**Specflow Tutorial: The Ultimate Guide to BDD Tool**

**What is Specflow?**

***Specflow is a testing framework supporting BDD practices in .NET framework. It’s an open source framework hosted on GitHub.***

*It aids in using ATDD (Acceptance test driver development) for .NET Applications. With this, we can define scenario in plain English defined by Gherkin language that is clearly understandable by anyone.*

*There are various tools for writing tests in the BDD approach like Cucumber/JBehave for Java, Lettuce for Python, Jasmine for Javascript,****Specflow****for .NET.*

**BDD** **(Behavior Driven Development) is a set of practices or an approach similar to TDD (Test Driven Development), which aims to bridge the communication gap between different stakeholders like Product, Developers, and Testers.**

**The end goal of the BDD approach is to create business requirements that could be understood by the entire team so as to avoid misunderstandings, and helps to ship the feature being developed in the most acceptable way**.

**Features of BDD**

**The key features of BDD are briefed below:**

**#1)** **It tries to define the behavior of the system or feature being developed through an example or scenario**. For instance, if you are building a simple Calculator Application then the different behaviors include addition, multiplication, division, etc.

Hence through BDD, all the stakeholders will first meet to decide the behavior of the application like Addition and will have scenarios as shown below.

|  |  |
| --- | --- |
| 1  2  3 | Given, I have 2 numbers 30 and 50 as input  When I add these 2 numbers  Then I should get an output of 80 |

If you see the above representation it’s a scenario in plain English that is clearly understandable by anyone and makes the requirements for a feature clear (as per the acceptance criteria). Hence the first step is to articulate these requirements.

**#2)** Now with a set of these scenarios, the QA writes tests against these and this will initially fail as the feature is not yet developed.

**#3)** Now, the developer writes a feature code and executes these tests again.

**#4)** The tests may pass or fail. If they fail – refactor code and repeat the process

**#5)** Once code refactoring is complete all the scenarios/tests should pass.

Hence, in essence, BDD uses TDD approach and takes it to the next level by having some common easily understandable specifications in the form of scenarios. They also represent the feature documentation in itself.

**The Keywords – Given, When & Then**

**From the unit testing world, most of us are familiar with 3 A’s i.e. Arrange, Act and Assert. Now, Given, When and Then are the replacements for these in the BDD world**.

Let’s take an **Example** for understanding each of these. Suppose you are listing down a scenario for validating a product that gets added to the shopping cart of an e-commerce application which requires you to be logged in as a pre-requisite.

**The specification can be written as follows:**

|  |  |
| --- | --- |
| 1  2  3  4 | Scenario: Products get added to cart for a logged in customer  Given I have a logged-in customer on my application  When I add 2 quantity of a product to my shopping cart  Then the shopping cart should get updated and have the right product and quantity |

**Given**: **This is used for describing a set of pre-conditions for the scenario being defined.** For instance, in the example, the scenario’s pre-requisite is a logged-in customer. Hence comparing to the *Arrange* analogy in a unit test, the step implementation will need to ensure that there is a logged in customer.

**When**: **This is used to describe an action or execution step**. In the example, it shows that the customer is trying to add a product to his shopping cart. Hence the step implementation for this step will take care of the simulation code to add a product to the cart. This can be compared to the *Act*step in the Unit tests.

**Then**: **This is used to describe the Outcome of the scenario and essentially where the validations should be placed in**. It can be compared to the *Assert* step in the Unit testing world. In the example here, the step implementation will assert whether the product got actually added and the quantity is the same as that was chosen by the customer.

**The Feature File**

**The feature file is essentially a grouping of multiple scenarios for the application under development or test**. It can also be simply thought of as different modules of the application by which the application can be logically separated.

**For Example:**

An e-commerce application can decide to have different high-level feature files like:

* Login/Logout functionality
* Shopping Cart
* Payment etc.

**What is Specflow?**

**Specflow is a tool supporting BDD practices in .NET framework. It’s an open source framework hosted on GitHub. It aids in using ATDD (Acceptance test driver development) for .NET Applications.**

It makes use of [Gherkin](https://cucumber.io/docs/gherkin/) syntax for creating features & scenarios. It also has an active discussion/developer [forum](https://groups.google.com/forum/#!forum/SpecFlow).

**Specflow – Getting Started**

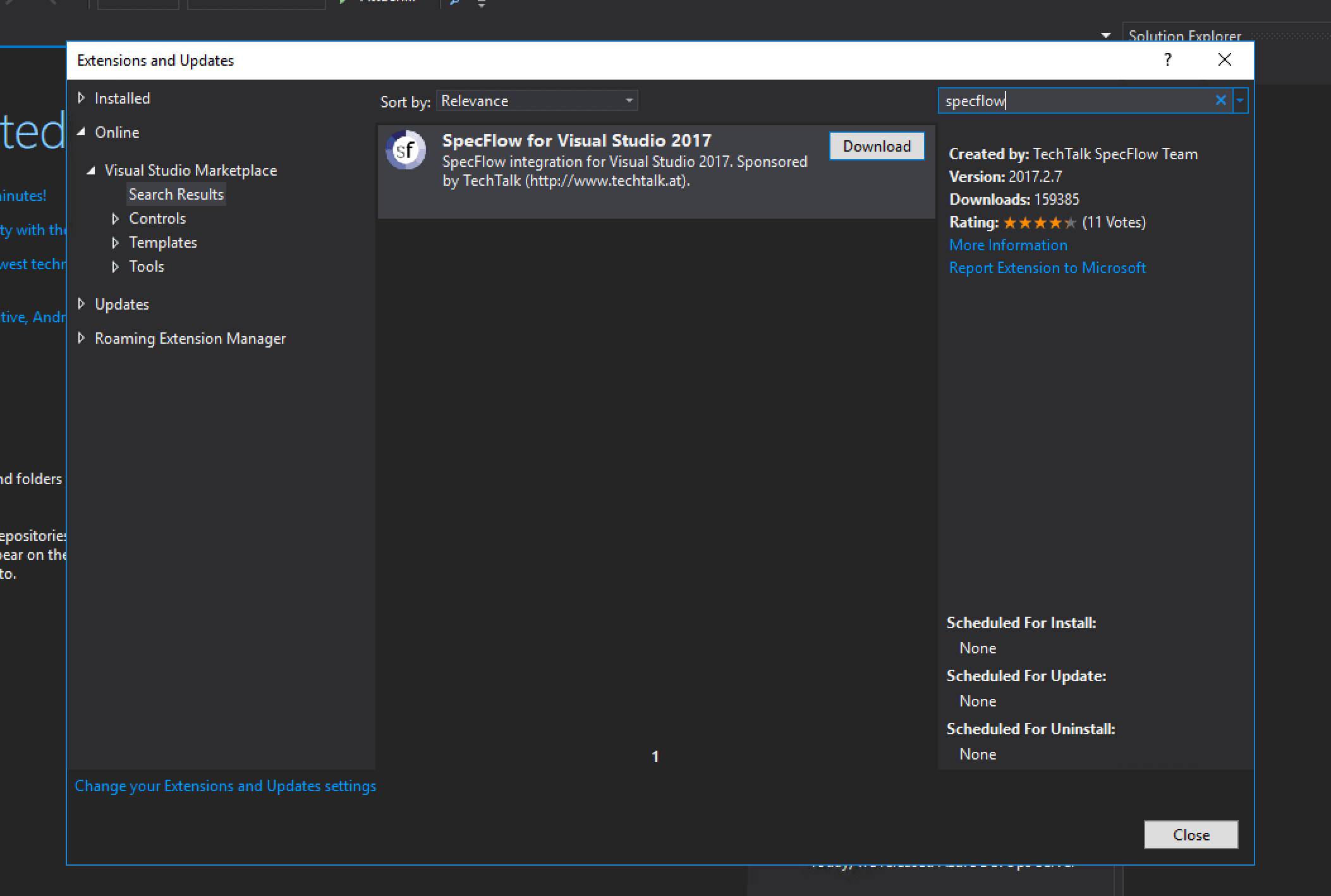
In this section, we will explore installing specflow in the Visual Studio IDE and creating feature files for a simple String Utility Application.

**Specflow Installation Guide**

**Specflow installation is a 2 step process**

**#1)** installing the required plugins in the Visual Studio IDE.

* To install the specflow plugin navigate to Tools -> Extension & Updates.
* Now click “Online” on the left panel.
* Now search for specflow in the right panel.
* From the search results select “Specflow for Visual Studio 2017”.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/Specflow-PluginInstallation.png)

**#2)** Setting up the project with feature files and step definitions.

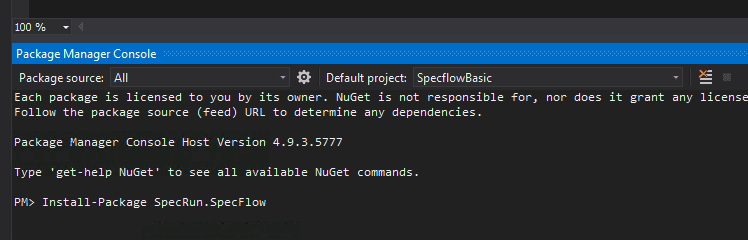
* Create a simple new project in Visual Studio. We can create any kind of project like Class Library / Console Application / Unit test project etc. For simplicity, we are taking up a Class Library project. Name the project as “SpecflowBasic”.
* In order to run the Specflow scenarios that we are going to create, we need a test runner. Specflow provides a runner out of the box called Specflow + Runner (which is a paid version and the free version introduces a delay).

(Other runners are also available for NUnit and MsTest )

To install Specflow + Runner – **Navigate to Tools -> NuGet Package Manager -> Package Manager Console.**

Once the Package Manager Console opens up – Run the command.

|  |  |
| --- | --- |
| 1 | Install-Package SpecRun.SpecFlow |

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/05/Specrun-Installation.png)

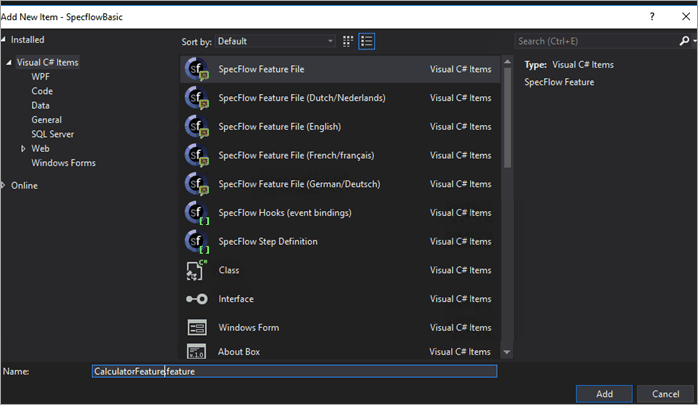
* Also, in order to Assert the values, we will need the help of a test framework. NUnit can be one of the options and the others include MsTest, etc. To install the NUnit framework to the application, open the Package Manager Console and type command.

|  |  |
| --- | --- |
| 1 | Install-Package NUnit |

**#3)** Create a new class named “CalculatorApplication” which will become our application under test. This is a simple class having functions to perform addition/multiplication/division/square root etc., for the given input. This is how the CalculatorApplication class looks like.

**#4)** Once the package gets installed, create 2 folders in the project and name them as Features and Step Definitions for storing the feature files and step bindings respectively. We will discuss in detail the reason for this folder organization for Feature & Step definitions.

**#5)** Now in the features folder, add a new Feature file and name it as CalculatorFeature.

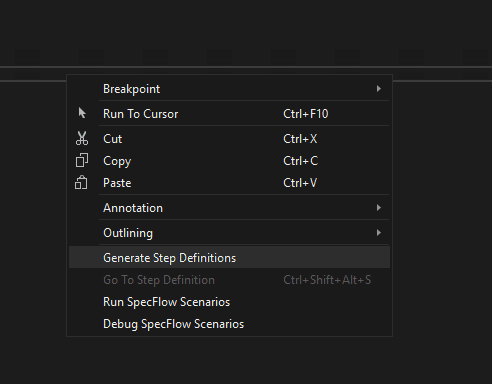
[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/Specflow-FeatureFileAdd.png)

You would see that by default the feature file has some description in Feature and Scenario.

