- `OverwriteReadOnlyFiles`: Set this to true to overwrite any read-only files in the target directory. This can be useful if your feature files are read-only and part of your repository.
- `ForceGeneration`: Set this to true to forces the code-behind files to be regenerated, even if the content of the feature has not changed.
- `VerboseOutput`: Set to true to enable verbose output for troubleshooting.

To change these options, add the corresponding element to your project file **before** the `<Import>` element you added earlier.

Example:

```
<PropertyGroup>
  <ShowTrace>true</ShowTrace>
  <VerboseOutput>true</VerboseOutput>
</PropertyGroup>
...
</ItemGroup>
<Import Project="$(MSBuildBinPath)\Microsoft.CSharp.targets" />
<Import Project="..\packages\SpecFlow.2.2.0\tools\TechTalk.SpecFlow.tasks" Condition=
  <Exists('..\packages\SpecFlow.2.2.0\tools\TechTalk.SpecFlow.tasks') />
<Import Project="..\packages\SpecFlow.2.2.0\tools\TechTalk.SpecFlow.targets" Condition=
  <Exists('..\packages\SpecFlow.2.2.0\tools\TechTalk.SpecFlow.targets') />
...
</Project>
```


2.65.3 Known Issues

SpecFlow prior to 2.4.1

When using SpecFlow NuGet packages prior to SpecFlow 2.4.1, Visual Studio sometimes does not recognize that a feature file has changed. To generate the code-behind file, you therefore need to rebuild your project. We recommend upgrading your SpecFlow NuGet package to 2.4.1 or higher, where this is no longer an issue.

Code-behind files not generating at compile time

When using the classic project system, the previous MSBuild target may no longer be located at the end of your project. NuGet ignores entries added manually. NuGet places the MSBuild imports at the end. However, the `AfterUpdateFeatureFilesInProject` target needs to be defined after the imports. Otherwise it will be overwritten with an empty definition. If this happens, your code-behind files are not compiled as part of the assembly.

Linked files are not included

If you link feature files into a project, no code-behind file is generated for them (see [GitHub Issue 1295](#)).

2.66 Reporting

2.66.1 SpecFlow+ LivingDoc

Reporting is now easier to generate and share thanks to SpecFlow+ LivingDoc.

You can see your test execution results in a dynamic way now and also check for any [unused step definitions](#) along with many other handy features that truly bring your reporting and documentation to life!

Head over to [SpecFlow+ LivingDoc](#) to read more.

Here is a quick snapshot:

The screenshot displays the SpecFlow+ LivingDoc reporting interface. On the left, a sidebar shows the project structure for 'BookShop.AcceptanceTests', which was generated on Nov 24, 2020, at 10:47 AM GMT+1. The sidebar lists several features, with 'Displaying Home Screen' selected. The main area shows the details for this feature, including a background scenario and a table of books. The scenario is 'Cheapest 3 books should be listed on the home screen'. The table lists books with their titles and prices.

BookShop.AcceptanceTests
generated Nov 24, 2020, 10:47 AM GMT+1

Living Documentation Analytics

Filter by Keyword Filter by X

Test results

BookShop.AcceptanceTests 8 Passed 0 Failed 0 Others

Features 8 Passed 0 Failed 0 Others

Setup Testenvironment 1 Passed 0 Failed 0 Others

Shopping Cart 4 Passed 0 Failed 0 Others

Displaying Home Screen

Cheapest 3 books should be listed on the home screen

Cheapest 3 books should be listed on the home screen (list syntax)

Cheapest 3 books should be listed on the home screen (table syntax)

Displaying book details

Searching for books

Feature: Displaying Home Screen

As a potential customer
I want to see the books with the best price
So that I can save money on buying discounted books.

Background:

Given the following books

Title	Price
Analysis Patterns	50.20
Domain Driven Design	46.34
Inside Windows SharePoint Services	31.49
Bridging the Communication Gap	24.75

Scenario: Cheapest 3 books should be listed on the home screen

When I enter the shop

Then the home screen should show the book 'Bridging the Communication Gap'

And the home screen should show the book 'Inside Windows SharePoint Services'

And the home screen should show the book 'Domain Driven Design'

2.66.2 Reporting prior to SpecFlow 3

These reports are only available prior to SpecFlow 3! They have been removed in SpecFlow 3.

SpecFlow provides various options to generate reports related to the acceptance tests.

Note: The `specflow.exe` [command line tool](#) that is used to generate reports can be found in the `packages\Specflow.{version number}\tools` directory, when you installed SpecFlow through NuGet. Start the tool with no parameters or use the `--help` option to display an overview of the available options.

You can find a repository containing the old report code [here](#). For information on why the reports were moved to a separate repo, please read see this [GitHub issue](#).

Test Execution Report

This report generates an HTML test execution report. The report contains a summary of the executed tests and their results, as well as a detailed report of the execution of individual scenarios.

The following sub-sections cover generating the test execution report for different unit test providers.

NUnit Test Execution Report

NUnit 2

In order to generate this report, execute the acceptance tests with the `nunit-console` runner. This tool generates an XML summary of the test executions. To include detailed scenario execution traces, you need to capture the test output using the `/out` and the `/labels` options, e.g.

```
nunit-console.exe /labels /out=TestResult.txt /xml=TestResult.xml bin\Debug\BookShop.  
↪AcceptanceTests.dll
```

NUnit 3

In order to generate this report, execute the acceptance tests with the `nunit3-console` runner, and set it to output the results in `nunit2` format. To include detailed scenario execution traces, you need to capture the test output using the `--out` and the `--labels=All` options (see the example below).

Important: The `NUnit.Extension.NUnitV2ResultWriter` package must be included in your project, otherwise you will receive the message: "Unknown result format: nunit2".

```
nunit3-console.exe --labels=All --out=TestResult.txt "--result=TestResult.xml;  
↪format=nunit2" bin\Debug\BookShop.AcceptanceTests.dll
```

SpecFlow Report Generator

Note: The examples and parameters are for version 2.4.* and higher. Older versions can be found by viewing past revisions in the GitHub wiki.

The report generation step is the same for both versions of NUnit. The two generated files can be used to invoke the SpecFlow report generation. If you use the default output file names shown above, you only need to specify information about the C# test project containing the *.feature files. Specflow uses the default `TestResult.xml` and `TestResult.txt` files to produce `TestResult.html`.

```
specflow.exe nunitexecutionreport --ProjectFile BookShop.AcceptanceTests.csproj
specflow.exe nunitexecutionreport --ProjectFile BookShop.AcceptanceTests.csproj --
  ↪ xmlTestResult CustomNunitTestResult.xml --testOutput CustomNunitTestOutput.txt --
  ↪ OutputFile CustomSpecflowTestReport.html
```

The following table contains the possible arguments for this command.

MsTest Test Execution Report

The following table contains the possible arguments for this command.

Step Definition Report

This report shows the usage and binding status of the steps in your entire project. You can use this report to find both unused code in the automation layer and scenario steps with no binding defined.

- Steps with a red background are steps that exist in the automation layer but are not used in any feature files.
- Steps with a yellow background are steps that exist in a feature file but do not have a corresponding binding defined.
- Steps without a special backgrounds are steps that exist both in feature files and the automation layer. Ideally, all your steps are like this.

```
specflow.exe stepdefinitionreport --ProjectFile BookShop.AcceptanceTests.csproj /
  ↪ BinFolder:bin/debug
```

The following table contains the possible arguments for this command.

2.67 Prerequisite

- Visual Studio 2019 or higher
 - Workloads
 - * ASP.NET and web development
 - * .NET desktop development
 - * .NET Core cross- platform development
- .NET Core SDK 3.1
- .NET 5 SDK
- .NET 6 SDK

2.68 Local Setup

2.68.1 Clone the code

Clone the repository with submodules

```
git clone --recurse-submodules https://github.com/techtalk/SpecFlow.git
```

You need to clone the repository with submodules, because the code for the SpecFlow.TestProjectGenerator is located in [another repository](#). This is due to the fact that this code is shared with other projects.

2.68.2 Setting environment variables

MSBUILDDISABLENODEREUSE

You have to set MSBUILDDISABLENODEREUSE to 1. Reason for this is, that SpecFlow has an MSBuild Task that is used in the TechTalk.SpecFlow.Specs project. Because of the using of the task and MSBuild reuses processes, the file is loaded by MSBuild and will then lock the file and break the next build.

This environment variable controls the behaviour if MSBuild reuses processes. Setting to 1 disables this behaviour.

See [here](#) for more info.

2.69 Definition of Terms

2.69.1 Runtime

Runtime is then, when scenarios are executed by a test runner

2.69.2 GeneratorTime

GeneratorTime is then, when the code-behind files are generated.

2.69.3 Code-Behind files

For every feature file, SpecFlow generates a code-behind file, which contains the code for the various test frameworks. It's generated when the project gets compiled. This is done by the SpecFlow.Tools.MsBuild.Generation MSBuild task.

