SpecFlow+ Runner

Jan 27, 2022

Migration

SpecFlow+ Runner is a dedicated test runner for SpecFlow that integrates directly with Visual Studio. You can also execute tests from the *command line*.

SpecFlow+ include the following features:

- Visual Studio integration (syntax coloring, auto completion, navigation, skeleton generation, Gherkin table formatting)
- Visual Studio Test Explorer integration (business readable scenario names, run/debug from feature file)
- Advanced test runner features: *execution profiles*, *target environments*, attach artifacts to test execution reports, TFS integration, TeamCity integration, *complex scenario filter expressions*
- Dedicated support for integration test execution (*config file transformation and deployment steps*, test targets: re-run tests for different environments and configurations, detect flickering scenarios)
- Parallel test execution (app domain isolation, process isolation, execution results consolidation)
- Advanced test execution scheduling (run previously failing tests first, retry scenarios, execution history available as OData)
- Customizable HTML *reports (Razor templates)*

If you are new to SpecFlow+, we recommend you to check out the Getting Started guides as a starting point.

A number of example projects have also been put together by the SpecFlow team and the community here .

CHAPTER 1

Migrating away from the SpecFlow+ Runner

The last official version of SpecFlow thats supports the SpecFlow+ Runner is `3.9.40`. If you add a reference to any newer version of SpecFlow in your project, then it is not guaranteed that the runner will continue to function as intended.

The information below outlines the differences between the options available as an alternative to the SpecFlow+ Runner and how to migrate your project to them safely.

1.1 Overview

Here is a simple overview that breaks down each alternative option to the SpecFlow+ Runner and its parallelisation and reporting features.

Runner	Parallelisation	Reporting
NUnit	Feature level	SpecFlow+ LivingDoc, NUnit report (prior to SpecFlow 3)
xUnit	Feature level	SpecFlow+ LivingDoc
MSTest	Feature level	SpecFlow+ LivingDoc, MsTest Test Execution Report (prior to SpecFlow 3)

1.2 Migration

1.2.1 NUnit

1. Packages
   - Remove the following packages from your project:
     - `SpecRun.SpecFlow`
   - Add the following packages from your project:
     - for SpecFlow to generate the code behind files discoverable by the NUnit execution engine

        * `SpecFlow.NUnit`
        * `NUnit`
    - for Test Discovery & Execution
        * `NUnit3TestAdapter`
        * `Microsoft.NET.Test.Sdk`

2. Parallelisation
    - NUnit offers parallel execution only at *feature* level when used with SpecFlow. This means that if you set the `LevelOfParallelism` to 2, then 2 features will execute their tests at the same time. This can be achieved by doing the following:
        - Add a new C# file to your project, for example `AssemblyInfo.cs`

```
using NUnit.Framework;

[assembly: Parallelizable(ParallelScope.Fixtures)]
[assembly: LevelOfParallelism(2)]
```

        - Note: SpecFlow does not support scenario level parallelization with NUnit (when scenarios from the same feature execute in parallel). If you configure a higher level NUnit parallelization than "Fixtures" your tests will fail with runtime errors.

3. Reporting

- **If you are using SpecFlow 3**, then you can generate reports using SpecFlow+ LivingDoc
    - SpecFlow+LivingDoc for Azure DevOps: If your team uses Azure DevOps then we suggest installing our dedicated extension to help you generate and share LivingDoc within the familiar Azure DevOps interface.
    - SpecFlow+LivingDoc Generator: If you want to generate a self-hosted HTML documentation with no external dependencies so you have the freedom to share it as you wish, then we suggest the SpecFlow plugin and command-line tool.
- **If you are using SpecFlow 2 or lower**, Then see here.

1.2.2 xUnit

1. Packages
    - Remove the following packages from your project:
        - `SpecRun.SpecFlow`
    - Add the following packages from your project:
        - for SpecFlow to generate the code behind files discoverable by the xUnit execution engine
            * `SpecFlow.xUnit`
            * `xUnit`
        - for Test Discovery & Execution
            * `xunit.runner.visualstudio`
            * `Microsoft.NET.Test.Sdk`
2. Parallelisation

  - By default xUnit runs all SpecFlow features in parallel with each other. No additional configuration is necessary.