**Replace that with what we are going to test.**

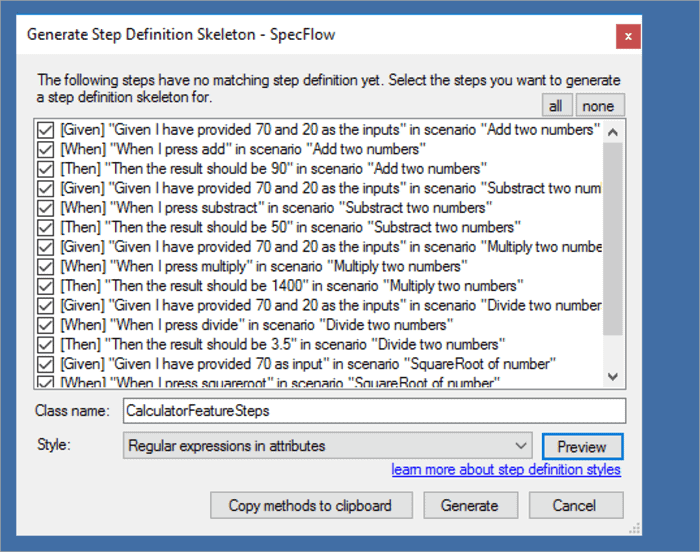
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | Feature: CalculatorFeature  In order to test my application  As a developer  I want to validate different operations of the application    Scenario: Add two numbers  Given I have provided 70 and 20 as the inputs  When I press add  Then the result should be 90    Scenario: Substract two numbers  Given I have provided 70 and 20 as the inputs  When I press substract  Then the result should be 50    Scenario: Multiply two numbers  Given I have provided 70 and 20 as the inputs  When I press multiply  Then the result should be 1400    Scenario: Divide two numbers  Given I have provided 70 and 20 as the inputs  When I press divide  Then the result should be 3.5    Scenario: SquareRoot of number  Given I have provided 70 as input  When I press squareroot  Then the result should be 8.37 |

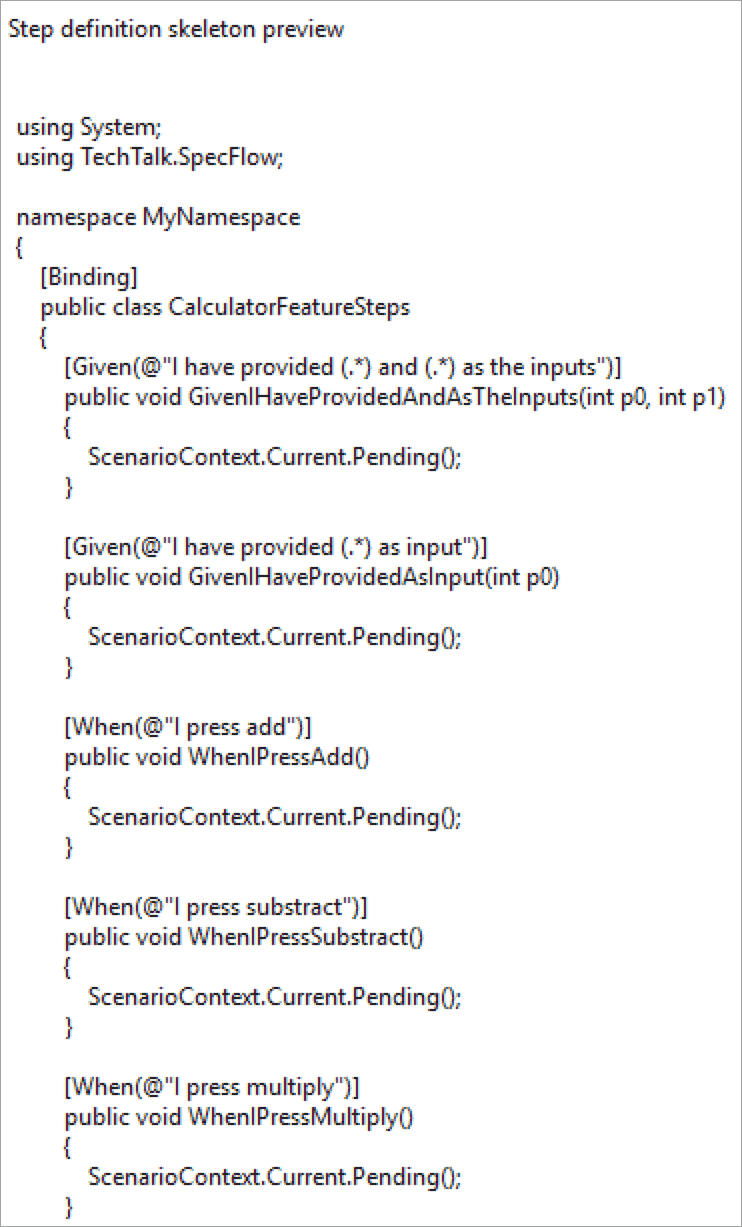
**#6)** Generating Step Definitions: Specflow provides an automated way to generate bindings/implementation for the different steps in feature file scenarios. This can be achieved by right-clicking on the feature file and clicking “Generate Step Definitions”.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/GenerateStep-Preview.png)

**This step does not guarantee an implementation**for all the steps, but it tries its best to group the common steps in scenarios and re-use as many bindings it can. However, it makes the job of avoiding boilerplate code every time when a scenario step needs to be implemented.

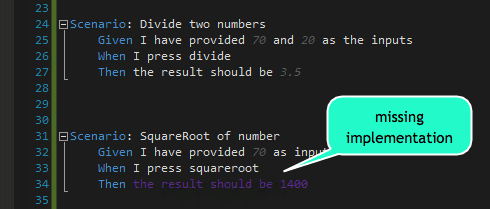
After clicking **“Generate Step Definitions”**, A window will show up listing the identified step implementations that the processor has detected. One can select or de-select as per the requirements.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/StepDetails-AutoGenerated.png)

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/StepDefinitions-Preview.png)

After creating Step definitions, still, if there are some unimplemented steps, the Feature files have a visual way of identifying the un-implemented applications. It shows those steps in a different color by making it absolutely simple to know that there are some steps which don’t have an implementation yet (or are having any ambiguous step definitions).

**A Sample Screen Depicts that Below:**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/Missing-Steps.png)

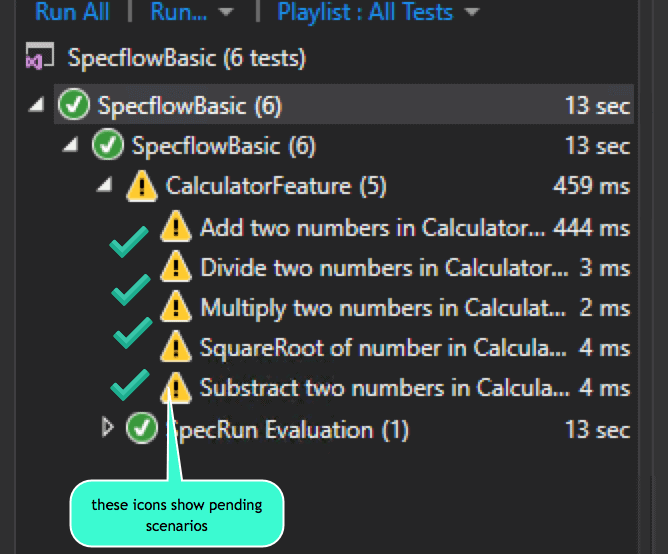
***Note:***The Step definitions can be created manually as well – Any .cs file having [Binding] Attribute is a Step implementation class and the Gherkin syntax will look for matching the implementation of the given scenario step

**Execution**

As we have already added Specflow+ Runner in the above section, executing the Scenarios is pretty straightforward (since it’s an evaluation version of Specrun, it introduces a variable delay of 10-20s before the scenarios execute. , This delay is not present for registered variants and other flavors of Specrun runner like NUnit and MsTest).

If all the steps have **not been** implemented and if there are still bindings that have a pending state. Then the output will show as pending.

Let’s try to run these tests/scenarios at this point when there is no implementation for the bindings, and the scenarios are all pending.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/PendingScenarios.png)

***Our upcoming tutorial will brief you on End to End example of using Specflow and Selenium Webdriver!***

**Specflow and Selenium Webdriver End to End Example**

we will see an end to end example of using Specflow based BDD specifications and the tests will be executed via Selenium Webdriver.

The tests could be as simple as testing the login functionality of an application. It is just that, we will describe those tests using Specflow and the Step implementations will use Selenium driver and commands and NUnit as assertion framework.

We will also use NUnit based Specflow runner (as opposed to Specrun which is not open source and introduces a random delay in the free version).

create a Unit test Project in Visual Studio and install the following pre-requisites:

**#1) Create a unit test project**

Install Nuget package for Nunit and Specflow.

Install-Package Specflow.NUnit

**#2) Install Selenium’s Webdriver for chrome browser.**

This can be installed via Nuget Package Manager console too through the below command.

Install-Package Selenium.WebDriver.ChromeDriver

**#3) Install Selenium Webdriver libraries and Selenium Wait Helpers for adding ExpectedCondition waits.**

Install-Package Selenium.WebDriver

Install-Package DotNetSeleniumExtras.WaitHelpers

**#4) Now remove the test.cs file from the project that is auto-created.**

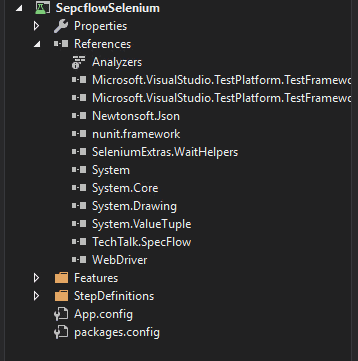
We are doing this step to avoid any confusion as we will be using feature files and step definition files for Specflow.

**#5) Create folders for Features and Step Definitions to store feature and Step definition implementation files.**

This is not a mandatory step but is useful to organize the features and step implementations in separate folders.

**#6) At the end of the above steps, the solution structure and the packages.config should look as shown below.**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <?xml version="1.0" encoding="utf-8"?>  <packages>  <package id="DotNetSeleniumExtras.WaitHelpers" version="3.11.0" targetFramework="net461" />  <package id="MSTest.TestAdapter" version="1.3.2" targetFramework="net461" />  <package id="MSTest.TestFramework" version="1.3.2" targetFramework="net461" />  <package id="Newtonsoft.Json" version="10.0.3" targetFramework="net461" />  <package id="NUnit" version="3.0.0" targetFramework="net461" />  <package id="Selenium.WebDriver" version="3.141.0" targetFramework="net461" />  <package id="Selenium.WebDriver.ChromeDriver" version="2.45.0" targetFramework="net461" /> |

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/SeleniumSpecflow-FolderStructure.png)

**Feature and Step Implementation**

Now let’s get started with the feature file and the actual step implementation.

**About the feature** – The sample feature will be of testing/validating the search functionality of Youtube Website. We will be searching for a keyword and asserting that the user got redirected to the search results page.

Add a new feature file and name it as *YoutubeSearchFeature.feature*

**Add a search functionality scenario & feature description as shown below:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | Feature: YoutubeSearchFeature In order to test search functionality on youtube  As a developer  I want to ensure functionality is working end to end  @mytag  Scenario: Youtube should search for the given keyword and should navigate to search results page  Given I have navigated to youtube website  And I have entered India as search keyword  When I press the search button  Then I should be navigate to search results page |

**The above scenario expects the test to:**

* **Navigate to Youtube Website:** This will require a Web automation framework like Selenium, which will use a Webdriver to navigate to a webpage on a browser.
* **Search for a keyword:** This step will involve looking for appropriate input elements and buttons in order to key-in the keyword and execute the search respectively.
* **Assert that the search results are displayed and the user is on the results page:** This step will involve assertions around verifying if the user landed on the correct page.

***Now let’s see the Step implementations each of the steps.***

Before that, let’s understand how we will be integrating Selenium logic/code in the existing Specflow definition.

[Selenium](https://www.softwaretestinghelp.com/selenium-tutorial-1/) or any other tool (or Unit testing stubs/mocks/drivers etc) are essentially an intermediate part of the Step Execution, but the key thing to understand is the way to integrate both these Frameworks.

Specflow enables users to write test specifications. It does not dictate the tool that should be used. Hence the test developer is free to choose as many testing tools as he wants to depend on the use case that is being solved.

**In order to use Selenium in this scenario, we need the following:**

* An instance of WebDriver (we will be using ChromeDriver for simplicity), which will enable the user to actually navigate to the webpage using a browser as per Driver implementation.
* Few WebElement declarations (and can be done as part of Step implementations only) which are required to interact with the user and pass inputs and perform actions etc.
* Few assertions on Window title, urls, etc which can be executed on driver instance.

We will be creating an instance of ChromeWebdriver in the Step Implementations file.

Hence, let’s create the Step Definition file. As we saw in the last article, Specflow does provide a mechanism to auto-generate the Step definitions (which can later be customized/modified as required).

* Navigate to the feature file, Right-click and select “Generate Step Definitions”.
* Create a new file in the StepDefinitions folder as we created earlier and name the file as YoutubeSearchFeatureSteps.cs
* Ensure that all the Steps of the scenario have been bound to the Step definitions appropriately.

**Tip – Modifying Auto-generated Step definitions:**

Now if you carefully observe, the Step definition that got generated for our search keyword step i.e. “I have entered India as search keyword” the auto-generated code, does not identify/separate the search keyword and hence it does not parameterize it.

|  |  |
| --- | --- |
| 1  2  3  4  5 | [Given(@&"I have entered India as search keyword")]  public void GivenIHaveEnteredIndiaAsSearchKeyword()  {  ScenarioContext.Current.Pending();  } |

But this is not what we want. We need the search keyword to be parameterized, otherwise, for every keyword search, we will have to create a custom Step definition.

So, let’s see, how to modify this Step definition to a more generic one, which will enable to parameterize the search Keyword. This can be done through simple regex matcher.