2.70 Projects

2.70.1 TechTalk.SpecFlow

This is the runtime part of SpecFlow.

2.70.2 TechTalk.SpecFlow.Generator

This is the main part of SpecFlow that is used at GeneratorTime

2.70.3 TechTalk.SpecFlow.Parser

This contains the parser for Feature- Files. We use Gherkin and added some additional features to the object model and parser.

2.70.4 TechTalk.SpecFlow.Utils

This project contains some helper classes.

2.70.5 SpecFlow.Tools.MsBuild.Generation

This project contains the MSBuild task that generates the code-behind files.

2.70.6 TechTalk.SpecFlow.GeneratorTests

This contains unit tests that are about the generation of the code-behind files.

2.70.7 TechTalk.SpecFlow.RuntimeTests

This contains unit tests that are about the runtime of Scenarios.

2.70.8 TechTalk.SpecFlow.MSTest.SpecFlowPlugin

This is the plugin for MSTest. It contains all specific implementations for MSTest.

2.70.9 TechTalk.SpecFlow.NUnit.SpecFlowPlugin

This is the plugin for NUnit. It contains all specific implementations for NUnit.

2.70.10 TechTalk.SpecFlow.xUnit.SpecFlowPlugin

This is the plugin for xUnit. It contains all specific implementations for xUnit.

2.70.11 TechTalk.SpecFlow.TestProjectGenerator

This project provides an API for generating projects, compile them and run tests.

2.70.12 TechTalk.SpecFlow.TestProjectGenerator.Tests

Tests for TechTalk.SpecFlow.TestProjectGenerator

2.70.13 TechTalk.SpecFlow.Specs

This project contains the integration tests for SpecFlow

2.70.14 TechTalk.SpecFlow.Specs.Generator.SpecFlowPlugin

This is a generator plugin, that generates the integration tests for various combinations. They can differ in Framework Version, Testing Framework, Project Format and programming language

2.71 Special files

2.71.1 Directory.Build.props

Explanation can be found [here](#)

In this file we set the general MSBuild properties for all projects.

Important to note is the PropertyGroup for different Framework versions. Here, we control which part of SpecFlow is compiled for which .NET Framework version.

2.71.2 TestRunCombinations.cs

This file controls in which combinations the integration tests should be generated.

2.72 Potential problems

2.72.1 Error “No templates matched the input template name” when executing tests

This error occurs, when somehow your local template cache has some problems. It's located in C:\Users\%username%\templateengine\dotnetcli\<used .NET Core SDK Version>\. To fix it, simple delete the cache. At the next execution, it will be regenerated.

2.73 Coding Style

Please use the same coding style as the one already used!

2.73.1 Static versus Instance Methods

We prefer instance methods, even if they can be made static because they do not use instance members. Making a static methods into an instance method happens relatively often and can entail a lot of work.

2.73.2 Naming Conventions for Tests

The test class should be named like the class it is testing, with a `Tests` suffix. So for example: if a class is named `Calculator`, then the test class is called `CalculatorTests`.

Each test method is named by three parts, separated by an underscore. The parts are “method or property under test”, “scenario” and “expected result”. For example, if we want to test the `Add` method with a small positive and a big negative argument and the result should be negative, then the text method would be called `Add_SmallPositiveAndBigNegativeArgument_ResultShouldBeNegative`.

2.73.3 Private fields

Private fields begin with a `_` (underscore).

2.74 Plugins

2.74.1 Introduction

SpecFlow provides a plugin infrastructure, allowing customization. You can develop SpecFlow plugins that change the behavior of the built-in generator and runtime components. For example, a plugin could provide support for a new unit testing framework.

To use a custom plugin it has to be enabled in the `[[configuration]]` (`app.config`) of your SpecFlow project:

```
<specFlow>
  <plugins>
    <add name="MyPlugin" />
  </plugins>
</specFlow>
```

2.74.2 Creating plugins

Creating a plugin is fairly simple. There are 3 types of plugins supported:

- Runtime
- Generator
- Runtime Generator

By default, SpecFlow assumes plugins are Runtime Generator plugins. If your plugin is either a Runtime or Generator plugin, you need to add this information to the configuration.

Example for a Runtime plugin:

```
<specFlow>
  <plugins>
    <add name="MyPlugin" type="Runtime"/>
  </plugins>
</specFlow>
```

Example for a Generator plugin:

```
<specFlow>
  <plugins>
    <add name="MyPlugin" type="Generator"/>
  </plugins>
</specFlow>
```

The steps required to create plugins of all types are similar. **All plugins require the suffix “.SpecFlowPlugin”.**

Generator Plugin

Needed steps for creating a Generator Plugin

1. A SpecFlow.CustomPlugin Nuget package added to the library that will contain the plugin.
2. A class that implements IGeneratorPlugin interface (which is defined in TechTalk.SpecFlow.Generator.Plugins namespace)
3. An assembly level attribute GeneratorPlugin pointing to the class that implements IGeneratorPlugin

Let's analyze all of these steps in detail.

I will advise you start a new class library for each plugin you intend to create. Once you create your class library as a first thing you should add the SpecFlow.CustomPlugin Nuget package to your project. Once this is done, you need to define a class that will represent your plugin. For this class in order to be seen as a SpecFlow plugin, it needs to implement the IGeneratorPlugin interface. By implementing the Initialize- Method on the IGeneratorPlugin interface, you get access to the GeneratorPluginEvents and GeneratorPluginParameters.

GeneratorPluginEvents

- *ConfigurationDefaults* – If you are planning to intervene at the SpecFlow configuration, this is the right place to get started.
- *CustomizeDependencies* – If you are extending any of the components of SpecFlow, you can register your implementation at this stage.
- *RegisterDependencies* – In case your plugin is of a complex nature and it has it's own dependencies, this can be the right place to set your Composition Root.

As an example, if you are writing a plugin that will act as unit test generator provider, you are going to register it in the following way, inside the *CustomizeDependencies* event handler:

```
public void Initialize(GeneratorPluginEvents generatorPluginEvents,
    GeneratorPluginParameters generatorPluginParameters)
{
    generatorPluginEvents.CustomizeDependencies += CustomizeDependencies;
```

(continues on next page)

(continued from previous page)

```

}

public void CustomizeDependencies(CustomizeDependenciesEventArgs eventArgs)
{
    eventArgs.ObjectContainer.RegisterTypeAs<MyNewGeneratorProvider,
    ↳ IUnitTestGeneratorProvider>();
}

```

In order for your new library to be picked up by SpecFlow plugin loader, you need to flag your assembly with the `GeneratorPlugin` attribute. This is an example of it, taking in consideration that the class that implements `IGeneratorPlugin` interface is called `MyNewPlugin`.

```
[assembly: GeneratorPlugin(typeof(MyNewPlugin))]
```

Runtime Plugin

Needed steps for creating a Runtime Plugin

1. A SpecFlow.CustomPlugin Nuget package added to the library that will contain the plugin.
2. A class that implements `IRuntimePlugin` interface (which is defined in `TechTalk.SpecFlow.Plugins` namespace)
3. An assembly level attribute `RuntimePlugin` pointing to the class that implements `IRuntimePlugin`

By implementing the `Initialize`- Method on the `IRuntimePlugin` interface, you get access to the `RuntimePluginEvents` and `RuntimePluginParameters`.

RuntimePluginsEvents

- *RegisterGlobalDependencies* - register new interfaces to the global container, see [Available-Containers-&-Registrations](#)
- *CustomizeGlobalDependencies* - override registrations in the global container, see [Available-Containers-&-Registrations](#)
- *ConfigurationDefaults* - adjust configuration values
- *CustomizeTestThreadDependencies* - override or register new interfaces in the test thread container, see [Available-Containers-&-Registrations](#)
- *CustomizeFeatureDependencies* - override or register new interfaces in the feature container, see [Available-Containers-&-Registrations](#)
- *CustomizeScenarioDependencies* - override or register new interfaces in the scenario container, see [Available-Containers-&-Registrations](#)

In order for your new library to be picked up by SpecFlow plugin loader, you need to flag your assembly with the `RuntimePlugin` attribute. This is an example of it, taking in consideration that the class that implements `IRuntimePlugin` interface is called `MyNewPlugin`.

```
[assembly: RuntimePlugin(typeof(MyNewPlugin))]
```

Note: Parameters are not yet implemented (Version 2.1)

2.74.3 Configuration details

In order to load your plugin, in your SpecFlow project, you need to reference your plugin in the app.config file without the ".SpecFlowPlugin" suffix. It is also handy to know that the path attribute considers that project root as a path root. The following example is used to load a plugin assembly called "MyNewPlugin.SpecFlowPlugin.dll" that is located in a folder called "Binaries" that is at the same level of the current project.

```
<specFlow>
  <plugins>
    <add name="MyNewPlugin" path="..\Binaries" />
  </plugins>
</specFlow>
```

2.74.4 Sample plugin implementations:

For reference, here are some sample implementations of an `IRuntimePlugin` and `IGeneratorPlugin`:

`SpecFlow.FsCheck`

`SpecFlow.Autofac`

2.75 Coded UI

2.75.1 Introduction

Note: Coded UI is no longer supported with SpecFlow 3. We recommend using Appium or WinAppDriver instead. The following is preserved for legacy users.