3. Reporting

- you can generate reports using SpecFlow+ LivingDoc
  - SpecFlow+LivingDoc for Azure DevOps: If your team uses Azure DevOps then we suggest installing our dedicated extension to help you generate and share LivingDoc within the familiar Azure DevOps interface.
  - SpecFlow+LivingDoc Generator: If you want to generate a self-hosted HTML documentation with no external dependencies so you have the freedom to share it as you wish, then we suggest the SpecFlow plugin and command-line tool.

1.2.3 MSTest

1. Packages
   - Remove the following packages from your project:
     - `SpecRun.SpecFlow`
   - Add the following packages from your project:
     - for SpecFlow to generate the code behind files discoverable by the MSTest execution engine
       * `SpecFlow.MSTest`
       * `MSTest.TestFramework`
     - for Test Discovery & Execution
       * `MSTest.TestAdapter`
       * `Microsoft.NET.Test.Sdk`
2. Parallelisation
   - MSTest offers parallel execution only at *feature* level when used with SpecFlow. This can be achieved by doing the following:
     - Add a new C# file to your project, for example `AssemblyInfo.cs`

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
[assembly: Parallelize(Scope = ExecutionScope.ClassLevel)]
```

       * Note: SpecFlow does not support scenario level parallelization with MsTest (when scenarios from the same feature execute in parallel). If you configure a higher level MsTest parallelization than 'ClassLevel' your tests will fail with runtime errors.
3. Reporting

- **If you are using SpecFlow 3**, then you can generate reports using SpecFlow+ LivingDoc
  - SpecFlow+LivingDoc for Azure DevOps: If your team uses Azure DevOps then we suggest installing our dedicated extension to help you generate and share LivingDoc within the familiar Azure DevOps interface.
  - SpecFlow+LivingDoc Generator: If you want to generate a self-hosted HTML documentation with no external dependencies so you have the freedom to share it as you wish, then we suggest the SpecFlow plugin and command-line tool.
- **If you are using SpecFlow 2 or lower**, Then see here.

CHAPTER 2

Installation

SpecFlow+ Runner (SpecRun) is distributed as a NuGet package, see the link below:

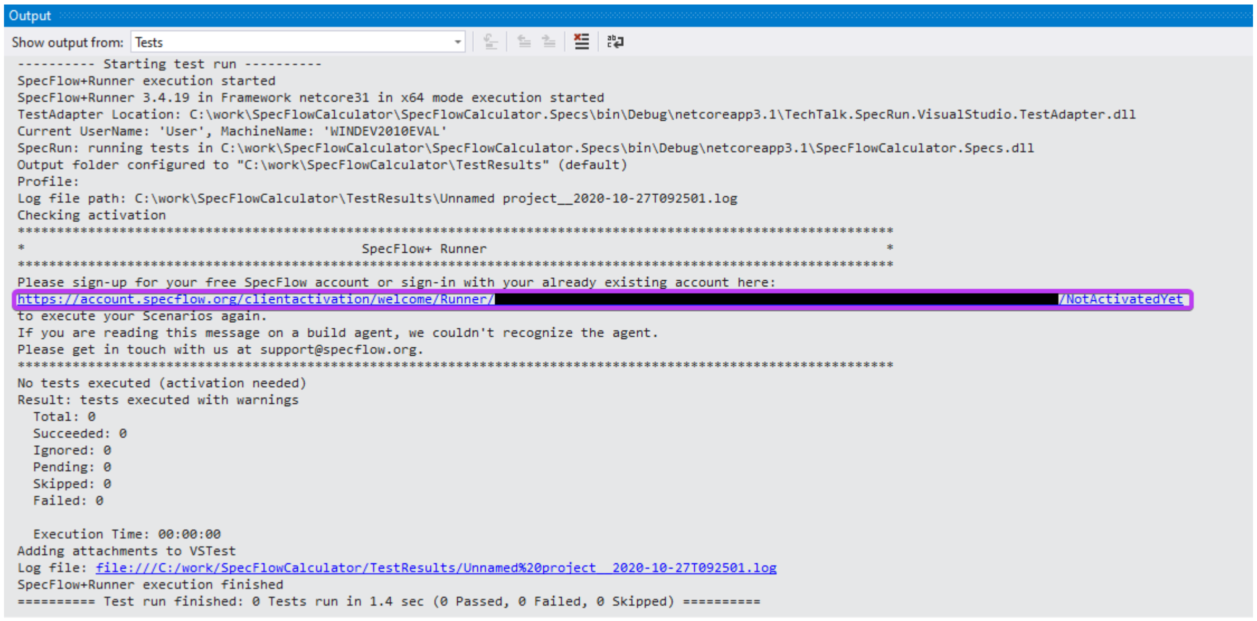
- SpecFlow+ Runner NuGet Package