Refer to the below code sample. We have replaced the search keyword through a regex matcher i.e. “***(.\*)***” What this will do is that it will replace the value of any keyword that you will pass from the Scenario and will set the value of the search keyword in the input parameter named “searchString” in the below code sample.

|  |  |
| --- | --- |
| 1  2  3  4  5 | [Given(@"I have entered (.\*) as search keyword")]  public void GivenIHaveEnteredIndiaAsSearchKeyword(String searchString)  {  ScenarioContext.Current.Pending()  } |

This way, it keeps the code modular and avoids repeated boilerplate code for each Step implementation.

**Selenium Integration and Step Definition Logic**

Now let’s see the actual integration of Selenium with Specflow. Once the step definitions are generated, we will now add code to them in order to execute the actual test Scenario.

Let’s see, where we can place & initialize the Selenium Web driver instance so that it is available throughout the Scenario execution. We will be placing the Driver as a private field of the Binding Class that got generated. The driver will be initialized as a part of the class Constructor.

In this way, the driver remains initialized for the entire course of the duration of the test as we just have one Binding file for all the Steps (and it gets initialized before the test execution starts).

Also note that we will also be implementing the *IDisposable* interface, so as to Dispose the driver instance after which it is no longer required. Placing it in Dispose() Method will guarantee that once the class’s Object is getting disposed of, the driver instance can be disposed of too.

**This is how the code for declaration and initialization of WebDriver instance looks like:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | [Binding]  public class YoutubeSearchFeatureSteps : IDisposable  {      private String searchKeyword;        private ChromeDriver chromeDriver;        public YoutubeSearchFeatureSteps() => chromeDriver = new ChromeDriver();        //... other Step defintion implementations        public void Dispose()     {         if(chromeDriver != null)         {             chromeDriver.Dispose();             chromeDriver = null;         }      }  } |

With the above, the driver instance can be used as part of any Step implementation which is a part of the scenario execution.

Let’s now see the Step Implementation of each individual scenario.

**#1) Arrange Steps:**

|  |  |
| --- | --- |
| 1  2 | Given I have navigated to youtube website  And I have entered India as search keyword |

Both of these steps involve interacting with the driver instance. The first step open’s the browser window and navigates to the youtube website

The second step looks for search input button and enters “India” as the search keyword.

**Below is the implementation for both of these steps:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | [[Given(@"I have navigated to youtube website")]  public void GivenIHaveNavigatedToYoutubeWebsite()  {      chromeDriver.Navigate().GoToUrl("https://www.youtube.com");          Assert.IsTrue(chromeDriver.Title.ToLower().Contains("youtube"));  }  7.  [Given(@"I have entered (.\*) as search keyword")]  public void GivenIHaveEnteredIndiaAsSearchKeyword(String searchString)  {  this.searchKeyword = searchString.ToLower();  var searchInputBox = chromeDriver.FindElementById("search");  var wait = new WebDriverWait(chromeDriver, TimeSpan.FromSeconds(2));  wait.Until(ExpectedConditions.ElementIsVisible(By.Id("search")));  searchInputBox.SendKeys(searchKeyword);  } |

For the first Step, notice the Assertion that it ensures that the navigation to youtube was successful by checking the window title.

**Note:** There can be various ways of placing Assertions on different web elements or driver properties, but the end goal of this tutorial is just to illustrate with the most simplistic way.

In the second step, we have added a Dynamic wait using **ExpectedConditions** which will ensure that the search box is visible before the code tries to key-in the search keyword.

Also, we are storing the searchString in a **private field searchKeyword.**This is done so that the searchKeyword can be used in other Step implementations too.

**Tip – Passing data across the Steps**

Passing/Storing data by this approach (i.e. through class variables) is one of the means through which data can be shared across Step bindings.

There are other ways to do this as well like Specflow itself provides a Dynamic Dictionary Object called ScenarioContext. We will see more details about this in the upcoming articles.

**#2) Act Step**

|  |  |
| --- | --- |
| 1 | When I press the search button |

Now let’s look at the actual action, which is clicking on the Search button. The step implementation file will search for the search button and click it in order to execute the scenario step.

The code for this step looks as shown below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | [When(@"I press the search button")]  public void WhenIPressTheSearchButton()  {  var searchButton = chromeDriver.FindElementByCssSelector("button#search-icon-legacy");      searchButton.Click();  } |

**#3) Finally the Assert Step:**

|  |  |
| --- | --- |
| 1 | Then I should navigate to search results page |

In this step, we are just verifying from the Driver properties as to whether the URL and the page title contains the search keyword or not.

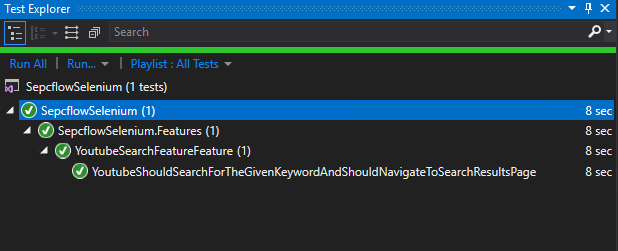
**The code for this step is shown below:**

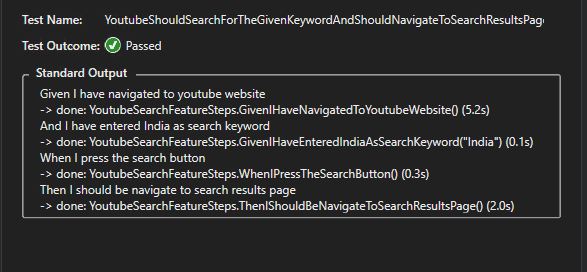
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | [Then(@&quot;I should be navigate to search results page&quot;)]  public void ThenIShouldBeNavigateToSearchResultsPage()  {      // After search is complete the keyword should be present in url as well as page title`      Assert.IsTrue(chromeDriver.Url.ToLower().Contains(searchKeyword));      Assert.IsTrue(chromeDriver.Title.ToLower().Contains(searchKeyword));  } |

**Execution**

Now, let’s try to execute the Scenario and see the results. Once the Scenario is executed, all the Scenario Steps will be executed Sequentially. The test will be opening a browser, navigating to a website and then performing some action.

The output of the test can be seen by clicking the “Output” button from the test summary which shows the success/failure of each individual step.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/SpecflowSelenium_TestOutput.png)

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/SpecflowSelenium_DetailedOutput.png)

**Tips**

**Intermediate failing Steps**

In case of a Scenario having intermediate steps that get failed, please note that in those scenarios Specflow will simply not execute any remaining Steps of that Scenario and will mark the result of the test as failed.

**Running tests with NUnit Adapter**

For this Example, we have executed our tests using the Specflow.NUnit test runner (that we had installed via Nuget Package Manager).

**This is different in a few ways as shown below from the Specrun runner that we had used in the earlier article.**

1. Specflow.NUnit runner is open source.
2. It does not introduce any delay while executing the tests.

**Conclusion**

In this article, we saw an end to end example of Integrating Selenium with Specflow framework through a simple test scenario of a video search on the Youtube application.

While integrating Selenium, we also went through, how to share data across different bindings through private class fields. We also covered running the test on NUnit runner Vs the Specrun runner and compared both in detail.

**Code Files**

**YoutubeSearchFeature.feature**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Feature: YoutubeSearchFeature  In order to test search functionality on youtube  As a developer  I want to ensure functionality is working end to end    @mytag  Scenario: Youtube should search for the given keyword and should navigate to search results page  Given I have navigated to youtube website  And I have entered India as search keyword  When I press the search button  Then I should be navigate to search results page |

**YoutubeSearchFeatureSteps.cs**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66 | using NUnit.Framework;  using OpenQA.Selenium;  using OpenQA.Selenium.Chrome;  using OpenQA.Selenium.Support.UI;  using SeleniumExtras.WaitHelpers;  using System;  using System.Collections.Generic;  using System.Linq;  using TechTalk.SpecFlow;    namespace SepcflowSelenium.StepDefinitions  {  [Binding]  public class YoutubeSearchFeatureSteps : IDisposable  {  private String searchKeyword;    private ChromeDriver chromeDriver;    public YoutubeSearchFeatureSteps() => chromeDriver = new ChromeDriver();    [Given(@"I have navigated to youtube website")]  public void GivenIHaveNavigatedToYoutubeWebsite()  {  chromeDriver.Navigate().GoToUrl("https://www.youtube.com");  Assert.IsTrue(chromeDriver.Title.ToLower().Contains("youtube"));  }    [Given(@"I have entered (.\*) as search keyword")]  public void GivenIHaveEnteredIndiaAsSearchKeyword(String searchString)  {  this.searchKeyword = searchString.ToLower();  var searchInputBox = chromeDriver.FindElementById("search");  var wait = new WebDriverWait(chromeDriver, TimeSpan.FromSeconds(2));  wait.Until(ExpectedConditions.ElementIsVisible(By.Id("search")));  searchInputBox.SendKeys(searchKeyword);  }    [When(@"I press the search button&quot;)]  public void WhenIPressTheSearchButton()  {  var searchButton = chromeDriver.FindElementByCssSelector("button#search-icon-legacy");  searchButton.Click();  }    [Then(@"I should be navigate to search results page")]  public void ThenIShouldBeNavigateToSearchResultsPage()  {    System.Threading.Thread.Sleep(2000);  // After search is complete the keyword should be present in url as well as page title`  Assert.IsTrue(chromeDriver.Url.ToLower().Contains(searchKeyword));  Assert.IsTrue(chromeDriver.Title.ToLower().Contains(searchKeyword));  }      public void Dispose()  {  if(chromeDriver != null)  {  chromeDriver.Dispose();  chromeDriver = null;  }  }  }  } |

***Watch out our upcoming tutorial to know more about Shared & Scoped Bindings, Hooks and Step Reuse in Specflow!***

**Advanced Specflow Shared & Scoped Bindings, Hooks and Step Reuse**

By **[Sruthy](https://www.softwaretestinghelp.com/author/remyasthva/)**  Updated March 7, 2024

**Advanced Specflow Tutorial on Shared & Scoped Bindings, Hooks and Step Reuse:**

In this [**Complete Guide on Specflow Training**](https://www.softwaretestinghelp.com/specflow-tutorial/)**,**we had a look at [**End to End Example of Using Specflow**](https://www.softwaretestinghelp.com/specflow-and-selenium/) in detail in our previous tutorial.

In our previous article, we saw, an example to share data using private instance variables of the binding class and then referred to the same instance across different steps.

Often at times, it is required that the code for a lot of step implementation (especially setup) can be shared across different scenarios, as well as the actual data that’s being used in different steps for performing different actions.

**Table of Contents:**[[Show](https://www.softwaretestinghelp.com/specflow-bindings-hooks-step-reuse/)]

**Shared & Scoped Bindings, Hooks and Step Reuse**

**Here are the video tutorials on Shared & Scoped Bindings, Hooks and Step Reuse**

**VIDEO #1: Specflow Context Sharing**



**VIDEO #2: Specflow Hooks and Scoped Bindings**



**This approach has a lot of shortcomings like:**

* How would you share the data across different binding files?
* It can be done by creating copies of instances/variables but it will end up muddying the code everywhere and will cause inconsistencies.

**Specflow provides 2 alternative approaches to handle this kind of scenarios:**

* Keeping context data in ScenarioContext object.
* Sharing data with the lightweight dependency/context injection framework that specflow provides.

**Multiple Binding files**

Let us first understand the use case of multiple binding/step implementation files. Generally in all projects, when you decide to use Specflow, you will be defining multiple features and grouping the binding implementation as per your needs.

**For Example** You may decide to keep all the bindings related to setup. Example, opening web page, navigating to menus etc. in one binding file, all the search input related definitions in another binding file and so on.

Hence, multiple binding files will essentially be a part of any project where Specflow is being used as the testing tool/framework. It also adds to a good design practice of **Separation of Concerns** which is a key pillar of the SOLID design framework.

Now in order to share the context across these binding files, we will look at the possible approaches in the upcoming sections of this tutorial and refer the same youtube search example (that was discussed/implemented in the last article) for all the approaches that we will discuss.

**Dependency Injection**

Let us first try to understand what Dependency Injection is?

Dependency Injection is a common term and practice with a lot of projects and in simple terms, it means to introduce/inject all the dependencies required by the code at run time. This can be done using Constructor Parameters when the Object of the class is getting initialized.

There are a lot of open source frameworks that handle dependency injections like AutoFac for .NET, Google Guice for [JAVA](https://www.softwaretestinghelp.com/java/) etc.

Since, the intention here is not to go deep into what’s and how’s of dependency injection, let’s aim to understand from Specflow’s point of view which already provides a lightweight context injection framework.

**Steps**

**Given below are the sequence of steps that need to be followed, in order to inject dependencies/context in binding files:**

1. Create Classes (POCOs) for the context object/data that needs to be shared.
2. For each of the binding classes, create a local instance of Context class type and instantiate it using the Constructor Parameter.
3. Now whatever changes are done to the context parameter will get passed to the next class where the object will be initialized using the Constructor Parameter.

**Example – Context Injection**

Let’s look at this Context Injection in action for our Youtube Search Example.

**We had instance variables for 2 things/data items:**

1. For storing the search term as it was used across multiple steps.
2. For storing the actual webdriver instance so that same instance can be reused/referred to across multiple steps.