The Microsoft Coded UI API can be used to create automated tests in Visual Studio, but is not directly compatible with SpecFlow as each Test Class requires the `[CodedUITest]` attribute, which SpecFlow does not generate by default.

Big thanks go to Thomy Kay for [pointing us in the right direction](#).

2.75.2 Solution

You need to ensure SpecFlow generates the `[CodedUITest]` attribute by creating a custom test generator provider, copying the DLL file into the `tools` directory where the SpecFlow NuGet package is installed, and ensure that any SpecFlow hooks also ensure the CodedUI API is initialized.

Generating the `[CodedUITest]` attribute with VS2010 and MSTest

1. Create a new VS project to generate an assembly that contains the class below. This will require a reference to `TechTalk.SpecFlow.Generator.dll` in the SpecFlow directory. If you are using version 1.7 or higher you will also need to add a reference to `TechTalk.SpecFlow.Utils.dll`
2. Add the following class to your new VS project:

SpecFlow version 1.6

```

namespace My.SpecFlow
{
    using System.CodeDom;
    using TechTalk.SpecFlow.Generator.UnitTestProvider;

    public class MsTest2010CodedUiGeneratorProvider : MsTest2010GeneratorProvider
    {
        public override void SetTestFixture(System.CodeDom.CodeTypeDeclaration typeDeclaration, string title, string description)
        {
            base.SetTestFixture(typeDeclaration, title, description);
            foreach (CodeAttributeDeclaration customAttribute in typeDeclaration.CustomAttributes)
            {
                if (customAttribute.Name == "Microsoft.VisualStudio.TestTools.UnitTesting.TestClassAttribute")
                {
                    typeDeclaration.CustomAttributes.Remove(customAttribute);
                    break;
                }
            }

            typeDeclaration.CustomAttributes.Add(new CodeAttributeDeclaration(new CodeTypeReference("Microsoft.VisualStudio.TestTools.UnitTesting.CodedUITestAttribute")));
        }
    }
}

```

SpecFlow version 1.7

```

using System.CodeDom;
using TechTalk.SpecFlow.Generator.UnitTestProvider;

namespace SpecflowCodedUiGenerator
{
    public class MsTest2010CodedUiGeneratorProvider : MsTest2010GeneratorProvider
    {
        public override void SetTestClass(TechTalk.SpecFlow.Generator.TestClassGenerationContext generationContext, string featureTitle, string featureDescription)
        {
            base.SetTestClass(generationContext, featureTitle, featureDescription);

            foreach (CodeAttributeDeclaration customAttribute in generationContext.TestClass.CustomAttributes)
            {
                if (customAttribute.Name == "Microsoft.VisualStudio.TestTools.UnitTesting.TestClassAttribute")
                {
                    generationContext.TestClass.CustomAttributes.Remove(customAttribute);
                    break;
                }
            }
        }
    }
}

```

(continues on next page)

(continued from previous page)

```
        generationContext.TestClass.CustomAttributes.Add(new_  
↪ CodeAttributeDeclaration(new CodeTypeReference("Microsoft.VisualStudio.TestTools.  
↪ UITesting.CodedUITestAttribute"))));  
    }  
}  
}
```

SpecFlow version 1.9

```
using System.CodeDom;  
using TechTalk.SpecFlow.Generator.UnitTestProvider;  
  
namespace SpecflowCodedUIGenerator  
{  
    public class MsTest2010CodedUiGeneratorProvider : MsTest2010GeneratorProvider  
    {  
        public MsTest2010CodedUiGeneratorProvider(CodeDomHelper codeDomHelper)  
            : base(codeDomHelper)  
        {  
        }  
  
        public override void SetTestClass(TechTalk.SpecFlow.Generator.  
↪ TestClassGenerationContext generationContext, string featureTitle, string_  
↪ featureDescription)  
        {  
            base.SetTestClass(generationContext, featureTitle, featureDescription);  
  
            foreach (CodeAttributeDeclaration customAttribute in generationContext.  
↪ TestClass.CustomAttributes)  
            {  
                if (customAttribute.Name == "Microsoft.VisualStudio.TestTools.  
↪ UITesting.TestClassAttribute")  
                {  
                    generationContext.TestClass.CustomAttributes.Remove(customAttribute);  
                    break;  
                }  
            }  
  
            generationContext.TestClass.CustomAttributes.Add(new_  
↪ CodeAttributeDeclaration(new CodeTypeReference("Microsoft.VisualStudio.TestTools.  
↪ UITesting.CodedUITestAttribute"))));  
        }  
    }  
}
```

1. Build the project to generate an assembly (.dll) file. Make sure this is built against the same version of the .NET as SpecFlow, and copy this file to your SpecFlow installation directory.
2. Add a config item to your CodedUI project's App.Config file

```
<?xml version="1.0" encoding="utf-8" ?>  
<configuration>
```

(continues on next page)

(continued from previous page)

```

<configSections>
  <section name="specFlow"
    type="TechTalk.SpecFlow.Configuration.ConfigurationSectionHandler,
TechTalk.SpecFlow"/>
</configSections>
<specFlow>
  <unitTestProvider name="MsTest.2010"
    generatorProvider="My.SpecFlow.MsTest2010CodedUiGeneratorProvider,
My.SpecFlow"
    runtimeProvider="TechTalk.SpecFlow.UnitTestProvider.MsTest2010RuntimeProvider,
TechTalk.SpecFlow"/>
</specFlow>
</configuration>

```

1. Now when you generate a new feature file, it will add the appropriate attributes.

Getting SpecFlow to generate the [CodedUITest] attribute with Visual Studio 2013+ and MSTest

1. Create a new Class Library project in Visual Studio (example: TechTalk.SpecFlow.CodedUI.MsTest).
2. Install the SpecFlow NuGet package via the Package Manager Console.
3. Create a new Class called SpecFlowCodedUITestGenerator.
4. Right click the Project in the Solution Explorer pane.
5. Click "Add..." then click "References...".
6. Add a reference to the following DLLs:
 - <Solution Directory>\packages\SpecFlow.X.Y.Z\tools\TechTalk.SpecFlow.Generator.dll
 - <Solution Directory>\packages\SpecFlow.X.Y.Z\tools\TechTalk.SpecFlow.Utills.dll
1. Copy the following code to the SpecFlowCodedUITestGenerator class:

```

using System.CodeDom;
using TechTalk.SpecFlow.Generator.UnitTestProvider;
using TechTalk.SpecFlow.Utills;

namespace TechTalk.SpecFlow.CodedUI.MsTest
{
    public class SpecFlowCodedUITestGenerator : MsTestGeneratorProvider
    {
        public SpecFlowCodedUITestGenerator(CodeDomHelper codeDomHelper) :
        ↪base(codeDomHelper)
        {
        }

        public override void SetTestClass(TechTalk.SpecFlow.Generator.
        ↪TestClassGenerationContext generationContext, string featureTitle, string
        ↪featureDescription)
        {
            base.SetTestClass(generationContext, featureTitle, featureDescription);
        }
    }
}

```

(continues on next page)

(continued from previous page)

```

        foreach (CodeAttributeDeclaration customAttribute in generationContext.
↳TestClass.CustomAttributes)
        {
            if (customAttribute.Name == "Microsoft.VisualStudio.TestTools.
↳UnitTesting.TestClassAttribute")
            {
                generationContext.TestClass.CustomAttributes.Remove(customAttribute);
                break;
            }
        }

        generationContext.TestClass.CustomAttributes.Add(new
↳CodeAttributeDeclaration(new CodeTypeReference("Microsoft.VisualStudio.TestTools.
↳UnitTesting.CodedUITestAttribute")));
    }
}

```

1. Right-click the Project in the Solution Explorer pane, and click "Properties".
2. Go to the "Build Events" tab.
3. In the "Post-build event command line" box, enter the following command:

```
copy $(TargetPath) $(SolutionDir)packages\SpecFlow.X.Y.Z\tools\
```

****Important!**** The DLL created by building the `TechTalk.SpecFlow.CodedUI.MsTest`
↳project needs to be copied to the `packages\SpecFlow.X.Y.Z\tools` directory of the
↳Visual Studio solution that contains your SpecFlow tests in order for this to work.