> *Note: Installation of this package also automatically installs the SpecFlow NuGet package.*

If you are **new to SpecFlow** and the SpecFlow+ ecosystem please check out our Getting Started guide. You will be guideded step by step through a sample project where you will learn how to install and use SpecFlow's core proudcts; SpecFlow, SpecFlow+Runner, and SpecFlow+LivingDoc.

2.1 Activation

1- If you picked SpecFlow+ Runner as your preferred test runner,a message is written to the console when you try to execute your tests with SpecFlow+ Runner for the first time. In Visual Studio it looks like this:



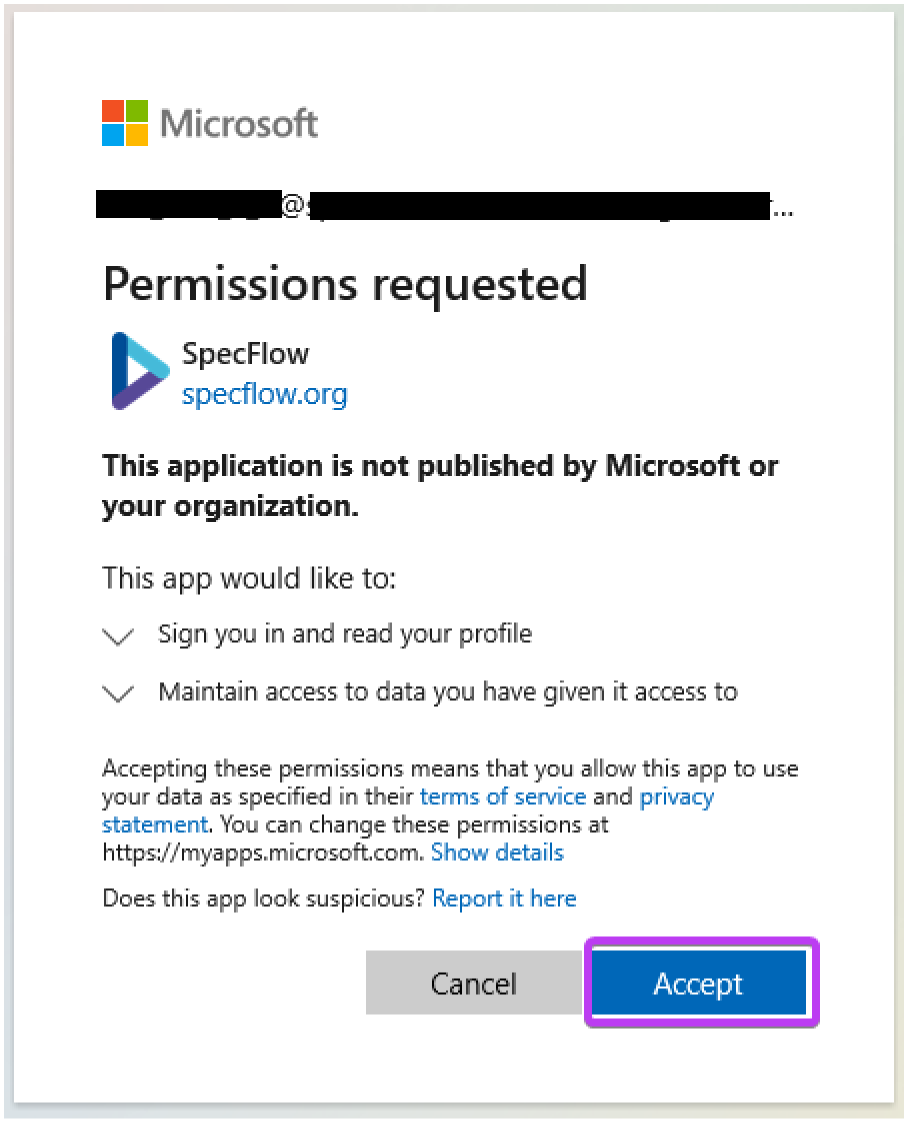
```
---------- Starting test run ----------
SpecFlow+Runner execution started
SpecFlow+Runner 3.4.19 in Framework netcore31 in x64 mode execution started
TestAdapter Location: C:\work\SpecFlowCalculator\SpecFlowCalculator.Specs\bin\Debug\netcoreapp3.1\TechTalk.SpecRun.VisualStudio.TestAdapter.dll
Current UserName: 'User', MachineName: 'WINDEV2010EVAL'
SpecRun: running tests in C:\work\SpecFlowCalculator\SpecFlowCalculator.Specs\bin\Debug\netcoreapp3.1\SpecFlowCalculator.Specs.dll
Output folder configured to "C:\work\SpecFlowCalculator\TestResults" (default)
Profile:
Log file path: C:\work\SpecFlowCalculator\TestResults\Unnamed project__2020-10-27T092501.log
Checking activation
****************************************************************************************************************
*                                           SpecFlow+ Runner                                                  *
****************************************************************************************************************
Please sign-up for your free SpecFlow account or sign-in with your already existing account here:
https://account.specflow.org/clientactivation/welcome/Runner/[redacted]/NotActivatedYet
to execute your Scenarios again.
If you are reading this message on a build agent, we couldn't recognize the agent.
Please get in touch with us at support@specflow.org.
****************************************************************************************************************
No tests executed (activation needed)
Result: tests executed with warnings
  Total: 0
  Succeeded: 0
  Ignored: 0
  Pending: 0
  Skipped: 0
  Failed: 0