**Pro Tip:** An important point to note here is that the lifetime of the context object instance is for Scenario Execution.  This means that each Scenario Execution will have a new copy of the context objects that will be initialized in the Binding classes where they are injected via a constructor.

Now, using Context Injection, we will be creating 2 POCOs i.e. 1 each for storing the search- related data and we will call it as **SearchContext.cs** and the other one for storing all the info related to the actual webdriver instance and we call it as – **WebDriverContext.cs**

SearchContext.cs file will have a String variable to store the search term.  
WebDriverContext.cs will contain an instance of ChromeDriver which will drive the entire test.

Let’s see below how both of these files look. Both of these are simple Plain Old C# files, which have instance fields to store the actual context information.

Both these files are added in a new namespace called Context to have a logical separation from Features and Bindings.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public class SearchContext  {  public SearchContext()  {  this.searchTerm = String.Empty;  }  public String searchTerm;  }  public class WebDriverContext  {  public WebDriverContext()  {  this.chromeDriver = new ChromeDriver();  }  public ChromeDriver chromeDriver;  } |

Now for **illustration purposes**, let’s split our Binding file into 2 files i.e. Just for illustration/explanation purpose, We will be moving the Validation (or all scenario steps for **Then** group i.e. Validation) and we will be calling it as “**YoutubeSearchVerificationSteps.cs**”

**So, we now have 2 binding files:**

1. YoutubeSearchFeatureSteps.cs
2. YoutubeSearchVerificationSteps.cs

**The breakup of steps is shown below:**

|  |  |
| --- | --- |
| 1  2  3  4  5 | Scenario: Youtube should search for the given keyword and should navigate to search results page      Given I have navigated to youtube website    //YoutubeSearchFeatureSteps.cs      And I have entered India as search keyword  //YoutubeSearchFeatureSteps.cs      When I press the search button  //YoutubeSearchFeatureSteps.cs      Then I should be navigate to search results page  //YoutubeSearchFeatureVerificationSteps.cs |

Once we have the context file/objects ready, then we are now in a position to utilize and initialize the context values as the scenario execution progresses.

Let’s first see, how and where the actual injection is taking place. For the individual binding files, we will be injecting the Context instances via Constructor as shown below.

The changes/updates to the instances done in one binding file will be injected in the other binding files as and when they get initialized.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | private readonly SearchContext \_searchContext;  private readonly WebDriverContext \_webDriverContext;    public YoutubeSearchFeatureSteps(SearchContext searchContext, WebDriverContext webDriverContext)  {      \_searchContext = searchContext;      \_webDriverContext = webDriverContext;  } |

The above code injects the objects of SearchContext and WebDriverContext through the constructor. The code remains the same in all binding files wherever the context needs to be initialized, just the name of the Constructor changes as per the Binding File Class Name.

Let’s try to go over each step of the Scenario and understand where we are updating our context variables.

**Step 1**

|  |  |
| --- | --- |
| 1 | Given I have navigated to youtube website //YoutubeSearchFeatureSteps.cs |

This step will update the properties of webDriver like URL, pageTitle etc.

**Step 2**

|  |  |
| --- | --- |
| 1 | And I have entered India as search keyword  //YoutubeSearchFeatureSteps.cs |

Here, in addition to updation of the state of webDriver, we will also update SearchContext’s -> searchTerm field and set it to the actual search keyword i.e. “India” from the Scenario Step.

**Step 3**

|  |  |
| --- | --- |
| 1 | When I press the search button //YoutubeSearchFeatureSteps.cs |

This will just result in further updation of the state of WebDriver instance.

Till this point, all these 3 steps belong to the same binding file and we have updated the stats of the Context objects that were injected.

**Step 4**

|  |  |
| --- | --- |
| 1 | Then I should be navigate to search results page //YoutubeSearchFeatureVerificationSteps.cs |

This step is a part of another Binding file, where also we have injected the context in the same way through Constructor as we did for the first binding file.

In the actual Step definition code, when you try to access the state of *WebDriverContext’s* *chromeDriver* or the values of instance members of *SearchContext*, you will see that the values that were set in the previous binding file have been propagated to the injected instance and can be accessed in the step implementation code anywhere in the binding file.

Now, let’s try executing this Scenario end to end by clicking the Run Scenario option from the Test Explorer. Now you will notice that the test passes and it can access all values of injected context classes across the binding files wherever it is injected through the respective *Constructors*.

**There are a lots of advantages of having context injection in place and some of them are listed below:**

1. Separation of concerns – test context is different from the actual test code.
2. Reuse of context across bindings.
3. Leads to better modular design.

**Using Scenario Context & Feature Context**

ScenarioContext and FeatureContext are static classes that are capable of holding the shared state/context during the execution of a scenario and feature respectively.

Think of these 2 classes as a key-value dictionary where the key being name of the state variable and value is the actual object which might be as simple as a primitive **int** or **String** and as complex as a user-defined custom Class.

An important point to note here is when the value of a key is retrieved, it always returns an Object that is required to be casted into a specific type as desired.

**Syntax:**

**Setting a value in ScenarioContext:**

|  |  |
| --- | --- |
| 1 | ScenarioContext.Current[“key“] = value; |

**Getting a value from ScenarioContext:** Depending upon the type of the value (the type of object essentially) that was set in ScenarioContext, while fetching the value, it needs to be cast to the respective type. Suppose the type of Object that was set was of type int then in order to get the value, we will need to do an explicit cast to int.

|  |  |
| --- | --- |
| 1 | var value = (int)ScenarioContext.Current[“key“] |

Please note, that the scope of ScenarioContext variables is for scenario execution.

Similar to ScenarioContext, there exists another static class called FeatureContext. The only difference here is that the scope of data for FeatureContext is until the entire Feature execution is complete.This is generally less rarely used as usually there is a lesser need to keep context/data alive for the entire feature as opposed to per scenario.

So, now let us write/implement the same piece of code using ScenarioContext instead of the Specflow’s context injection framework.

To illustrate the usage of *ScenarioContext*, we will be using it to hold states of the search term in our example i.e. wherever the search term is first used, at that time, we will set the value in *ScenarioContext* and for all future usages, we will refer to the value fetched from the *ScenarioContext*.

So, essentially we will be making changes in our Act (When) step where we perform a search for the given search term and then we will fetch the value from ScenarioContext in our Assertion Step (Then) where we validate that the results are for a specified search term.

**Look at the implementation of Step definitions below with ScenarioContext:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | [Given(@"I have entered (.\*) as search keyword")]  public void GivenIHaveEnteredIndiaAsSearchKeyword(String searchString)  {  ScenarioContext.Current["searchTerm"] = searchString.ToLower();  var searchInputBox = \_webDriverContext.chromeDriver.FindElementById("search");  var wait = new WebDriverWait(\_webDriverContext.chromeDriver, TimeSpan.FromSeconds(2));  wait.Until(ExpectedConditions.ElementIsVisible(By.Id("search")));  searchInputBox.SendKeys(searchString);  } |
| 1  2  3  4  5  6  7  8  9 | [Then(@"I should be navigate to search results page")]  public void ThenIShouldBeNavigateToSearchResultsPage()  {  var expectedSearchTerm = (String)ScenarioContext.Current["searchTerm"];  System.Threading.Thread.Sleep(2000);  // After search is complete the keyword should be present in url as well as page title`  Assert.IsTrue(\_webDriverContext.chromeDriver.Url.ToLower().Contains(expectedSearchTerm));  Assert.IsTrue(\_webDriverContext.chromeDriver.Title.ToLower().Contains(expectedSearchTerm));  } | |

As you can see above, in the Assertion step, we have fetched the value of context using the stored key and used it for validating the output with the supplied input.

In the next section, we will discuss the special events provided by Specflow called Hooks which can be utilized to execute setup and cleanup code at various stages of test execution.

**Hooks in Specflow**

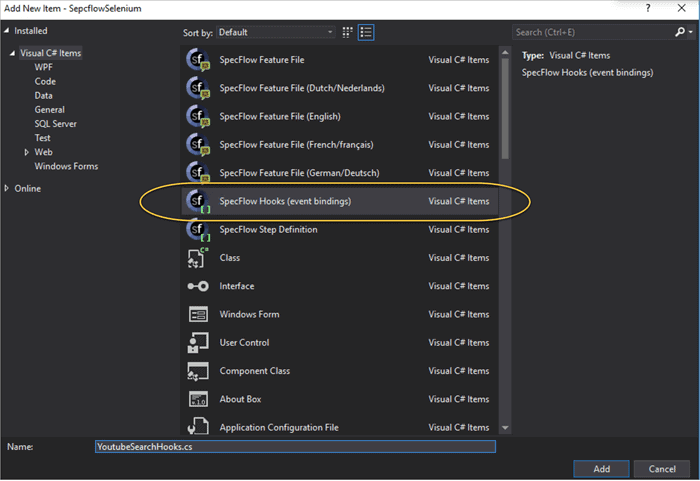
Hooks are special events that are raised by the Specflow framework while it is executing a feature and a scenario. These events when generated, provide an opportunity to write an event handler and any code that you want to associate with the specific event.

**For Example,** let’s look at “BeforeScenario” hook and from the name itself it is evident, that this event will be raised before running any scenario from the feature. Think of it as test initialize setup in other [unit testing](https://www.softwaretestinghelp.com/unit-testing/) frameworks like MSUnit (for C#) and [Junit](https://www.softwaretestinghelp.com/selenium-junit-framework-selenium-tutorial-11/) (for Java)

First, let’s see, how the hooks are added as part of the tests. Though hooks can be added to any file which has Binding attribute, Specflow also provides a separate type of class file of type hooks (which also has a Binding attribute attached).

It’s generally a good idea to keep the hooks in a separate file in order to have more intuitive logical grouping.

As shown in the below figure, add a specflow Hooks file to the Step Definitions folder (and name it as YoutubeSearchHooks.cs)

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/AddingHooksFile.png)

Let’s see what all hooks are provided by Specflow along with examples (for our Youtube Search example) in the below section (To keep it simple, We will be looking at them in the form of pairs):

**#1) BeforeTestRun, AfterTestRun:** This is the top level hook and it allows the execution of code before a test run starts and after the test run completes. Please note that this is above the Feature level as well.

Hence, if your code has multiple feature files – then the code mentioned in these hooks will run once before the test run starts and once after the code execution completes.

**There are a few important points that should be noted here:**

* Please note that when tests are run with multiple threads then both of these hooks are executed once for each of the threads.
* The bindings for these hooks should be static as the test context has still not been set before the test (and is not available after the test).

Simply add 2 static methods in the binding file that we created and add attributes [BeforeTestRun] and [AfterTestRun] respectively.

For simplicity, we have just written console statements (but these hooks can contain any initialization and cleanup logic as required)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | [BeforeTestRun]  public static void BeforeTestRun()  {  Console.WriteLine("In Method Before test run!!");  }    [AfterTestRun]  public static void AfterTestRun()  {  Console.WriteLine("In Method After test run!!");  } |

**#2) BeforeFeature, AfterFeature:** As the name implies, these hooks will execute once for each feature files before and after completion respectively. (If there is just one feature file then the hooks will execute just once). Like Before and After the test run, the methods that contain the code for these hooks should be static.

Add these as 2 static methods in the binding file that was created for hooks.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | [BeforeFeature]  public static void BeforeFeature()  {      Console.WriteLine("In Method Before Feature!");  }    [AfterFeature]  public static void AfterFeature()  {      Console.WriteLine("In Method After Feature!");  } |

**Pro Tip:**Please note that it’s not essential for any of these hooks to appear in pairs i.e. if you are writing a hook for *BeforeFeature,*then you are not obliged to write another hook for AfterFeature as well. Any of these hooks can stay independent of each other.

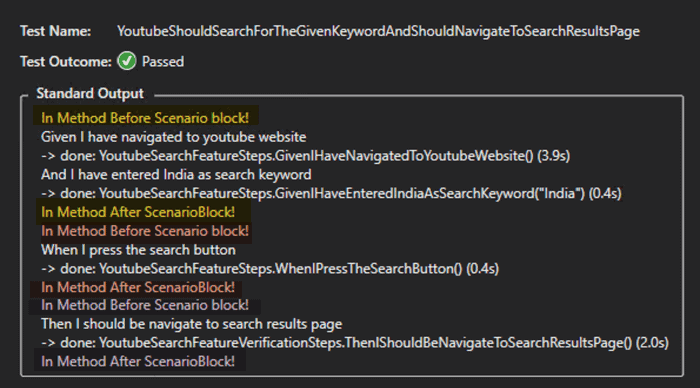
**#3) BeforeScenarioBlock, AfterScenarioBlock:** These hooks run before every type of scenario block i.e. before every group of “Given”, “When” & “Then”

Unlike hooks for test and feature level, the binding methods for these hooks need not be static.