1. In the "Properties" for the TechTalk.SpecFlow.CodedUI.MsTest project, go to the "Application" tab
2. Choose ".NET Framework 3.5" for SpecFlow 1.9 or ".NET Framework 4.5" for SpecFlow 2.0+ in the "Target framework" drop-down.
3. Change the <unitTestProvider> in App.config to use the new test generator:

```

<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <specFlow>
    <unitTestProvider name="MsTest"
      generatorProvider="TechTalk.SpecFlow.CodedUI.MsTest.SpecFlowCodedUITestGenerator,
↳TechTalk.SpecFlow.CodedUI.MsTest"
      runtimeProvider="TechTalk.SpecFlow.UnitTestProvider.MsTestRuntimeProvider,
↳TechTalk.SpecFlow" />
    </specFlow>
  </configuration>

```

If Visual Studio prompts you to regenerate the feature files, do so. If not, right-click on the project containing your SpecFlow tests and click "Regenerate Feature Files".

Ensuring the SpecFlow hooks can use the CodedUI API

If you want to use any of the SpecFlow Hooks as steps such as [BeforeTestRun],[BeforeFeature], [BeforeScenario], [AfterTestRun], [AfterFeature] or [AfterScenario], you will receive the following error: `Microsoft.VisualStudio.TestTools.UITest.Extension.TechnologyNotSupportedException: The browser is currently not supported`

Solve this by adding a `Playback.Initialize();` call in your [BeforeTestRun] step, and a `Playback.Cleanup();` in your [AfterTestRun] step.

2.75.3 Other Information

[Blog series on using Specflow with Coded UI Test API] (<http://rburnham.wordpress.com/2011/03/15/bdd-ui-automation-with-specflow-and-coded-ui-tests/>)

2.76 Upgrade from SpecFlow 2.* to 3.*

This guide explains how to update your SpecFlow 2.* project to the latest SpecFlow 3.* version

2.76.1 Make a Backup!

Before upgrading to the latest version, ensure you have a backup of your project (either locally or in a source control system).

2.76.2 Visual Studio Integration

The Visual Studio integration for SpecFlow has been updated for SpecFlow 3. You will need to update the extension in order to upgrade. If you previously set the extension to not update automatically, please enable automatic upgrades once your projects have been migrated to SpecFlow 2.3.2 or higher.

2.76.3 App.config Deprecated

Changes to How Unit Test Providers are Configured

In previous versions of SpecFlow, the unit test provider used to execute tests was configured in your app.config file. As of SpecFlow 3, you need to configure your unit test provider by installing one of the available packages (see below).

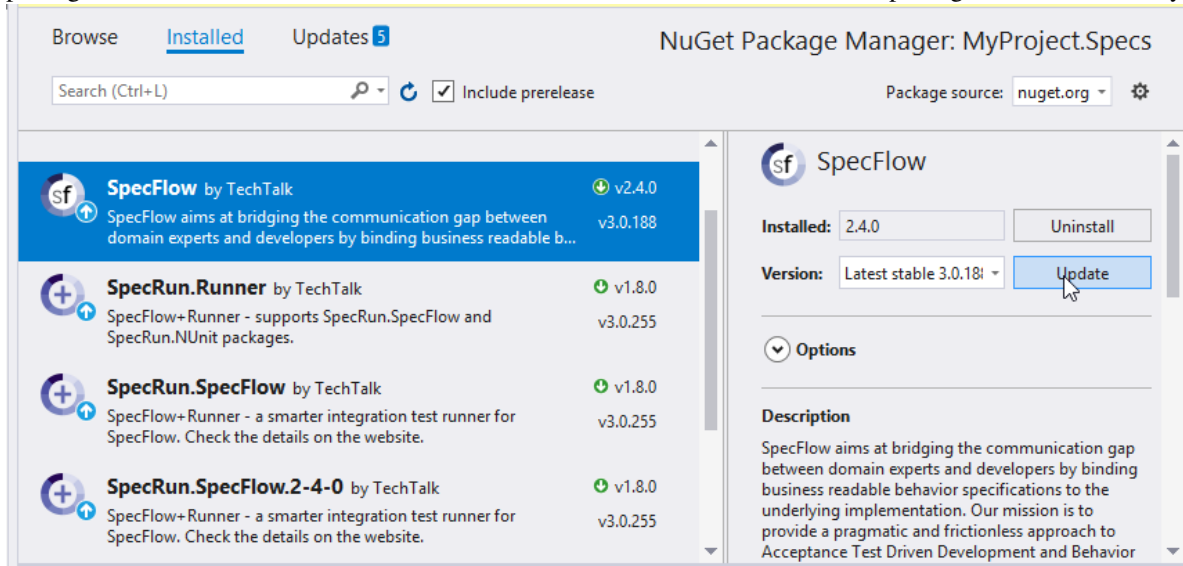
specflow.json

Moving forward, we recommend using specflow.json to configure SpecFlow, rather than app.config. .NET Core projects require specflow.json (app.config is not supported). While using specflow.json is optional for Full Framework projects, we recommend migrating to the new format. For more details, see [Configuration](#) in the documentation.

2.76.4 Updating SpecFlow

To upgrade a solution from SpecFlow 2.x to SpecFlow 3:

1. Open your solution, and check that it compiles, all tests are discovered and that all source files have been committed.
2. Right-click on your solution and select Manage NuGet Packages for Solution.
3. Switch to Updates in the list on the left and locate SpecFlow in the list of packages. Use the search box to restrict the listed packages if necessary.



4. Select the SpecFlow package in the list and click on Update.
5. Add one of the following packages to your specifications project (the one containing your tests) to select your unit test provider. You will receive an error if you add more than one of these packages to your project:
 - SpecRun.SpecFlow
 - SpecFlow.xUnit
 - SpecFlow.MsTest
 - SpecFlow.NUnit
6. Remove "SpecFlowSingleFileGenerator" from the Custom Tool field in the Properties of your feature files.

2.76.5 Updating SpecFlow+ Runner

If you want to update both SpecFlow and SpecFlow+ Runner to version 3, the easiest way to do this is to simply upgrade the SpecRun package. This automatically updates SpecFlow as well.

To update SpecFlow and SpecFlow+ Runner:

1. Open your solution, and check that it compiles, all tests are discovered and that all source files have been committed.
2. Right-click on your solution and select Manage NuGet Packages for Solution.
3. Uninstall any SpecRun.SpecFlow.* packages you have installed.
4. Install/update the following packages:

- SpecFlow
- SpecRun.SpecFlow

5. Remove “SpecFlowSingleFileGenerator” from the Custom Tool field in the Properties of your feature files.

SpecFlow+ Runner Report Templates

If you have customized the SpecFlow+ runner templates, a small change needs to be made to the template for SpecFlow 3:

Open the CSHTML file in the editor of your choice. Replace the first line with the following:

```
@inherits SpecFlow.Plus.Runner.Reporting.CustomTemplateBase<TestRunResult>
```

2.77 Tools

2.77.1 Removed in SpecFlow 3

Note that this tool has been removed in SpecFlow 3. This topic only applies to earlier versions of SpecFlow.

2.77.2 Tools Overview

Besides executing your tests and verifying that acceptance criteria are met, SpecFlow provides a number of other tools to support the development process. These tools are accessed using the `specflow.exe` command line tool located in the `/packages/SpecFlow.x.y.a/tools` directory of your project.

A number of commands are available. If you start `specflow.exe` without any additional options, the available commands are listed.

2.77.3 help

Use the `help` command to display the detailed information on a specific command's syntax. For example, `specflow.exe help generateall` displays the options available for the `generateall` command.

2.77.4 generateall

Use the `generateall` command to re-generate outdated unit test classes based on your feature files. This can be useful when upgrading to a newer SpecFlow version, or if feature files are modified outside of Visual Studio.

The following arguments are available.

The following command line regenerates unit tests for the BookShop sample application.

```
specflow.exe generateall --ProjectFile BookShop.AcceptanceTests.csproj
```

SpecFlow's `generateall` function can also be used with MSBuild, allowing you to update the files before compiling the solution. This can be particularly useful if feature files are regularly modified outside of Visual Studio. See [Generate Tests from MsBuild](#) for details on configuring MSBuild to use the `generateall` function.

2.77.5 Generating Reports

Please note that the following reports are only supported prior to SpecFlow 3. As of SpecFlow 3, these options are no longer available.

stepdefinitionreport

Generates a report that includes information on the usage and binding status of the steps in the entire project. For more details, see [Reporting](#).

nunitexecutionreport

Generates an execution report based on the output of the NUnit Console Runner. For more details, see [Reporting](#).

mstestexecutionreport

Generates an execution report based on the TRX output generated by VSTest. For more details, see [Reporting](#).

2.78 Troubleshooting

2.78.1 Cannot find custom tool SpecFlowSingleFileGenerator

If Visual Studio displays the error message Cannot find custom tool 'SpecFlowSingleFileGenerator' on this system. when right-clicking on a feature file and selecting Run Custom Tool, make sure the SpecFlow extension is installed and enabled.

To enable the extension in Visual Studio, select **Tools | Extensions and Updates...**, select the “SpecFlow for Visual Studio” extension, then select **Enable**.