  Execution Time: 00:00:00
Adding attachments to VSTest
Log file: file:///C:/work/SpecFlowCalculator/TestResults/Unnamed%20project__2020-10-27T092501.log
SpecFlow+Runner execution finished
========== Test run finished: 0 Tests run in 1.4 sec (0 Passed, 0 Failed, 0 Skipped) ==========
```

2- Copy the activation link from the test output and open it in your browser.

3- You should now see a welcome screen, click the **Sign in with Microsoft** button. Preferably, use your work or student Microsoft account to sign in, but if your professional account is restricted and you run into issues you can always use your personal Microsoft account.

> *If run into issues here, it might be that your Active Directory admin needs to grant you and your team permission to use the SpecFlow+ Runner due to your organizations' Active Directory configuration. Learn more about admin consents here.*

4- After the authentication with the selected account, Microsoft will ask for your permission to sign in to Specflow and share your basic profile information with Specflow. You have to "Accept" the permission request to continue.



Microsoft

[redacted]@[redacted]...

Permissions requested

SpecFlow
specflow.org

This application is not published by Microsoft or your organization.

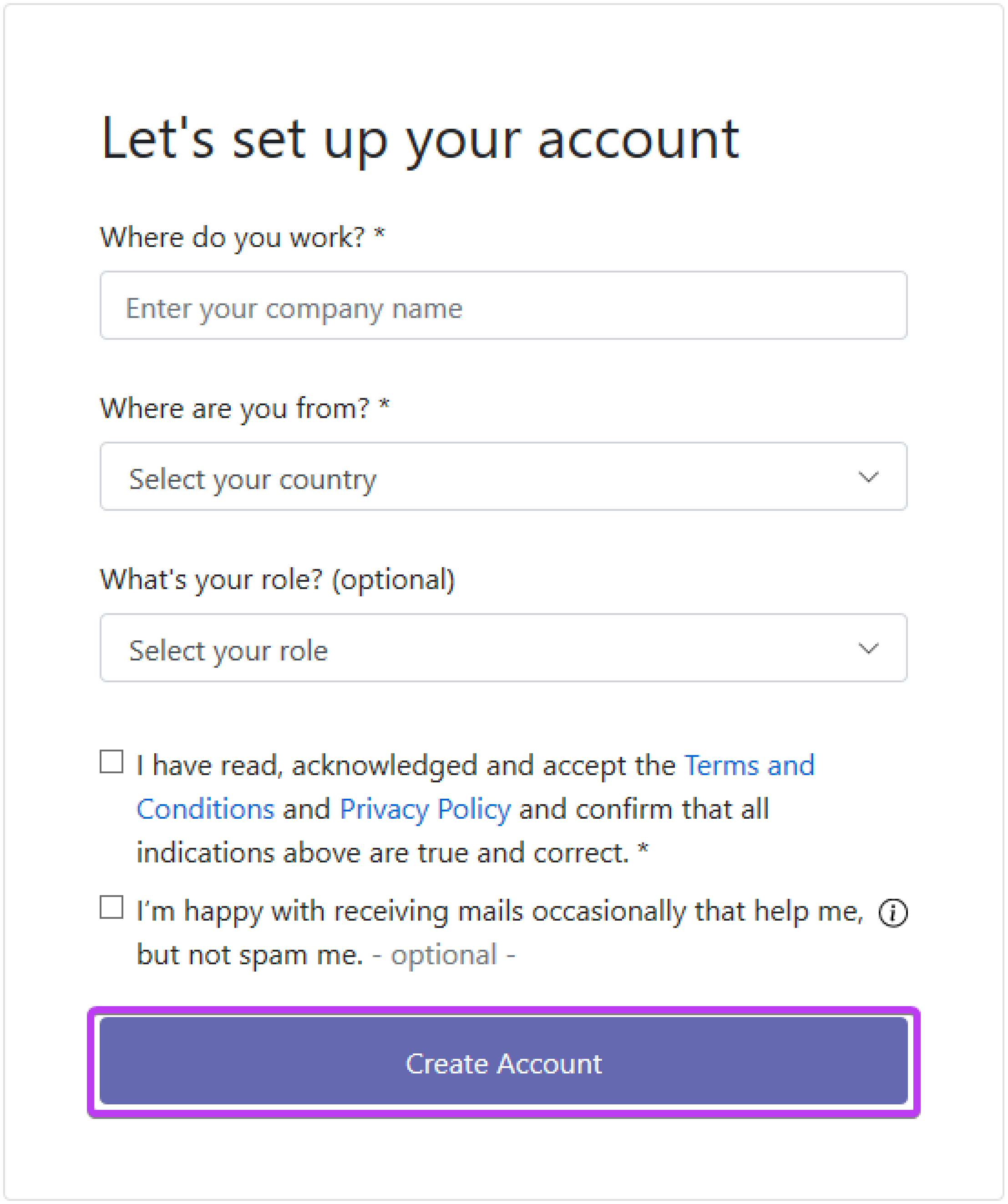
This app would like to:

- Sign you in and read your profile
- Maintain access to data you have given it access to

Accepting these permissions means that you allow this app to use your data as specified in their terms of service and privacy statement. You can change these permissions at https://myapps.microsoft.com. Show details

Does this app look suspicious? Report it here

5- Next, you will be asked three simple questions, fill them in and hit create account.



Let's set up your account

Where do you work? *

Enter your company name

Where are you from? *

Select your country

What's your role? (optional)

Select your role

☐ I have read, acknowledged and accept the Terms and Conditions and Privacy Policy and confirm that all indications above are true and correct. *

☐ I'm happy with receiving mails occasionally that help me, but not spam me. - optional -

6- After you finished the sign-up, return to your IDE and run your tests again.

> ***Note:*** *Please note that an activation is needed for each user/machine.*

- This video is part of our Getting Started guide which covers the **activation** of SpecFlow+Runner. If you are new to SpecFlow, we highly recommend that you go through this guide from the first step. You will learn the basics on not only SpecFlow+Runner but also SpecFlow and SpecFlow+LivingDoc.

CHAPTER 3

Supported OS and .NET Versions

3.1 Supported Operating Systems

- Windows
- Linux (not in Docker)
- Mac OS

3.2 Supported .NET versions

- .NET Framework >= 4.6.1
- .NET Core >= 2.1
- .NET 5