**Let’s look at the code Example below:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | [BeforeScenarioBlock]  public void BeforeScenarioBLock()  {  Console.WriteLine("In Method Before Scenario block!");  }    [AfterScenarioBlock]  public void AfterScenarioBlock()  {  Console.WriteLine("In Method After ScenarioBlock!");  } |

If you execute the same scenario, with the above hooks, you can see the console statements getting printed in the test output before and after each scenario block. Refer to the below screenshot, with highlighted Scenario blocks and console output.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/ScenarioBlockHooks.png)

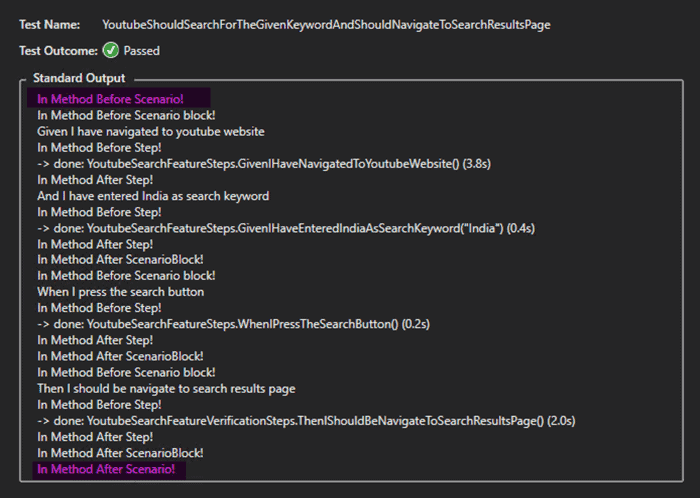
**Pro Tip:**The organization of “Given”, “When” and “Then” does not impose any restrictions on the programmer. You can specify these any number of times and in any order for a given scenario.

**#4) BeforeScenario, AfterScenario:**Again as the name implies, these hooks will be executed once per each scenario, before it starts and after it completes. These are the most commonly used hooks as they are the perfect place to have setup and cleanup logic respectively.

Simply add method bindings with attributes BeforeScenario and AfterScenario for implementing these hooks in the code.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | [BeforeScenario]  public void BeforeScenario()  {  Console.WriteLine("In Method Before Scenario!");  }    [AfterScenario]  public void AfterScenario()  {  Console.WriteLine("In Method After Scenario!");  } |

**Let’s look at the output for scenario execution (hook output shown as highlighted below)**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/BeforeScenarioHook.png)

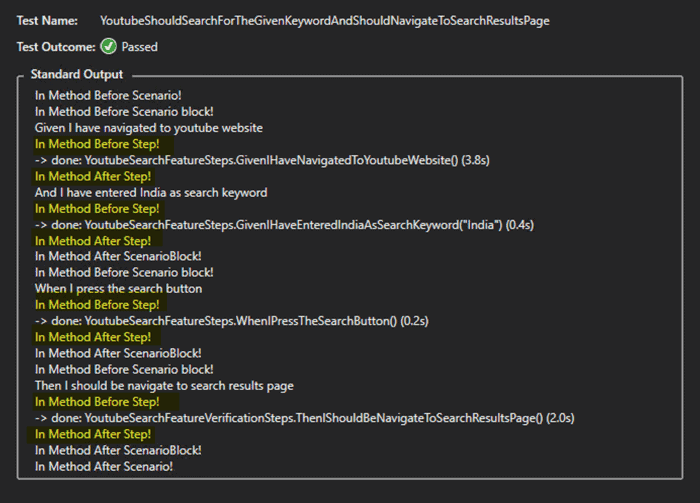
**#5) BeforeStep, AfterStep:** Both these hooks are per step level i.e. these execute for each individual step of the scenario.

To implement, add method bindings in the hooks file with attributes BeforeStep and AfterStep respectively.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | [BeforeStep]  public void BeforeStep()  {  Console.WriteLine("In Method Before Step!");  }    [AfterStep]  public void AfterStep()  {  Console.WriteLine("In Method After Step!");  } |

Hence, if we have 4 steps in a scenario, these hooks will execute once for each step.

**Let’s see, how the output looks like for these hooks (highlighted below):**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/BeforeStepHook.png)

**Scoped Bindings**

Scoping is one of the most important feature of specflow and it allows you to add a lot of flexibility to your test organization and execution by using tags for the tests, and restrict bindings to be applied to a Feature or Scenario or Feature/Scenario having some tags.

Think of it like executing/defining bindings depending on the scope defined, like a feature/scenario title or a tag (this overrides the global nature of binding and hides some bindings by restricting to a more confined and limited scope)

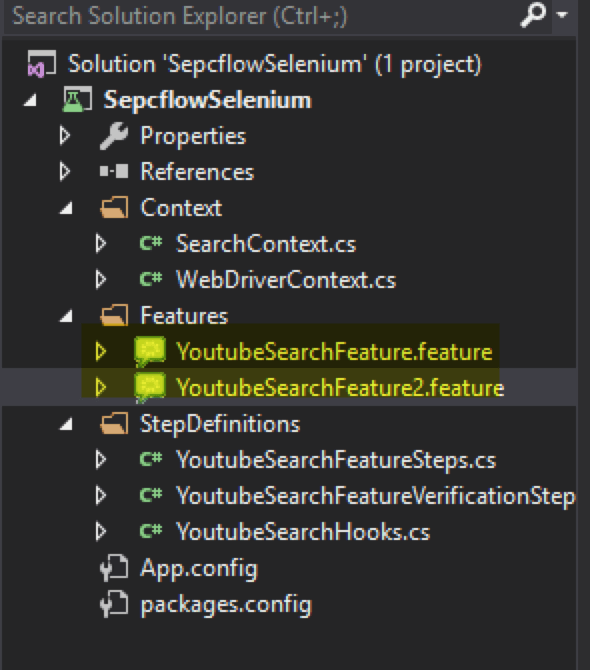
**This comes in handy in a lot of situations like:**

* The text of the scenario step is the same, but you may want different treatment for it depending on the scenario that is getting executing.
* You don’t want to reuse/share a binding implementation across features/scenarios

Let’s understand the different types of scoping techniques and apply them on the same youtube search feature example.

We will be copying the same feature file and will be just changing the name of the feature to illustrate the Scoping rules in a better way.

**Hence after creating a copy of the feature, our code structure will look as shown below:**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/ProjectStructure-AfterFeatureCopy.png)

**Scoping by Tag**

Scoping by tag, restricts the binding files by tag values on feature or scenario.

Let’s first try to understand what is a tag in Specflow and where all it can be applied?

A tag in Specflow is nothing but a way to categorize features and scenarios in a group (essentially logical grouping) and can help in using features like binding scopes, running scenarios/features with a particular tag etc.

**Let’s apply a tag “test” on Scenario in any one of the feature files.**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @test  Scenario: Youtube should search for the given keyword and should navigate to search results page      Given I have navigated to youtube website      And I have entered India as search keyword      When I press the search button      Then I should be navigate to search results page |

**Now, let’s add tag scope to any one of the step definitions i.e. let’s say for the first step like this:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | [Scope(Tag ="test")]  [Given(@"I have navigated to youtube website")]  public void GivenIHaveNavigatedToYoutubeWebsite()  {  \_webDriverContext.chromeDriver.Navigate().GoToUrl("https://www.youtube.com");  Assert.IsTrue(\_webDriverContext.chromeDriver.Title.ToLower().Contains("youtube"));  } |

Once the above binding is scoped, it will be executed only for those scenario/features, that will have this tag available.

Also, please note that this tag can not only be placed around a Scenario but, also at Feature level. Let’s apply this tag to the second feature file (which is actually a copy of the first one).

|  |  |
| --- | --- |
| 1  2  3  4  5 | @test  Feature: YoutubeSearchFeature2  In order to test search functionality on youtube  As a developer  I want to ensure functionality is working end to end |

You’ll see that as soon as this tag is added to feature, the bindings which became hidden for the scenario in this feature file will again start getting executed.

**Scoping by Scenario Title**

Similar to the approach for Scoped bindings with tag, Scenario title can also be used for defining the Scope of a binding method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | [[Scope(Scenario = "Youtube should search for the given keyword and should navigate to search results page")]  [Given(@"I have navigated to youtube website")]  public void GivenIHaveNavigatedToYoutubeWebsite()  {      \_webDriverContext.chromeDriver.Navigate().GoToUrl("https://www.youtube.com");      Assert.IsTrue(\_webDriverContext.chromeDriver.Title.ToLower().Contains("youtube"));  } |

**Scoping by Feature Title**

Similar to Scenario-based scoping, you can apply Feature-based scoping by just mentioning the Feature title in Scope.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | [Scope(Feature= "YoutubeSearchFeature")]  [Given(@"I have navigated to youtube website")]  public void GivenIHaveNavigatedToYoutubeWebsite()  {  \_webDriverContext.chromeDriver.Navigate().GoToUrl("https://www.youtube.com");  Assert.IsTrue(\_webDriverContext.chromeDriver.Title.ToLower().Contains("youtube"));  } |

**Scoping – Tips and Tricks**

**#1)** It is possible to use a combination of scoping rules as well. Like a combination of tag and ScenarioTitle, so suppose there is a tag at feature level and you want to restrict a binding scope to a specific scenario in that Feature file then you could use a combination of Tag and ScenarioTitle.

|  |  |
| --- | --- |
| 1  2 | [Scope(Tag=”test” Feature= "YoutubeSearchFeature")]  [Given(@"I have navigated to youtube website")] |

**#2)** Scoping rules can also be defined at the class level itself along with the Binding attribute. This will result in all binding implementations specified in that class to follow the Scoped restrictions.

|  |  |
| --- | --- |
| 1  2 | [Binding, Scope(Tag ="test")]  public class YoutubeSearchFeatureSteps : IDisposable |

**#3)** Scoping can be applied on hooks as well. This comes in handy when you want to have for instance different initialization logic based on tags available on scenarios. Had this been not the case then you would have to write all initialization logic as a part of the individual steps as well.

**Let’s see an Example below:**

|  |  |
| --- | --- |
| 1  2  3  4  5 | [BeforeStep("test")]  public void BeforeStep()  {  Console.WriteLine("In Method Before Step!");  } |

**Other Specflow Keywords**

**Background**

Specflow provides a special keyword termed as “Background” which is kind of Scenario setup in a feature file, in the way we have hooks for test setup, and its similar to Scenario setup. With “Background” you can define initial data/steps that are required by all the scenarios of the file and it essentially helps to avoid duplicating the steps in each and every scenario in the feature file.

In order to illustrate this, let’s add a new scenario in our file which searches for another keyword say “America”. Now both the scenarios have something in common. Like both the scenarios have a pre-requisite to be on the Youtube site. We can add this pre-requisite step in the “Background” section too.

**Look at the code Example below:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | Background:      Given I have navigated to youtube website    Scenario: Youtube keyword search 1      And I have entered India as search keyword      When I press the search button      Then I should be navigate to search results page      Scenario: Youtube keyword search 2      And I have entered America as search keyword      When I press the search button      Then I should be navigate to search results page |

So, with the above example, you can see that we have added one step in the Scenario Background. Now whenever any scenario from this feature file is run, the background step will execute before each scenario.

**Few points to note:**

1. A feature file can have only one Background section.
2. The Background section should be defined before the first scenario of the feature file.

**ScenarioOutline & Examples**

ScenarioOutline is another useful keyword from Specflow which is used for supporting data-driven tests and is a quite common practice that is especially for unit tests, where we try to test the same function with different inputs.

It also helps to avoid duplication of scenario steps for each different data sample that needs to be verified.

Let’s use the same example as above for youtube and create a scenario outline for searching 2 different input search terms – India and America. This is a perfect example of data driven test where just the search parameter changes and the rest all remains the same.

Also, ScenarioOutline can be combined with other keywords like Background to make it even more powerful.

**Few important points to note:**

1. The data values for which scenario needs to be run are placed in the “Examples” section at the end of the scenario.
2. The first row of the Examples section acts as column/variable names that are referred to in the actual step and replaced with the actual input value when the scenario gets executed.

**Let’s look at the Example below:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Background:  Given I have navigated to youtube website    Scenario Outline: Youtube keyword search  And I have entered &lt;searchTerm&gt; as search keyword  When I press the search button  Then I should be navigate to search results page  Examples:  | searchTerm |  | India |  |America | |

**Conclusion**

With the out of the box context injection framework of Specflow, it makes it really simple to inject context and reuse across different binding files. The DI framework that Specflow internally uses is [BoDI](https://github.com/gasparnagy/BoDi" \t "_blank)

The other approaches to reuse object/data across Scenario is Specflow hooks for ScenarioContext (and FeatureContext if you want some data to be retained across the entire feature execution).

Both of these approaches provide separation of concern and allow the context to be reused across bindings in contrast to keeping local instance variables in each and every Binding file.

**We also looked at other Specflow features like**

* Hooks – which allow special logic to be placed during different events that take place during test execution like BeforeScenario/AfterScenario, BeforeFeature/AfterFeature etc.
* Specflow keywords like Background and ScenarioOutline help to avoid duplicating the scenarios and keep the feature files crisp and simple.