2.78.2 Enabling Tracing

You can enable traces for SpecFlow. Once tracing is enabled, a new SpecFlow pane is added to the output window showing diagnostic messages.

To enable tracing, select **Tools | Options | SpecFlow** from the menu in Visual Studio and set **Enable Tracing** to ‘True’.

Steps are not recognised even though there are matching step definitions

The SpecFlow Visual Studio integration caches the binding status of step definitions. If the cache is corrupted, steps may be unrecognised and the highlighting of your steps may be wrong (e.g. bound steps showing as being unbound). To delete the cache:

1. Close all Visual Studio instances.
2. Navigate to your %TEMP% folder and delete any files that are prefixed with specflow-stepmap-, e.g. specflow-stepmap-SpecFlowProject-607539109-73a67da9-ef3b-45fd-9a24-6ee0135b5f5c.cache.
3. Reopen your solution.

You may receive a more specific error message if you enable tracing (see above).

Tests are not displayed in the Test Explorer window when using SpecFlow+ Runner

Note: As of Visual Studio 2017 15.7 the temporary files are no longer used. The following only applies to earlier versions of Visual Studio.

The Visual Studio Test Adapter cache may also get corrupted, causing tests to not be displayed. If this happens, try clearing your cache as follows:

1. Close all Visual Studio instances
2. Navigate to your %TEMP%\VisualStudioTestExplorerExtensions\ folder and delete any sub-folders related to SpecFlow/SpecRun, i.e. that have "SpecFlow" or "SpecRun" in their name.
3. Reopen your solution and ensure that it builds.

Unable to find plugin in the plugin search path: SpecRun` when saving / generating feature files

SpecFlow searches for plugins in the NuGet packages folder. This is detected relative to the reference to TechTalk.SpecFlow.dll. If this DLL is not loaded from the NuGet folder, the plugins will not be found.

A common problem is that the NuGet folder is not yet ready (e.g. not restored) when opening the solution, but TechTalk.SpecFlow.dll is located in the bin\Debug folder of the project. In this case, Visual Studio may load the assembly from the bin\Debug folder instead of waiting for the NuGet folder to be properly restored. Once this has happened, Visual Studio remembers that it loaded the assembly from bin\Debug, so reopening the solution may not solve this issue. The best way to fix this issue is as follows:

1. Make sure the NuGet folders are properly restored.
2. Close Visual Studio.
3. Delete the bin\Debug folder from your project(s).
4. Reopen your solution in Visual Studio.

Tests are not displayed in the Test Explorer window when using SpecFlow+ Runner, even after restoring the NuGet package

The SpecRun.Runner NuGet package that contains the Visual Studio Test Explorer adapter is a solution-level package (registered in the .nuget\packages.config file of the solution). In some situations, NuGet package restore on build does not restore solution-level packages.

To fix this, open the NuGet console or the NuGet references dialog and click on the restore packages button. You may need to restart Visual Studio after restoring the packages.

VS2015: Tests are not displayed in the Test Explorer window when using SpecFlow+ Runner

It seems that VS2015 handles solution-level NuGet packages differently (those registered in the .nuget\packages.config file of the solution). As a result, solution-level NuGet packages must be listed in the projects that use them, otherwise Test Explorer cannot recognise the test runner.

To fix this issue, either re-install the SpecFlow+ Runner NuGet packages, or add the dependency on the SpecRun.Runner package (`<package id="SpecRun.Runner" version="1.2.0" />`) to the packages.config file of your SpecFlow projects. You might need to restart Visual Studio to see your tests.

2.78.3 When trying to run my tests in Visual Studio, I receive a Missing [assembly:GeneratorPlugin] attribute error. How can I solve this?

Sample output:

```
Missing [assembly:GeneratorPlugin] attribute in SpecFlow.Plus.Excel.SpecFlowPlugin.dll
#error TechTalk.SpecFlow.Generator
#error Server stack trace:
#error at TechTalk.SpecFlow.Generator.Plugins.GeneratorPluginLoader.
↪LoadPlugin(PluginDescriptor pluginDescriptor)
...
```

If you are receiving this error, try setting the **Generation Mode** in SpecFlow to “OutOfProcess”. To do so:

1. Select **Tools | Options** from the menu in Visual Studio.
2. Select SpecFlow from the list on the left.
3. Locate the **Generation Mode** setting and set it to “OutOfProcess”.

2.78.4 After upgrading to SpecFlow 2 from 1.9, I get the message “Trace listener failed. -> The ScenarioContext.Current static accessor cannot be used in multi-threaded execution. Try injecting the scenario context to the binding class”

Make sure you have regenerated the .feature.cs files after upgrading. If you do not do this, you will receive this exception when accessing ScenarioContext.Current.

To regenerate these files:

- Open a feature file in your solution. If you see a popup informing you that the feature files were generated with an earlier version of SpecFlow, click on **Yes** to regenerate these files. Depending on the size of your project, this may take a while.
- If you are using an earlier version of Visual Studio, you need to force the feature files to be regenerated. Right-click on your project, and select **Regenerate Feature Files** from the menu.

2.78.5 Build error due to using spaces and special characters in project name

Using special characters in your project name will cause build errors as the feature files will fail to generate properly.

The build error may look like this:

```
Build started...
1>----- Build started: Project: My proj (new), Configuration: Debug Any CPU -----
1>SpecFlowFeatureFiles: Features\Calculator.feature
1>SpecFlowGeneratedFiles: Features\Calculator.feature.cs
1>C:\Work\repos\My proj\My proj (new)\obj\Debug\netcoreapp3.1\NUnit.AssemblyHooks.cs(12,
↪22,12,23): error CS1514: { expected
1>C:\Work\repos\My proj\My proj (new)\obj\Debug\netcoreapp3.1\NUnit.AssemblyHooks.cs(12,
↪22,12,23): error CS1513: } expected
1>C:\Work\repos\My proj\My proj (new)\obj\Debug\netcoreapp3.1\NUnit.AssemblyHooks.cs(12,
↪22,12,27): error CS8400: Feature 'top-level statements' is not available in C# 8.0.
↪Please use language version 9.0 or greater.
1>C:\Work\repos\My proj\My proj (new)\obj\Debug\netcoreapp3.1\NUnit.AssemblyHooks.cs(12,
↪22,12,27): error CS8803: Top-level statements must precede namespace and type
↪declarations.
```

(continued from previous page)

```

1>C:\Work\repros\My proj\My proj (new)\obj\Debug\netcoreapp3.1\NUnit.AssemblyHooks.cs(12,
↪26,12,27): error CS1526: A new expression requires an argument list or (), [], or {}_
↪after type

```

The most obvious solution to this is to avoid using **special characters e.g. (paranthesis)** for your project name. Spaces are ok as they get replaced with underlines_.

Example

If you have already created your project with special characters you can still change that. Lets take a look at an example. The below project name was created as *****My proj (new)*****.