> ***Note:*** .NET 6 is currently **not** supported

CHAPTER 4

Installing the Licensing Tool (V3.0 & 3.1)

This is only needed for SpecFlow+ 3.1 and earlier. From SpecFlow 3.2 onwards you need to sign up for a free SpecFlow account in order to use SpecFlow+. Learn more about the SpecFlow account here.

4.1 SpecFlow.Plus.License

Until SpecFlow+ 3.1, licenses need to be registered using the SpecFlow.Plus.License tool. It is used to assign, remove and display the current licensing status of a machine.

4.1.1 Setup

Prerequisites

SpecFlow.Plus.License requires the .NET Core SDK 2.1 or higher to be installed. Information on setting up the .NET Core SDK can be found in the official Microsoft guide.

Installing SpecFlow.Plus.License

To install SpecFlow.Plus.License:

1. Open a command prompt.
2. Run the following command:`dotnet tool install --global SpecFlow.Plus.License`**Note:** If you want to install a specific version, use the `--version` option to specify the desired version:`dotnet tool install --global SpecFlow.Plus.License --version <desired version>`
3. You can test that the installation was successfull and display your license status using the following command:`specflow-plus-license about`

Uninstalling SpecFlow.Plus.License

To uninstall SpecFlow.Plus.License:

1. Open a command prompt.
2. Run the following command:`dotnet tool uninstall --global SpecFlow.Plus.License`

Updating SpecFlow.Plus.License

To update SpecFlow.Plus.License:

1. Open a command prompt.
2. Run the following command:`dotnet tool update --global SpecFlow.Plus.License`

4.1.2 Migrating Previous Licenses

Prior to SpecFlow 3, license keys were stored in the registry. However the .NET Core licensing tool cannot access licenses stored in the registry. If you intend to use SpecFlow+ with .NET Core and have already registered your license key, you can migrate your old key using the following command: `specrun migrate-license`

This will copy your existing license from the registry to a local file that can be read by the .NET Core licensing tool.

CHAPTER 5

Registering SpecFlow+ (V3.0 & 3.1)

This is only needed for SpecFlow+ 3.1 and earlier. From SpecFlow 3.2 onwards you need to sign up for a free SpecFlow account in order to use SpecFlow+. Learn more about the SpecFlow account here.

SpecFlow.Plus.License is used to register/unregister your license and display information on your current license. Information on installing the licensing tool can be found *here*.

SpecFlow.Plus.License uses the getopt syntax. Options need to be specified with a double dash before the option, e.g. `specflow-plus-license register --help` displays the help for the `register` command.

5.1 Commands

5.1.1 register

This command registers a license key for the current Windows user. The license is written to a file stored in %appdata%\TechTalk\SpecFlowPlus\license.

Syntax