**Code Files**

**Feature Files**

**#1) YoutubeSearchFeature.feature**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Feature: YoutubeSearchFeature  In order to test search functionality on youtube  As a developer  I want to ensure functionality is working end to end    @test  Scenario: Youtube should search for the given keyword and should navigate to search results page  Given I have navigated to youtube website  And I have entered India as search keyword  When I press the search button  Then I should be navigate to search results page |

**Step Definition files**

**#1) YoutubeSearchFeatureSteps.cs**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | [Binding]  public class YoutubeSearchFeatureSteps : IDisposable  {  private readonly SearchContext \_searchContext;  private readonly WebDriverContext \_webDriverContext;    public YoutubeSearchFeatureSteps(SearchContext searchContext, WebDriverContext webDriverContext)  {  \_searchContext = searchContext;  \_webDriverContext = webDriverContext;  }    [Given(@"I have navigated to youtube website")]  public void GivenIHaveNavigatedToYoutubeWebsite()  {  \_webDriverContext.chromeDriver.Navigate().GoToUrl("https://www.youtube.com");  Assert.IsTrue(\_webDriverContext.chromeDriver.Title.ToLower().Contains("youtube"));  }    [Given(@"I have entered (.\*) as search keyword")]  public void GivenIHaveEnteredIndiaAsSearchKeyword(String searchString)  {  \_searchContext.searchTerm = searchString.ToLower();  var searchInputBox = \_webDriverContext.chromeDriver.FindElementById("search");  var wait = new WebDriverWait(\_webDriverContext.chromeDriver, TimeSpan.FromSeconds(2));  wait.Until(ExpectedConditions.ElementIsVisible(By.Id("search")));  searchInputBox.SendKeys(\_searchContext.searchTerm);  }    [When(@"I press the search button")]  public void WhenIPressTheSearchButton()  {  var searchButton = \_webDriverContext.chromeDriver.FindElementByCssSelector("button#search-icon-legacy");  searchButton.Click();  }    public void Dispose()  {  if (\_webDriverContext.chromeDriver != null)  {  \_webDriverContext.chromeDriver.Dispose();  \_webDriverContext.chromeDriver = null;  }  }  } |

**#2) YoutubeSearchFeatureVerificationSteps.cs**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | [Binding]  public sealed class YoutubeSearchFeatureVerificationSteps  {  private readonly SearchContext \_searchContext;  private readonly WebDriverContext \_webDriverContext;    public YoutubeSearchFeatureVerificationSteps(SearchContext searchContext, WebDriverContext webDriverContext)  {  \_searchContext = searchContext;  \_webDriverContext = webDriverContext;  }    [Then(@"I should be navigate to search results page")]  public void ThenIShouldBeNavigateToSearchResultsPage()  {    System.Threading.Thread.Sleep(2000);  // After search is complete the keyword should be present in url as well as page title`  Assert.IsTrue(\_webDriverContext.chromeDriver.Url.ToLower().Contains(\_searchContext.searchTerm));  Assert.IsTrue(\_webDriverContext.chromeDriver.Title.ToLower().Contains(\_searchContext.searchTerm));  }    [AfterScenario]  public void AfterScenarioCompletion()  {  \_webDriverContext.chromeDriver.Close();  }  } |

**#3)** **YoutubeSearchHooks.cs**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63 | [Binding]  public sealed class YoutubeSearchHooks  {  [BeforeScenario]  public void BeforeScenario()  {  Console.WriteLine("In Method Before Scenario!");  }    [AfterScenario]  public void AfterScenario()  {  Console.WriteLine("In Method After Scenario!");  }    [BeforeStep("test")]  public void BeforeStep()  {  Console.WriteLine("In Method Before Step!");  }    [AfterStep]  public void AfterStep()  {  Console.WriteLine("In Method After Step!");  }    [BeforeScenarioBlock]  public void BeforeScenarioBLock()  {  Console.WriteLine("In Method Before Scenario block!");  }    [AfterScenarioBlock]  public void AfterScenarioBlock()  {  Console.WriteLine("In Method After ScenarioBlock!");  }    [BeforeFeature]  public static void BeforeFeature()  {  Console.WriteLine("In Method Before Feature!");  }    [AfterFeature]  public static void AfterFeature()  {  Console.WriteLine("In Method After Feature!");  }    [BeforeTestRun]  public static void BeforeTestRun()  {  Console.WriteLine("In Method Before test run!!");  }    [AfterTestRun]  public static void AfterTestRun()  {  Console.WriteLine("In Method After test run!!");  }  } |

***Our upcoming tutorial will brief you on Step Argument Transformations & Specflow tables in detail!***

# Step Argument Transformations & Specflow Tables

By **[Sruthy](https://www.softwaretestinghelp.com/author/remyasthva/)**  Updated June 17, 2023

**Step Argument Transformations & Specflow Tables Tutorial:**

Our previous Specflow tutorial briefed us all about [**Shared & Scoped Bindings, Hooks and Step Reuse**](https://www.softwaretestinghelp.com/specflow-bindings-hooks-step-reuse/) in detail. Here in this tutorial, we will explore more about Step Argument Transformations in Specflow.

Feel free to read through our [**Complete Specflow Training Guide for Beginners**](https://www.softwaretestinghelp.com/specflow-tutorial/) for a clear understanding of the concept. Step Argument Transformation feature of Specflow, allows a user to provide custom transformation for the parameters supplied in the Steps.

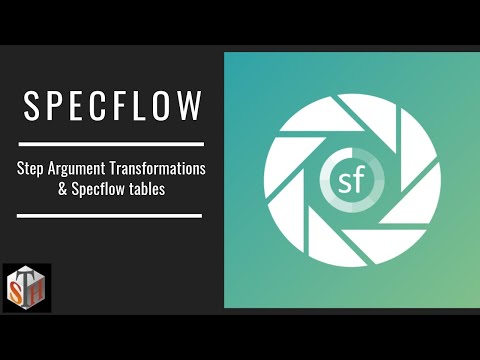
It allows custom logic to be added to convert input parameters into a specific parameter. **For Example,** you can directly create a class object from the parameters and return the constructed object from the transformation function.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/Step-Argument-Transformations-Specflow-tables.png)

Another feature of Specflow that we will look at it is Specflow Tables which allow passing input data in tabular form with a single step and Table helpers can get it mapped directly to an Object instance as desired.

**Watch the VIDEO:**

**Here is a video tutorial on Step Argument Transformations & Specflow tables:**



**Table of Contents:**[[Show](https://www.softwaretestinghelp.com/step-argument-transformations-specflow-tables/)]

## Step Argument Transformations

To understand Argument transformations in a better way, let’s first try to figure out how exactly Specflow does match the parameters. As we’ve seen in our previous articles, for YouTube search example, we were passing the search term as a parameter for the scenario to execute.

The parameter matching usually happens through a regular expression and the matching regex results in setting the method parameter to the supplied search term in the step.

Let’s first try to understand what are the default supported conversions in Specflow and when argument transformations can be helpful.

### Supported Conversions

Specflow does support a lot of conversions out of the box, by looking at the data type itself after regex match. It can automatically take care of conversions like – String, integer, GUID, Enums, etc.

**Let’s see an Example for some of these below:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | Scenario: Get Transactions in my account    Given I have entered customer name as Test Customer  And I have entered customer account id as 0f8fad5b-d9cb-469f-a165-70867728950e  And I select sorting order as DESCENDING  And I select number of transactions to be displayed as 25  Then I should see my account transactions |

In the above code sample, we have highlighted different input types that we are passing in the Steps, and in the step implementations, these are getting converted to the respective data types.

**Let’s see the step implementations for these below (for simplicity we have just did a console out for each of the steps to illustrate that argument supplied is automatically converted to the expected type):**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | [Given(@"I have entered customer name as (.\*)")]  public void GivenIHaveEnteredCustomerNameAsTestCustomer(String customerName)  {  Console.Out.WriteLine(customerName);  }    [Given(@"I have entered customer account id as (.\*)")]  public void GivenIHaveEnteredCustomerAccountIdAs(Guid accountId)  {  Console.Out.WriteLine(accountId.ToString());  }    [Given(@"I select sorting order as (.\*)")]  public void GivenISelectSortingOrderAsAscending(SortOrder sortOrder)  {  Console.Out.WriteLine(sortOrder.ToString());  }    [Then(@"I should see my account transactions")]  public void ThenIShouldSeeMyAccountTransactions()  {  Console.Out.WriteLine("success!");  }    [Given(@"I select number of transactions to be displayed as (.\*)")]  public void GivenISelectNumberOfTransactionsToBeDisplayedAs(int p0)  {  Console.Out.WriteLine(p0.ToString()); |

On executing the above scenario, the output prints all the values successfully by indicating that the auto conversion of arguments to the expected data types was successful.

**This is how the output looks like:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | Given I have entered customer name as Test Customer  Test Customer  -> done: SupportedSpecflowConversions.GivenIHaveEnteredCustomerNameAsTestCustomer("Test Customer") (0.0s)  And I have entered customer account id as 0f8fad5b-d9cb-469f-a165-70867728950e  0f8fad5b-d9cb-469f-a165-70867728950e  -> done: SupportedSpecflowConversions.GivenIHaveEnteredCustomerAccountIdAs(0f8fad5b-d9cb-469...) (0.0s)  And I select sorting order as DESCENDING  DESCENDING  -> done: SupportedSpecflowConversions.GivenISelectSortingOrderAsAscending(DESCENDING) (0.0s)  And I select number of transactions to be displayed as 25  25  -> done: SupportedSpecflowConversions.GivenISelectNumberOfTransactionsToBeDisplayedAs(25) (0.0s)  Then I should see my account transactions  success!  -> done: SupportedSpecflowConversions.ThenIShouldSeeMyAccountTransactions() (0.0s) |

### Argument Transformations

Let’s see an example in action to understand this. Support, you have an application that converts the given time and converts it into minutes. **Example:** If the user input is 1 day – the output is – 1440, if user input is 1 day 2 hour 2 minutes, then the output should be 1562.

Now, it can be seen that to support different types of inputs, one will need to write different binding implementations depending on the type of inputs. **For Example:** For inputs having just daypart, there will be a separate step implementation, for inputs having a day, month part – there will be separate step implementation, etc.

Let’s see how this can be implemented through a single step implementation through Step Argument transformation and the provided input is simply converted into timestamp object and returned to the original step that is called the step transformation.

Think of it as a first level regex scan to your input which returns the partially transformed value to the calling step.

Look at the feature file having 3 different input variants, with a single transform by converting it into complete timespan object and returning back.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | Scenario: Convert timestamp to minutes - variant 1    Given I have entered 50 days into the timestamp to minute converter  When I press calculate  Then the result should be 72000.00 on the screen    Scenario: Convert timestamp to minutes - variant 2    Given I have entered 1 day, 2 hours, 3 minutes into the timestamp to minute converter  When I press calculate  Then the result should be 1563.00 on the screen    Scenario: Convert timestamp to minutes - variant 3    Given I have entered 1 day, 1 hour, 1 minute, 30 seconds into the timestamp to minute converter  When I press calculate  Then the result should be 1501.50 on the screen |

Look at the highlighted values in the above code example. All of these will be taken care of the exact same transformation and the end result will be a transformed TimeSpan input value that is sent back to the calling Specflow Step.

**Let’s look at the implementation of the Transformation below:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | [StepArgumentTransformation(@"(?:(\d\*) day(?:s)?(?:, )?)?(?:(\d\*) hour(?:s)?(?:, )?)?(?:(\d\*) minute(?:s)?(?:, )?)?(?:(\d\*) second(?:s)?(?:, )?)?")]  public TimeSpan convertToTimeSpan(String days, String hours, String minutes, String seconds)  {  int daysValue;  int hoursValue;  int minutesValue;  int secondsValue;    int.TryParse(days, out daysValue);  int.TryParse(hours, out hoursValue);  int.TryParse(minutes, out minutesValue);  int.TryParse(seconds, out secondsValue);    return new TimeSpan(daysValue, hoursValue, minutesValue, secondsValue);  } |

For the framework to know that it is a transformation binding, StepArgumentTransformation Attribute needs to be added to the method implementing the Argument conversion.

**The other important points to be noted regarding Argument conversions is:**

**#1)** Step Argument transformations run for each matching step i.e. irrespective of the type of step i.e. whether it is Given, When or Then, Transformation will happen for each matching regex.

**#2)** Depending upon the return type of the transformed output, if the actual calling step does not have the matching return type for the input parameter, then transformation will not take place.

What this means is, suppose the calling step requires a transformed input but it has the mentioned timestamp of input as something which does not match the return type of the transformed method, then the regex match will be overridden and the conversion will not take place.

**Let’s look at the implementation of calling the “Given” step:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | private TimeSpan ts;    [Given(@"I have entered (.\*) into the timestamp to minute converter")]  public void GivenIHaveEnteredDaysIntoTheTimestampToMinuteConverter(TimeSpan tsTransformed)  {  ts = tsTransformed;  } |

Look at the type of input parameter here i.e. its TimeSpan, which matches the type returned from the transformation step if this is changed to some other type. For instance String, then the argument conversion will not happen and the regex match will be overridden by the original Step implementation.

**Pro Tip:** An important point to note here is the entire text that needs to be transformed should be fed/matched through step argument transformation. Hence, the Given step will now wrap all the possible input formats into a single string and transformation regex will convert it into a TimeSpan object and return back.

## Specflow Tables

Specflow tables are a way to pass a list of the values to step implementation function. In our previous articles, we looked at the way to implement data-driven tests using Scenario outline and Examples. But that was to primarily execute the scenario with different inputs.

Here, in tables, it’s about passing all the data at once in tabular form to the step implementation which is supplying data.