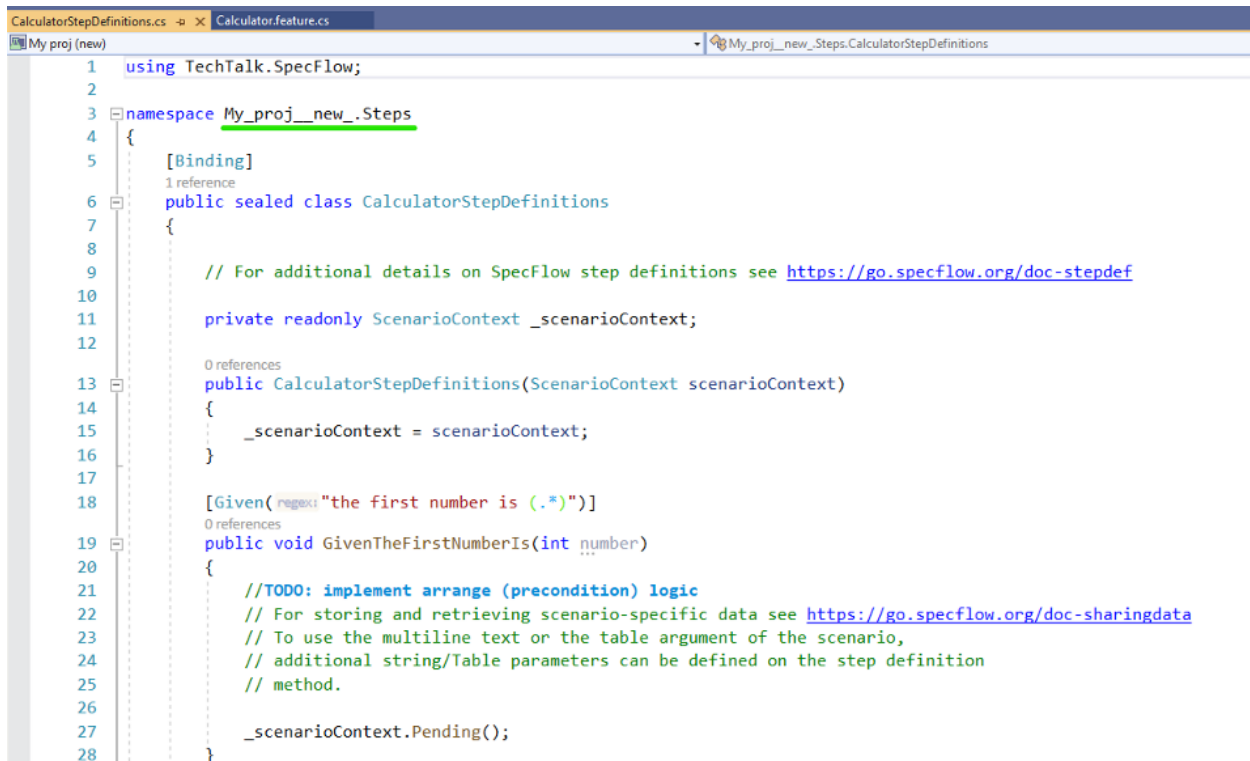
- If you open the generated code you can find that although the spaces have been replaced with underscore characters the parentheses are kept in the namespace causing a compilation error. Visual Studio also highlights this invalid code.

```

1 // -----
2 // <auto-generated>
3 //   This code was generated by SpecFlow (https://www.specflow.org/).
4 //   SpecFlow Version:3.9.0.0
5 //   SpecFlow Generator Version:3.9.0.0
6 //
7 //   Changes to this file may cause incorrect behavior and will be lost if
8 //   the code is regenerated.
9 // </auto-generated>
10 // -----
11 #region Designer generated code
12 #pragma warning disable
13 namespace My_proj_(new).Features
14 {
15     using TechTalk.SpecFlow;
16     using System;
17     using System.Linq;
18
19
20     [System.CodeDom.Compiler.GeneratedCodeAttribute("TechTalk.SpecFlow", "3.9.0.0")]
21     [System.Runtime.CompilerServices.CompilerGeneratedAttribute()]
22     [NUnit.Framework.TestFixtureAttribute()]
23     [NUnit.Framework.DescriptionAttribute("Calculator")]
24     public partial class CalculatorFeature_
25     {
26
27         private TechTalk.SpecFlow.ITestRunner testRunner;
28
29         private string[] _featureTags = ((string[])(null));

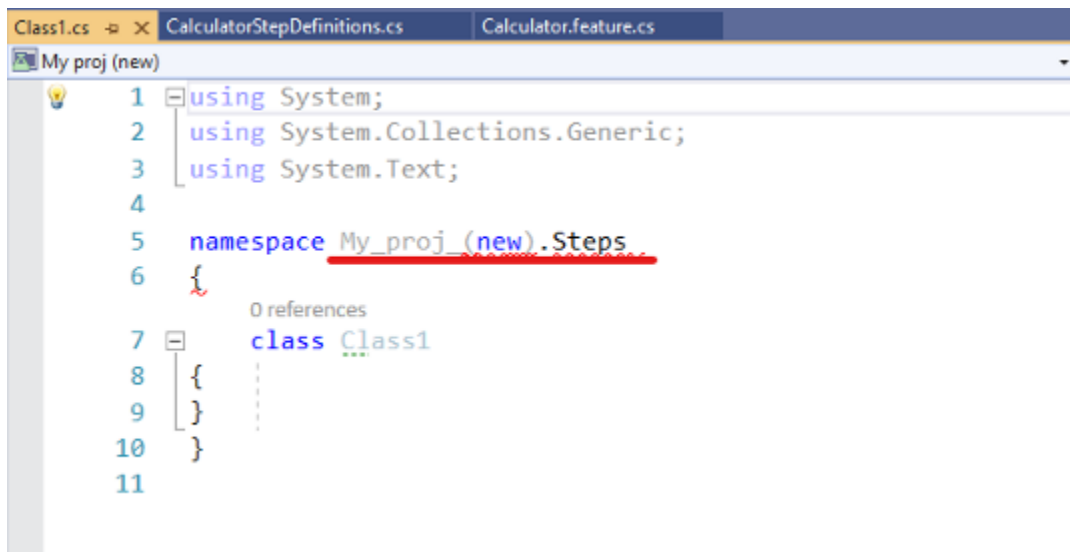
```

- If you use the SpecFlow project template there is also a binding class added already to the project. In the C# file the generated namespace is correct, because the parentheses are also automatically replaced with underscore:



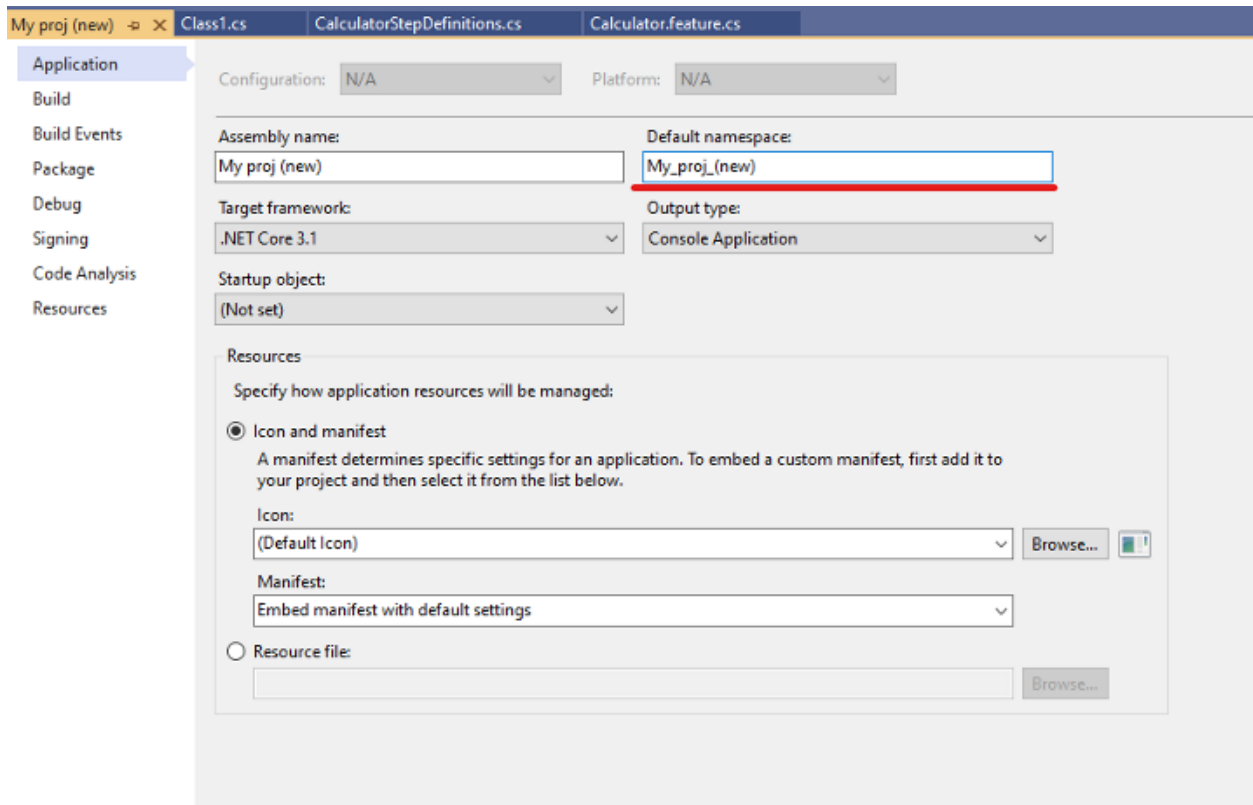
```
1 using TechTalk.SpecFlow;
2
3 namespace My_proj_new.Steps
4 {
5     [Binding]
6     public sealed class CalculatorStepDefinitions
7     {
8         // For additional details on SpecFlow step definitions see https://go.specflow.org/doc-stepdef
9
10        private readonly ScenarioContext _scenarioContext;
11
12        public CalculatorStepDefinitions(ScenarioContext scenarioContext)
13        {
14            _scenarioContext = scenarioContext;
15        }
16
17        [Given("the first number is (.*)")]
18        public void GivenTheFirstNumberIs(int number)
19        {
20            //TODO: implement arrange (precondition) logic
21            // For storing and retrieving scenario-specific data see https://go.specflow.org/doc-sharingdata
22            // To use the multiline text or the table argument of the scenario,
23            // additional string/Table parameters can be defined on the step definition
24            // method.
25
26            _scenarioContext.Pending();
27        }
28    }
```

- However, you can see that the project is generally in an “unhealthy state” if you try to add a new C# file from Visual Studio. The new file will have a similar invalid namespace like the SpecFlow generated code:



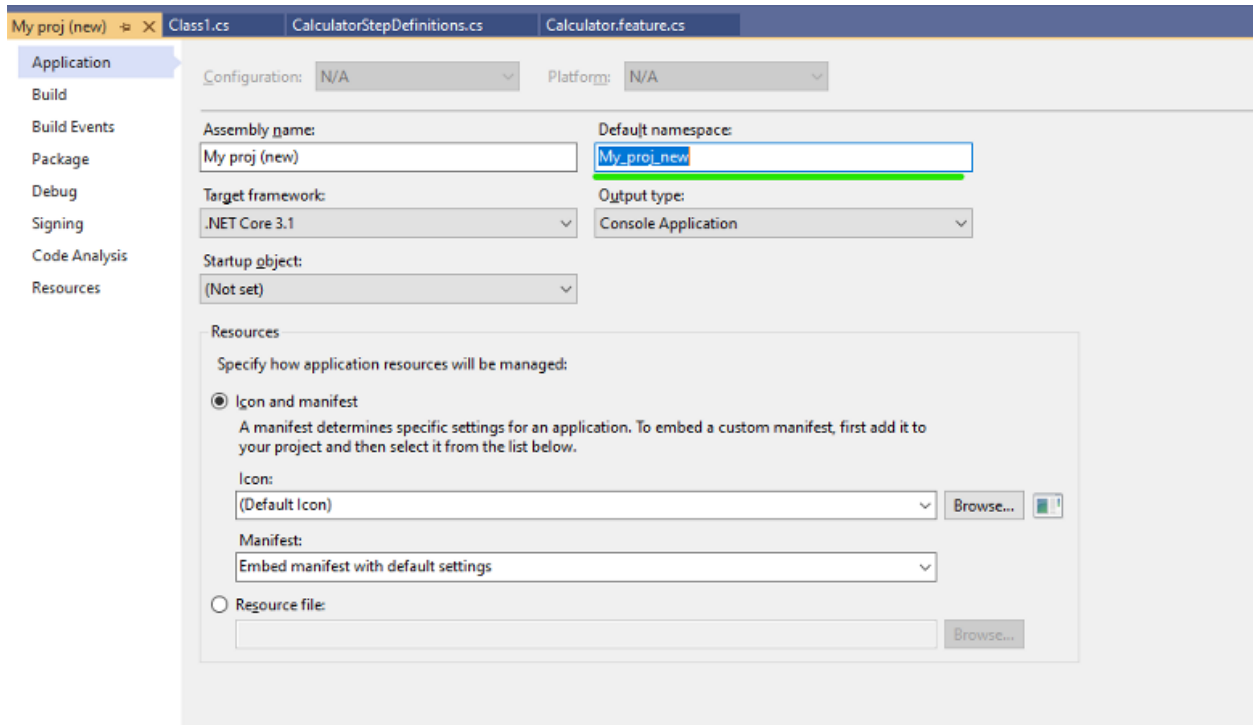
```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace My_proj_(new).Steps
6 {
7     class Class1
8     {
9     }
10 }
11
```

- This is caused by the default namespace of the project (go to project properties in Visual Studio). By default the root namespace equals to the project name (equals to the assembly name). However, while the parenthesis are valid in the project name and in the assembly name, these characters are not valid in the namespace. Note that the spaces are automatically replaced with underscore, but the parentheses are not:

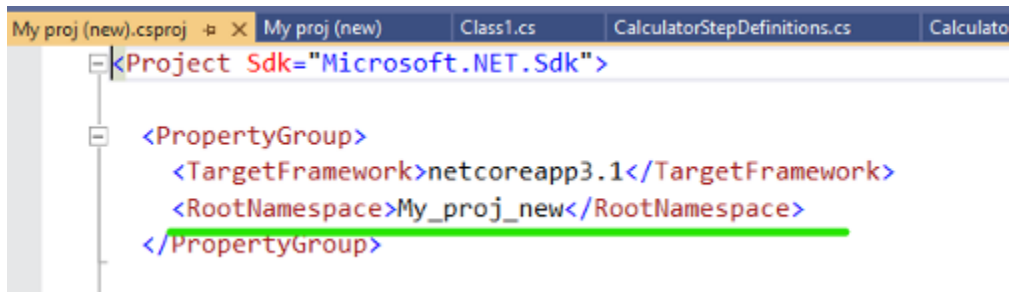


Solution

- To fix the project the default namespace should be changed to a valid namespace:



- You can also fix the issue in the csproj file directly. The property “RootNamespace” has to be set accordingly:

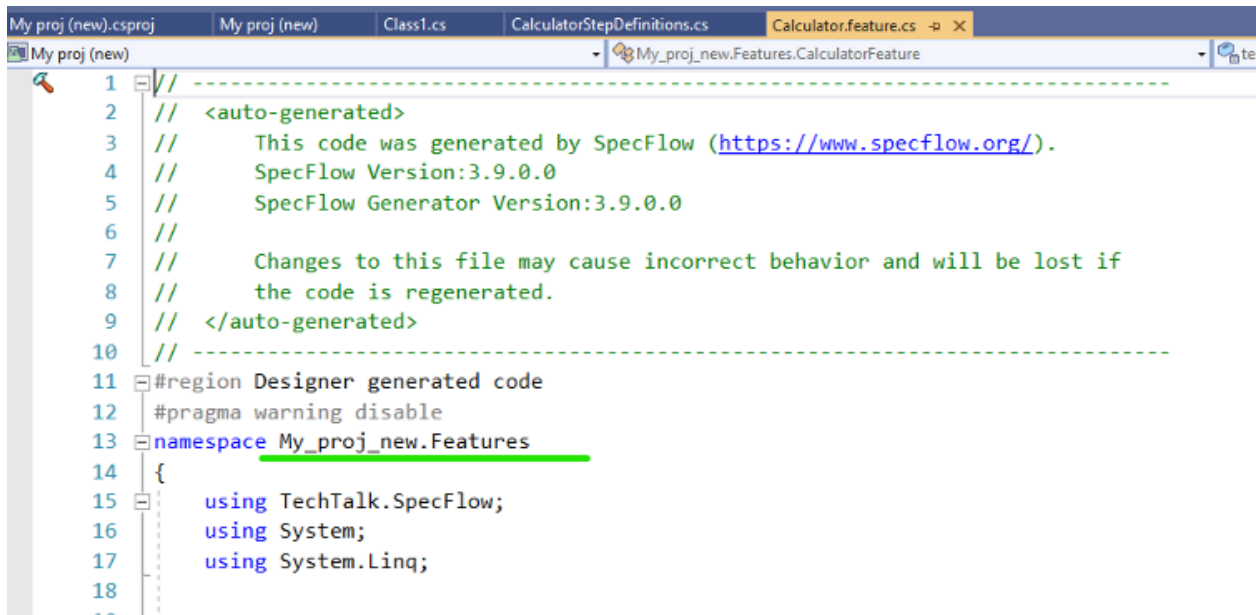


```
My proj (new).csproj  My proj (new)  Class1.cs  CalculatorStepDefinitions.cs  Calculator

<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
    <RootNamespace>My_proj_new</RootNamespace>
  </PropertyGroup>
```

- The default namespace will be applied by SpecFlow code generation and the generated classes will be valid now:



```
My proj (new).csproj  My proj (new)  Class1.cs  CalculatorStepDefinitions.cs  CalculatorFeature.cs  Calculator

My proj (new)  My_proj_new.Features.CalculatorFeature

1 // -----
2 // <auto-generated>
3 //   This code was generated by SpecFlow (<a href="https://www.specflow.org/">https://www.specflow.org/)</a>).
4 //   SpecFlow Version:3.9.0.0
5 //   SpecFlow Generator Version:3.9.0.0
6 //
7 //   Changes to this file may cause incorrect behavior and will be lost if
8 //   the code is regenerated.
9 // </auto-generated>
10 // -----
11 #region Designer generated code
12 #pragma warning disable
13 namespace My_proj_new.Features
14 {
15     using TechTalk.SpecFlow;
16     using System;
17     using System.Linq;
18 }
```

> **Note:** if you added a C# class with the invalid default namespace you have to fix that file manually. This is the standard behavior of the default namespace, it will be used for newly created files only. The SpecFlow generated classes get however automatically fixed, because they are (re-)generated during the build.

2.79 Known Issue

2.79.1 Using the “Rule” Gherkin keyword breaks syntax highlighting in Visual Studio

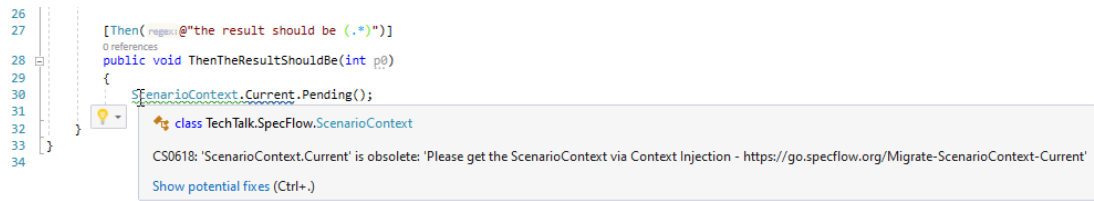
The Visual Studio extension does not yet support the “Rule” Gherkin keyword. Therefore, using this keyword will stop syntax highlighting from working in Visual Studio. Syntax highlighting for the “Rule” keyword will be added in a future release.