```
specflow-plus-license register --licenseKey <LicenseKey> --issuedTo <IssuedTo>
<options>
```

The license key and licensee (issued to) are required to register a license. **If the licensee contains a space, make sure to enclose it in quotes.**

You will receive an email containing your license key and the licensee (issued to). **Please keep the email with your license details in case you need to register the license again on a different machine!**

Example:

```
specflow-plus-license register --licenseKey ABCDEFG123 --issuedTo "John Doe"
```

5.1.2 Migrating Previous Licenses

Prior to SpecFlow 3, license keys were stored in the registry. However the .NET Core licensing tool cannot access licenses stored in the registry. If you intend to use SpecFlow+ with .NET Core and have already registered your license key, you can migrate your old key using the following command: `specrun migrate-license`

This will copy your existing license from the registry to a local file that can be read by the .NET Core licensing tool.

5.1.3 unregister

This command removes the license registered to the current Windows user.

Syntax

```
specflow-plus-license unregister
```

5.1.4 about

This command displays information about the license registered on the current machine, as well as information on the version of the licensing tool.

Syntax

```
specflow-plus-license about
```

Sample Output

```
********************************
*** SpecFlow+ Licensing Tool ***
********************************

Version 1.8.5+g20ae9ada16
Assembly Version 1.8.0.0
Released 2015-03-03

Copyright c 2011-2018 TechTalk
http://www.specflow.org/plus

Build date: 2015-03-03
The registered license is registered for John Doe and is will expire <expiration date>
↪.
```

5.1.5 version

This command displays the assembly's version information.

Syntax

```
specflow-plus-license version
```

Note: You can also append `--version` as an option to any available command to display the version information, e.g. `specflow-plus-license register --version`.

5.1.6 help

This command displays the help information, either for a specific command, or for all available commands.

Example: `specflow-plus-license help register`

Syntax