For instance, consider an example where you are testing a Student Management System and in order to create a new Student object, you are requested to fill in a lot of details like first name, last name, age, year of birth, etc.

One way is to pass each of this information as a separate Step which will be essentially a lot of boilerplate code and in each step, you will end up updating the same object that needs to be tested. Another way can be building a complex regex and trying to pass all data in the same step but it’s quite error prone and flaky.

Tables come to our rescue here. All the student related input data can be sent into the same step implementation in a nice tabular way through the table feature of specflow.

**Let’s see a code sample below for Feature and Step implementation:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | Scenario: Pass data through Specflow tables for StudentInfo object    Given I have entered following info for Student  | FirstName | LastName | Age | YearOfBirth |  | test | student | 20 | 1995 |  When I press add  Then i student should get added to database and entered info should be displayed on the screen |

Table data is highlighted in the above Scenario Step.

Specflow does provide a lot of TableHelpers, which directly allow for useful features like creating an object instance from the user-supplied input data rather than parsing each field on your own.

**Let’s see the step implementation below:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | private StudentInfo studInfo;    [Given(@"I have entered following info for Student")]  public void GivenIHaveEnteredFollowingInfoForStudent(Table table)  {  // converting supplied input data directly to instance of StudentInfo object  studInfo = table.CreateInstance<StudentInfo>();  } |

Look at the highlighted section above. Here is just one small line of code, the entire StudentInfo object (which is a POCO containing the student data fields i.e. first name, last name, age, year of birth, etc)

**Some other features/concepts related to Specflow tables are shown below:**

**#1)** Tables can be horizontal or vertical. Vertical tables are more like key-value pairs and in the above scenario more like name-value mappings whereas horizontal tables contain all data for an object in a single row (just like we saw in our example).

**#2)** Vertical tables can be mapped to just a single .NET object whereas horizontal tables can also be mapped to a Set or Collection of Objects.

**#3)** Each field value in the table should be atomic as it will be mapped to a single corresponding field in the parsed object.

An important point to note here is that even if you **auto-generate** step bindings with the tabular data, the Specflow binding generator will automatically account for such input types and recognize it as a valid tabular data.

### Conclusion

In this article, we tried to explain 2 important and handy concepts in Specflow.

The first step is the **Step Argument transformations** which allow custom type conversions for Specflow arguments to avoid boilerplate code (and enables the test script to look more modularized and logical) and the second feature we looked at is **Specflow Tables** which come in handy when you need to pass in a lot of fields/data in a single step in a user friendly tabular format.

***In our upcoming tutorial, we will learn more about, how you can auto-generate beautiful documentation using Specflow in different formats using open source tools like Pickles which can serve an easy reference for all stakeholders of the project.***

**Generate Living Documentation with Pickles For Specflow Feature Files**

By **[Sruthy](https://www.softwaretestinghelp.com/author/remyasthva/)**  Updated March 7, 2024

**How to Generate Living Documentation Through Pickles Using Specflow Files:**

In this [**Beginners Guide on Specflow**](https://www.softwaretestinghelp.com/specflow-tutorial/), we explored all about [**Step Argument Transformations & Specflow tables**](https://www.softwaretestinghelp.com/step-argument-transformations-specflow-tables/) in detail in our previous tutorial.

In this tutorial, we will see how you can generate good looking living documentation through an open source framework called pickles using your existing Specflow files.

This documentation can serve as a ready reference for your documentation and should be a helpful resource for anyone working in the team and can help to understand the various features of the application under test and what kind of output it would produce against different types of supplied inputs.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/Specflow-living-documentation-with-Pickles.png)

**Table of Contents:**[[Show](https://www.softwaretestinghelp.com/specflow-pickles/)]

**What is Pickles?**

Pickles is an Open Source Solution to generate living documentation for files written in Gherkin Syntax (which itself is open source and is used by other tools like Cucumber, RSpec as well).

As Specflow follows Gherkin syntax too, Pickles is a perfect fit to create documentation for Specflow Feature files as well.

**What Documentation Format can I Create?**

**It supports 4 different formats of documentation as shown below:**

1. HTML site with search (or a static HTML site)
2. Word file
3. Excel file
4. Raw JSON – which can be utilized to create documentation with any custom template/library.

**Step by Step Guide For Creating Documentation**

We will be creating the pickles documentation using a Powershell script which uses a Powershell.dll part of the Pickles package that we are going to install.

**There are other options as well like:**

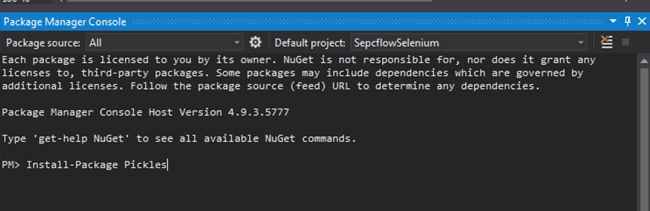
1. Generating documentation through MSBuild scripts.
2. Through Pickles extension package for Visual Studio, it provides a UI to set up options.

**Let’s see the step by step procedure.**

**Step 1:**

Install Pickles package through Package Manager by running the below command in the package manager.

|  |  |
| --- | --- |
| 1 | Install-Package Pickles |

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/PicklesPakage.png)

**Step 2:**

Now, let’s create a Powershell script to create the documentation for the features. We will be mentioning the required parameters that are used for generating documentation.

Create a Scripts folder in the project and create a new file i.e. *Pickles\_documentation.ps1*

**This will contain the required config parameters as shown below:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | # Setup variables  $root = "{rootDirOfProject}"  $FeatureDirectory = "$root\Features"  $OutputDirectory = "$root\Docs\{DocumentationType}"  $DocumentationFormat = "word"    # Import the Pickles-comandlet  Import-Module $root\packages\Pickles.2.20.1\tools\PicklesDoc.Pickles.Powershell.dll    # Call pickles  Pickle-Features -FeatureDirectory $FeatureDirectory `  -11111OutputDirectory $OutputDirectory `  -DocumentationFormat $DocumentationFormat |

**Let’s try understanding these parameters in detail:**

**#1) root:** This is the root directory of the Visual Studio Project that we want to create our documentation for

**#2) FeatureDirectory:** Path of the Features folder (i.e. the folder/directory containing all feature files or specs)

**#3) OutputDirectory:** Path to the output folder where documentation will get generated. Refer to the {DocumentationType} parameter here. This is for simplicity purpose and we will be creating documentation of different formats in different folders.

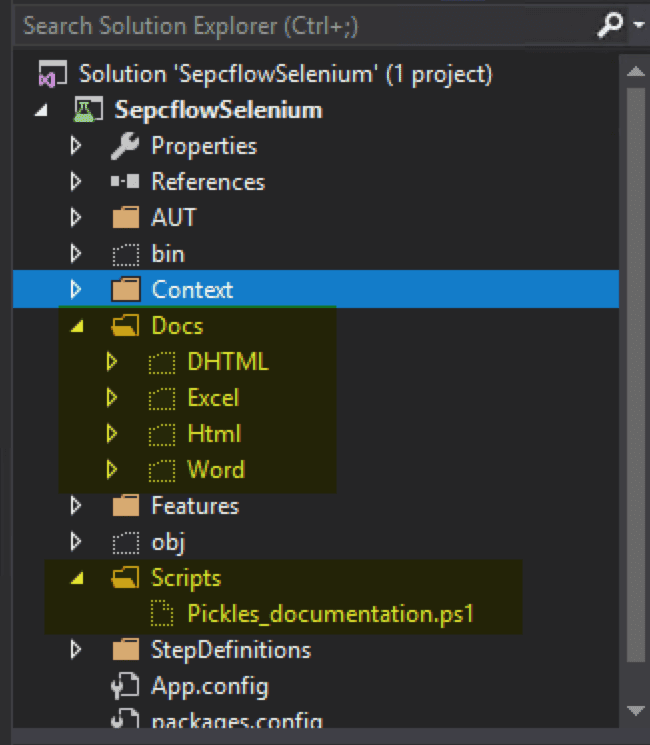
**#4) DocumentationFormat:**The format in which we expect the documentation to get generated.

**The supported formats are:**

* DHTML (static html site)
* HTML (Different html pages for each spec or feature)
* Word (word / open office format)
* Excel (excel / open office format)

**#5)** The ImportModule commandlet expects the path to the ***Powershell dll*** library for pickles package and this can vary Depending on the version of Pickles installed.

After adding the Powershell script to the Specflow project and creating placeholders for different documentation formats, the project structure should look as shown below:

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/PackageView.png)

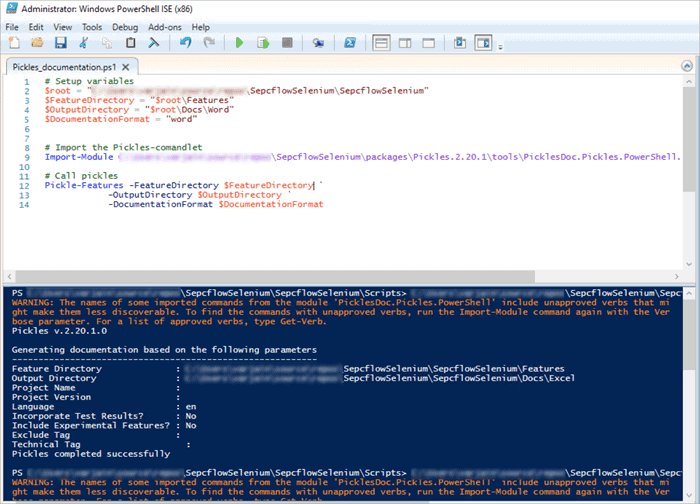
**Step 3:**

Let’s try executing this Powershell script now, and see how the generated documents would look like. We will see for 3 different formats i.e. DHTML, HTML, and Word

Let’s first run for format “Word” – In our Powershell script we will change 2 parameters i.e. OutputDirectory will contain a folder named “Word” to store the generated document, and the DocumentationFormat parameter will be *“word”.*

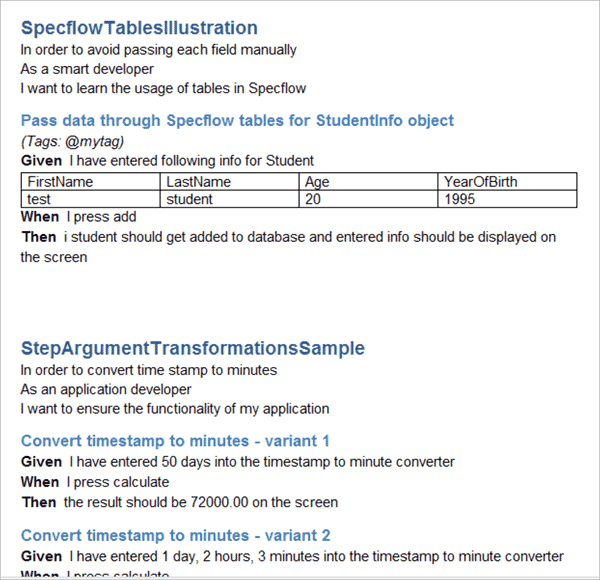
The script can be executed by opening the script in Power Shell script editor and executed from there.

**Refer to the screenshot below for more details.**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/ExecuteDocumentationScript.png)

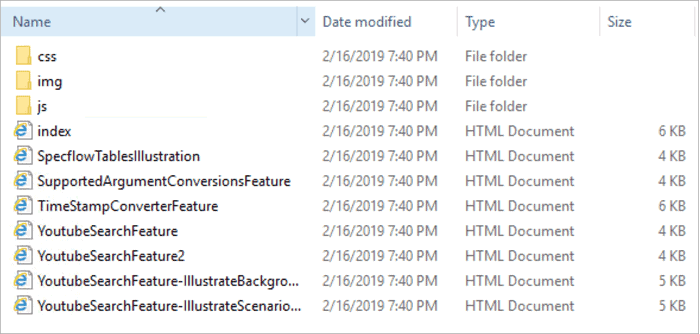
**Let’s see how different documentation format’s look like:**

**Word Format:** This is a formatted word document and is a single doc file containing all the features and scenarios.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/Word_Format.png)

**HTML Format:** This representation creates individual HTML files for each of the feature with separate formatting files for CSS and JS (This is the ideal format to place or host on a web server for documentation hosting over the web).

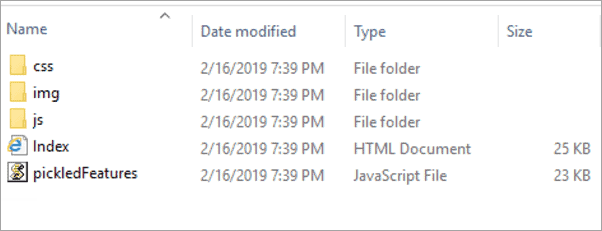
**The folder structure for HTML format should look as shown below.**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/Html_FolderStructure.png)

**DHTML format:**This creates a single static web page for all the features and scenarios present in the application.

This comes in really handy as it provides search feature as well to search for features/scenarios and is really helpful for large applications containing features/scenarios spanning multiple files were looking for a specific feature or scenario can be achieved directly by doing a search.