2.79.2 Generating step definitions provides deprecated code

The “*Generate Step Definitions*” command generates obsolete code in Visual Studio. E.g.:

```
[Then(regex:@"the result should be (.*)")]
0 references
public void ThenTheResultShouldBe(int p0)
{
    ScenarioContext.Current.Pending();
}
```

A warning is shown in Visual Studio:



The generation of obsolete code will be removed in a future release.

2.80 Troubleshooting Visual Studio Integration

2.80.1 Conflicts with Deveroom

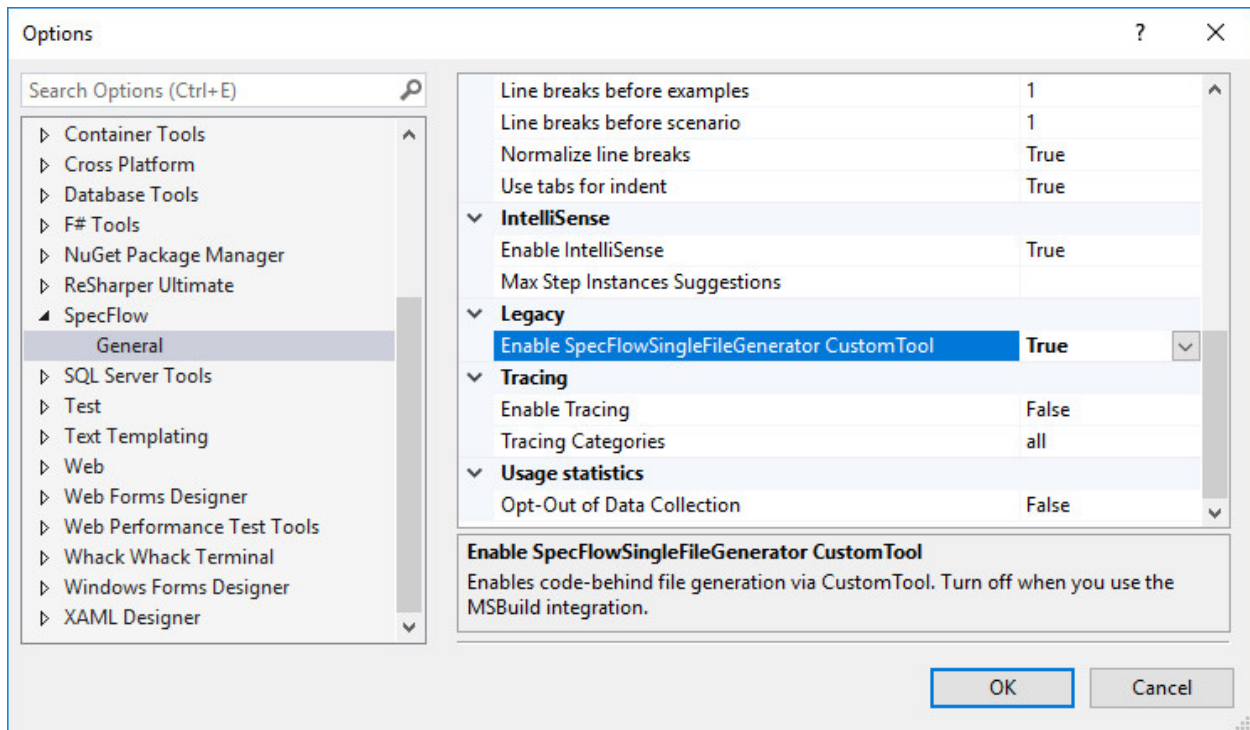
If you are using Deveroom, do not install the SpecFlow Visual Studio extension; you should only install one of these 2 extensions.

2.80.2 Enable Extension

If Visual Studio displays the error message Cannot find custom tool 'SpecFlowSingleFileGenerator' on this system. when right-clicking on a feature file and selecting Run Custom Tool, make sure the SpecFlow extension is enabled.

To enable the extension in Visual Studio, select **Tools | Extensions and Updates...**, select the “SpecFlow for Visual Studio” extension, then select **Enable**.

If the error still occurs, select **Tools | Options | SpecFlow** and set **Enable SpecFlowSingleFileGenerator Custom Tool** to ‘True’.



2.80.3 Enable Tracing

You can enable traces for SpecFlow. Once tracing is enabled, a new SpecFlow pane is added to the output window showing diagnostic messages.

To enable tracing, select **Tools | Options | SpecFlow** from the menu in Visual Studio and set **Enable Tracing** to 'True'.

2.80.4 Troubleshooting

Steps are not recognised even though there are matching step definitions

The SpecFlow Visual Studio integration caches the binding status of step definitions. If the cache is corrupted, steps may be unrecognised and the highlighting of your steps may be wrong (e.g. bound steps showing as being unbound). To delete the cache:

1. Close all Visual Studio instances.
2. Navigate to your %TEMP% folder and delete any files that are prefixed with specflow-stepmap-, e.g. specflow-stepmap-SpecFlowProject-607539109-73a67da9-ef3b-45fd-9a24-6ee0135b5f5c.cache.
3. Reopen your solution.

You may receive a more specific error message if you enable tracing (see above).

Tests are not displayed in the Test Explorer window when using SpecFlow+ Runner

Note: As of Visual Studio 2017 15.7 the temporary files are no longer used. The following only applies to earlier versions of Visual Studio.

The Visual Studio Test Adapter cache may also get corrupted, causing tests to not be displayed. If this happens, try clearing your cache as follows:

1. Close all Visual Studio instances
2. Navigate to your %TEMP%\VisualStudioTestExplorerExtensions\ folder and delete any sub-folders related to SpecFlow/SpecRun, i.e. that have "SpecFlow" or "SpecRun" in their name.
3. Reopen your solution and ensure that it builds.

Unable to find plugin in the plugin search path: SpecRun` when saving / generating feature files

SpecFlow searches for plugins in the NuGet packages folder. This is detected relative to the reference to TechTalk.SpecFlow.dll. If this DLL is not loaded from the NuGet folder, the plugins will not be found.

A common problem is that the NuGet folder is not yet ready (e.g. not restored) when opening the solution, but TechTalk.SpecFlow.dll is located in the bin\Debug folder of the project. In this case, Visual Studio may load the assembly from the bin\Debug folder instead of waiting for the NuGet folder to be properly restored. Once this has happened, Visual Studio remembers that it loaded the assembly from bin\Debug, so reopening the solution may not solve this issue. The best way to fix this issue is as follows:

1. Make sure the NuGet folders are properly restored.
2. Close Visual Studio.
3. Delete the bin\Debug folder from your project(s).
4. Reopen your solution in Visual Studio.

Tests are not displayed in the Test Explorer window when using SpecFlow+ Runner, even after restoring the NuGet package

The SpecRun.Runner NuGet package that contains the Visual Studio Test Explorer adapter is a solution-level package (registered in the .nuget\packages.config file of the solution). In some situations, NuGet package restore on build does not restore solution-level packages.

To fix this, open the NuGet console or the NuGet references dialog and click on the restore packages button. You may need to restart Visual Studio after restoring the packages.

VS2015: Tests are not displayed in the Test Explorer window when using SpecFlow+ Runner

It seems that VS2015 handles solution-level NuGet packages differently (those registered in the .nuget\packages.config file of the solution). As a result, solution-level NuGet packages must be listed in the projects that use them, otherwise Test Explorer cannot recognise the test runner.

To fix this issue, either re-install the SpecFlow+ Runner NuGet packages, or add the dependency on the SpecRun.Runner package (`<package id="SpecRun.Runner" version="1.2.0" />`) to the packages.config file of your SpecFlow projects. You might need to restart Visual Studio to see your tests.

2.81 Usage Analytics

In order to improve the quality of SpecFlow and understand how it is used, we collect anonymous usage data via Azure Application Insights, an analytics platform provided by Microsoft. We do not collect any personally identifiable information, but information such as the Visual Studio version, operating system, MSBuild version, target frameworks etc.

For more details on the information collected by Application Insights, see our [privacy policy](#).

You can disable these analytics as follows:

- Select **Tools | Options** from the menu in Visual Studio, and navigate to **SpecFlow* in the list on the left. Set **Opt-Out of Data Collection** to “True”. This disables analytics collected by the Visual Studio extension (see “SpecFlow Visual Studio Extension” in the [privacy policy](#) for details).
- Define an environment variable called `SPECFLOW_TELEMETRY_ENABLED` and set its value to 0. This disables all analytics, i.e. those collected by both the extension and SpecFlow itself (see “SpecFlow” and “SpecFlow Visual Studio Extension” in the [privacy policy](#) for details)