**The DHTML folder structure will look as shown below:**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/DHTML_FolderStructure.png)

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/DHTML_Format.png)

**Other Alternatives**

Similar to Pickles, there are other document generators as well like Relish & Speclog. But both of them are not free (and the free versions have limited or restricted usage only). For more info, you can refer to [Pickles Packages](http://www.picklesdoc.com/).

**Conclusion**

In this tutorial, we discussed all about creating living documentation for our feature files, which in turn provide an easy, user-friendly way to browse/search for different scenarios & features and can serve as an easy reference for the application for which features are created.

As Specflow features are already written in plain English language, it makes complete sense to document product’s Business requirements in the form which really makes it a single source of truth guide for all stakeholders involved in the application right from the Product to a developer and the tester.

***Check out our upcoming Specflow tutorial to know more about the ways to Generate Test Reports and Execute Selective tests!***

**Specflow Reporting: How to Generate Test Reports and Execute Selective Tests**

By **[Sruthy](https://www.softwaretestinghelp.com/author/remyasthva/)**  Updated June 28, 2023

**Specflow Report Generator Tutorial: Ways to Generate Test Reports and Execute Selective Tests in Specflow:**

In this [**Specflow Training Tutorials for All**](https://www.softwaretestinghelp.com/specflow-tutorial/), we explored all about [**Generating Living Documentation with Pickles**](https://www.softwaretestinghelp.com/specflow-pickles/)in our previous tutorial.

In this tutorial, we will discuss the various techniques to create Reports for the tests written in Specflow framework.

The type of report generated also largely depends upon the type of test runner that is being used. Both free and paid variants of runners are available with each having their own pros and cons.

We will be discussing more on creating reports using a combination of NUnit as test runner and Specflow reporting templates to generate HTML reports out of the NUnit generated test output files.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/SPECFLOW-REPORTING.png)

**Table of Contents:**[[Show](https://www.softwaretestinghelp.com/specflow-reporting/)]

**Specflow Reporting**

**Watch the VIDEO: Watch the Video tutorial on Specflow Reporting**



**Generating Test Reports – Prerequisites**

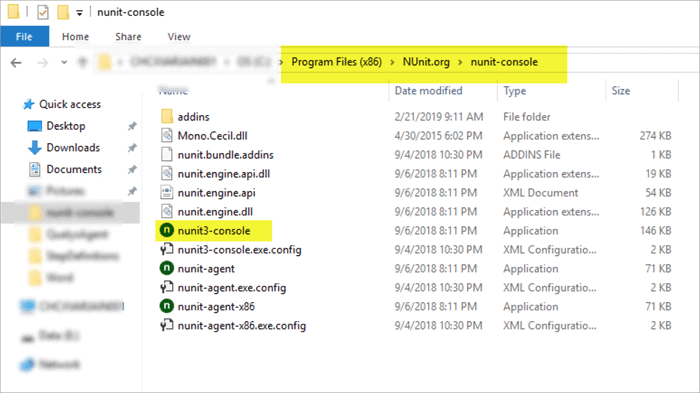
Before starting with generating a test report, let’s ensure that all the prerequisites are available or installed.

We will use the same test example of YouTube search which uses Specflow template with NUnit test runner. In order to start generating reports, you will need the NUnit console to execute the Specflow tests.

Install NUnit console by downloading the executable package file for windows [**here**](https://github.com/nunit/nunit-console/releases/tag/v3.9). Download NUnit-Console -3.9.0 MSI package and complete the step by step installation of NUnit Console by running the downloaded package file.

Once installed successfully, you should be able to see the Nunit folder in the ProgramFiles folder.

**Refer to the screenshot below:**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/NUnitInstallLocation.png)

Specflow report generation also works with other frameworks like MSUnit in the same way as NUnit.

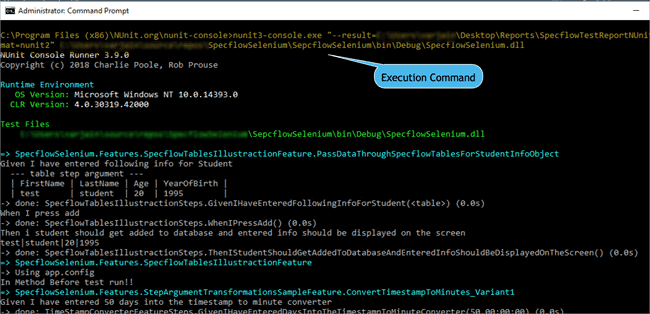
**Step by Step Process**

**#1)** Execute the tests using NUnit Console to create an XML report.

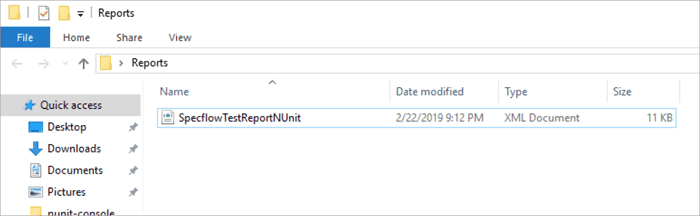
* Compile the SpecflowProject in Visual Studio and ensure that there is no compile errors project dll file created. The created DLL file should be available in the Debug/Release folder of the Visual Studio Project.
* Now Open the Command Prompt and Navigate to the base location of the NUnit Console.
* Run the following command in order to execute the Specflow tests.

**Let’s see the syntax first:**

|  |  |
| --- | --- |
| 1 | nunit3-console.exe "result={PathToReportLocation};format=nunit2" {SpecflowProjectDllPath} |

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/Specflow_NUnit_Report_Generation.png)

Once the above command is executed, you will see all your scenarios in the feature files to start getting executed. After the test completion, ensure that the XML result file from the NUnit execution gets generated at the location specified in the **result** parameter in the above command.

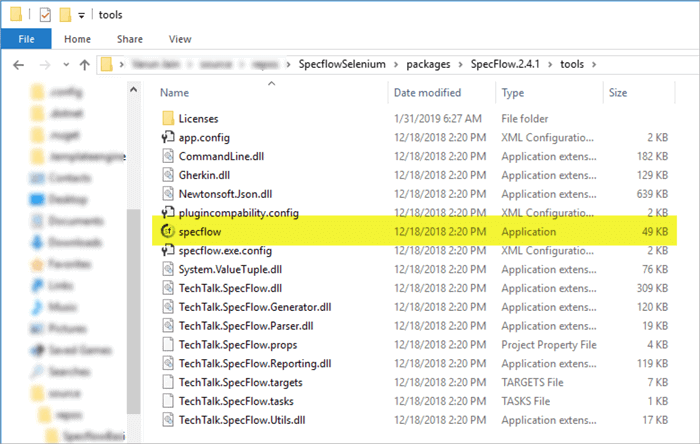
[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/05/NUnitReportLocation.png)

Also please note here that the format of the generated report that we have is selected as “NUnit2” as the current version of Specflow does not support NUnit 3. Once this support is available, the report format option can be removed.

**#2)** Now use the specflow.exe application to create the HTML report.

The specflow.exe application runs over the XML report that’s generated by NUnit and generates good looking HTML reports. specflow.exe is an executable application present in the npm package folder of the project where specflow was installed.

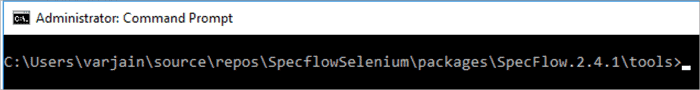
**See the screenshot below for locating the Specflow Executable.**

*[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/SpecflowExecutableLocation.png)*

The report generated by this application is of HTML format and contains a summary of executed Scenarios grouped by features.

Execute the following steps to generate an HTML report out of the XML report that got generated through NUnit console execution.

* Open the command prompt and navigate to the Specflow executable.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/SpecflowConsole.png)

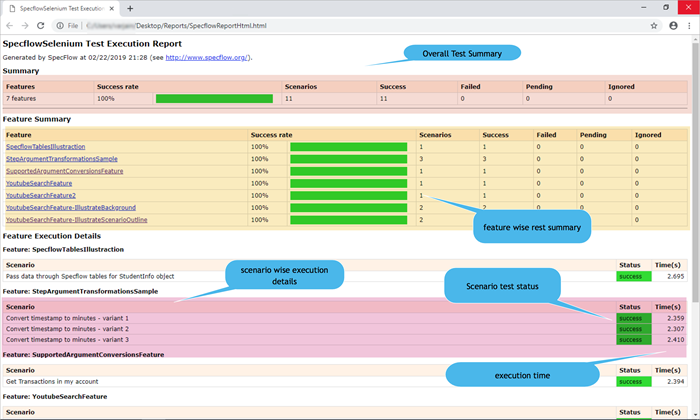
* Now run the command with the below syntax to generate the HTML report.

|  |  |
| --- | --- |
| 1 | specflow.exe nunitexecutionreport --ProjectFile {PathToProjectSln.csproj} --xmlTestResult {PathToNUnitReportXml} --OutputFile {DesiredPathToGenerateHtmlReport} |

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/SpecflowReportExecutionCommand.png)

* Once the above command completes execution, ensure that the HTML report gets generated at the specified folder location.

Let’s open the generated report and see the different sections. The important sections depicted by the report are highlighted below.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/SpecflowHTMLReport.png)

As you can see, the above report is far more easy to read and interpret and is useful for all stakeholders of the project to get a glimpse of the overall test execution summary.

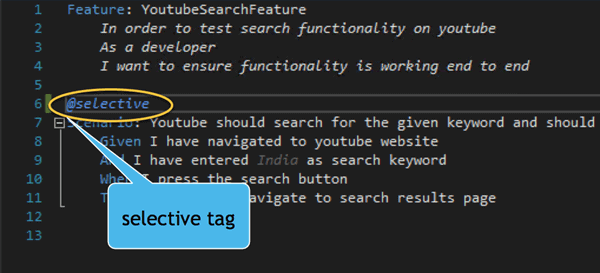
**Executing Selective Tests**

Very often, you are required to execute a few selective tests or a subset of the entire test suite or tests marked with some special categories. All this is possible via Specflow tags that we saw in our previous articles (which can be applied both at Feature level as well as Scenario level).

Thus, in order to execute just a subset of tests, we can use Specflow tags to mark/label such scenarios and while running through the NUnit console runner the categories can be mentioned as a part of the command.

**Let’s see the step by step process to perform selective test execution:**

**#1)** Open the Visual Studio Project and let’s tag a Scenario/Feature which we want to run selectively with a tag/category named as “selective”.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/SelectiveTag.png)

**#2)** Compile the solution for the changes to take place and a new DLL with updated changes gets created.

**#3)** Now open the command prompt and follow the steps as mentioned above for executing the tests with NUnit console application.

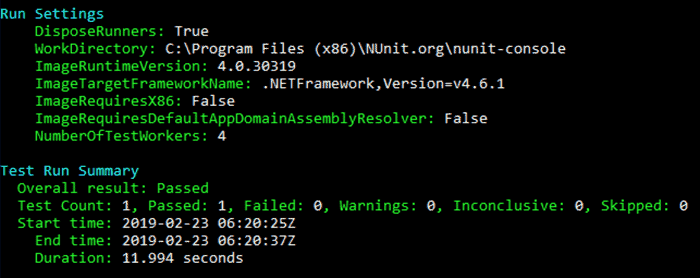
**#4)** The NUnit executable looks at the test filter and tries to search for all the tests matching the specified filter. In our case, it will try to match all the feature/scenarios with category “selective” and execute only those tests.

Since we just tagged one scenario with this tag, we should expect the NUnit test runner to execute just one test.

|  |  |
| --- | --- |
| 1 | nunit3-console.exe "--result={PathToReportFile};format=nunit2" --where "cat == selective" {PathToSpecflowSeleniumProjectDll} |

As you can see above, we’ve mentioned filter like “cat == selective”. The NUnit application interprets it as a category filter with the value “selective”.

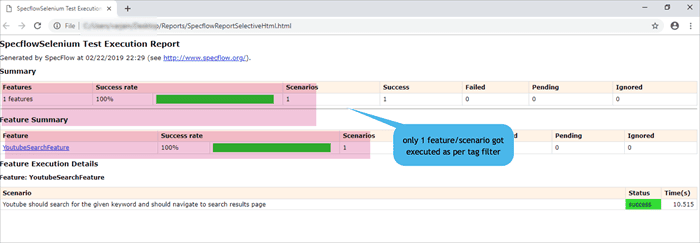
**#5)** Once the above command execution completes, we can see the test summary on the console (as well as the generated XML report by NUnit).

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/ConsoleTestSummary.png)

**#6)** Now, we can create an HTML report out of it (by following the steps to generate HTML through Specflow executable). This is how the generated HTML will look like.

As NUnit just executed one test, the generated HTML report should show only one feature and one scenario.

**Let’s see how the HTML report looks like:**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/03/SelectiveHtmlReport.png)

**Conclusion**

In this tutorial, we explored the different ways of executing Specflow feature tests along with generating HTML reports through the Specflow executable. The generated reports are easy to read & interpret and are useful for all the stakeholders of the project.

We also saw the techniques to execute a subset of tests rather than the entire test suite through Specflow tags which enables NUnit to look/filter for specific scenarios and execute only those which match the specified filter criteria.

***Check out our upcoming tutorial to explore the Most Frequently Asked Interview Questions on Specflow!***