```
specflow-plus-license help <command>
```

The command is optional; if no command is specified, the help information for all commands is displayed.

Note: You can also append `--help` as an option to any available command to display the help for that command, e.g. `specflow-plus-license register --help`.

CHAPTER 6

Registering SpecFlow+ (earlier versions)

6.1 SpecFlow+ Runner

Installing SpecFlow+ Runner's NuGet packages installs SpecFlow and SpecFlow+ Runner to the `/packages` folder of your Visual Studio solution. After installing the packages, register your license by starting `SpecRun.exe` (in `/packages/SpecRun.Runner.x.y.z/tools`) from the command line using the following syntax:

SpecRun.exe register <LicenseKey> "<Licensee>"

Replace the placeholders (<LicenseKey>, <Licensee>), including the angled brackets, with your license key and licensee. For example:

SpecRun.exe register –licenseKey KBh8227Ahb9382QAAA=== –issuedTo "ACME Corp."

IMPORTANT: The licensee (issued to value) is **case-sensitive**!

Note: If you purchased SpecFlow+ via SWREG, the licensee is the email address you used to make the purchase and that the license key was sent to.

6.2 SpecFlow+ Excel

Installing SpecFlow+ Excel's NuGet packages installs SpecFlow and SpecFlow+ Excel to the `/packages` folder of your Visual Studio solution. After installing the packages, register your license by starting `SpecFlow.Plus.Excel.Converter.exe` (in `/packages/SpecRun.Plus.Excel.x.y.z/tools`) from the command line using the following syntax:

SpecFlow.Plus.Excel.Converter.exe register <LicenseKey> "<Licensee>"

Replace the placeholders (<LicenseKey>, <Licensee>), including the angled brackets, with your license key and licensee, for example:

SpecFlow.Plus.Excel.Converter.exe register KBh8227Ahb9382QAAA=== "ACME Corp."

IMPORTANT: The licensee (issued to value) is **case-sensitive**!

CHAPTER 7

Configure Visual Studio Test Explorer Integration

SpecFlow+ Runner provides tighter integration with the Visual Studio Test Explorer. Once you have built your solution, your tests are displayed with their business-readable scenario titles in the Test Explorer.

When running tests from the Test Explorer window, tests are executed using the following defaults.

7.1 Profile

SpecFlow+ Runner uses *test profiles* to configure a variety of settings that determine how tests are executed. By default, the Test Explorer uses the `VisualStudio.srprofile` file in your project, if present. If the file does not exists, the `Default.srprofile` file is used instead.

You can specify a different profile in your .runsettings file (see below).

7.2 Processor Architecture

Unless specified otherwise in the test profile, tests are executed using the test processor architecture setting in Visual Studio. By default, this is set to `x86` in Visual Studio.

To change this setting in Visual Studio, select **Test|Test Settings|Default Processor Architecture|** from the menu and choose the desired architecture (X86 or X64).

7.3 Report file

By default, the name of the *report* generated by SpecFlow+ Runner is based on the project name and current time.

7.4 Custom Configuration

To customize your configuration, you need to use a combination of the Visual Studio test settings file and the SpecFlow+ Runner test profile. This requires the following steps:

1. Add a test settings file to your project based on the template located in `packages/SpecRun.Runner.{version}/docs/Sample.runsettings` using **Add Existing Item…** in Visual Studio.
2. Change the settings in the `.runsettings` file as needed (see below).
3. Select **Test|Test Settings|Select Test Settings** from the menu and choose your file.

7.4.1 General run settings

SpecFlow+ Runner uses the following Visual Studio general settings. You can find details about these settings on MSDN.

- `ResultsDirectory`: The directory where test results are placed.
- `TargetFrameworkVersion`: Default target framework version (to test using the .NET 3.5 framework, set the `TargetFrameworkVersion` to `Framework40` and use the SpecFlow+ Runner profile to specify .NET 3.5)
- `TargetPlatform`: Default processor architecture (can be overridden in the SpecFlow+ Runner profile)

Sample run settings file with general settings:

```
<RunSettings>
  <RunConfiguration>
    <ResultsDirectory>.\TestResults</ResultsDirectory>
    <TargetPlatform>x86</TargetPlatform>
    <TargetFrameworkVersion>Framework40</TargetFrameworkVersion>
  </RunConfiguration>
  ...
</RunSettings>
```

7.4.2 SpecFlow+ Runner settings

SpecFlow+ Runner settings can be specified in the `<SpecRun>` element of your `.runsettings` file. See the “Execution defaults” section for information on the default settings.

Available options:

- `Profile`: Specifies the SpecFlow+ Runner *test profile* to use.
- `ReportFile`: Specifies the name of the report file.
- `GenerateSpecRunTrait`: If set to `true`, all SpecFlow+ Runner tests are marked with the `SpecRun` trait. This can be useful for distinguishing SpecFlow+ Runner tests from unit tests in the Test Explorer window.
- `GenerateFeatureTrait`: If set to `true`, all SpecFlow+ Runner tests are marked with traits using the feature title. The “Group by Class” view of the Test Explorer window can also be used to group tests by feature.

Sample `.runsettings` file with SpecFlow+ Runner settings:

```
<RunSettings>
  <RunConfiguration>
    ...
```

(continues on next page)

(continued from previous page)

```
  </RunConfiguration>

  <!-- Configurations for SpecFlow+ Runner -->
  <SpecRun>
    <Profile>MyProfile.srprofile</Profile>
    <ReportFile>CustomReport.html</ReportFile>
    <GenerateSpecRunTrait>false</GenerateSpecRunTrait>
    <GenerateFeatureTrait>false</GenerateFeatureTrait>
  </SpecRun>
</RunSettings>
```

CHAPTER 8

Setting Up the SpecFlow+ Server

The SpecFlow+ Runner server collects execution statistics for your tests at a central location. You can use this data to improve the efficiency of your test execution by using the adaptive test scheduling option. When using this option, tests are executed in an order based on the previous execution results, i.e. failing tests are executed first, and stable tests are executed last.

8.1 Prerequisites

Before you can set up the server, **you need to install the SpecFlow+ Runner NuGet packages** that contain the server components. Information on installing the NuGet packages can be found here.

> ***Note:*** *SpecFlow+ Server is no longer included in SpecRun Nuget Packages 3.2 and above, and not supported on .NET Core.*

8.2 Installing the Server

To install the SpecFlow+ Runner server:

1. Create a new SQL database instance used to store the execution statistics.
2. Locate the "server" directory in your solution's `\packages\SpecRun.Runner.x.y.z\tools` directory (created when you install the NuGet package). Copy the contents of the "server" directory to your server.
3. Enter your database connection string in the `<Connection Strings>` element of the SpecRun.Server.exe.config file. **Note:** The `name` of the connection is hard-coded, and must be either `ReadModel` or `Database`.
4. Initialise the database using `SpecRun.Server.exe initdatabase` from the command line.
5. Start the service using `SpecRun.Server.exe start` from the command line. **Note:** You may need to start the service as a user with elevated privileges, e.g. start the command line as administrator.

6. The connection to the server takes place via port 6365; ensure that your firewall on the server allows connections via this port.
7. Enter the server's URL in the `.srprofile` file in your Visual Studio project (`<Server ServerURL ="http://MyServer:6365" publishResults="true">`).
8. Rebuild your solution and run your tests. You can verify that you have set up everything correctly by checking if records have been added to the database on the server.

8.3 Adaptive Test Execution Order

You can use these statistics to execute failing and new tests first. To do so, set the `testSchedulingMode` attribute in the `<Execution>` element of your `.srprofile` file to `Adaptive`. SpecRun then uses the statistics in the pass/fail history to determine which tests to run first. Previously failing tests and new tests are executed before successful and stable tests.

CHAPTER 9

SpecFlow+ Runner Server Data

The SpecFlow+ server can be used to store execution statistics for your tests. This data forms the basis for executing tests using the "Adaptive" `testSchedulingMode` in your *profile*. Information on setting up the server can be found here.

9.1 Adaptive Tests Scheduling

SpecFlow+ uses the test history to determine the order that tests are executed when using the "Adaptive" `testSchedulingMode`. Previously failing tests and new tests are executed before successful and stable tests.

Use the "Adaptive" test scheduling mode to prioritise tests failing tests, particularly in combination with the `stopAfterFailures` setting.

9.2 Data in the Database

The execution statistics are stored in an SQL database. The TestItems store data on the previous test results. This is in a format similar to the following:

```
<d:TestHistoryRaw>[{"Result":100,"ExecutionTime":"00:00:05.3810000"},
{"Result":100,"ExecutionTime":"00:00:05.5930000"},{"Result":100,"ExecutionTime":
↪"00:00:04.5470000"},
{"Result":100,"ExecutionTime":"00:00:05.7500000”},{"Result":100,"ExecutionTime":
↪"00:00:08.9010000]</d:TestHistoryRaw>
```

The test results per execution are stored separately (comma-separated). For each test execution, the result of the test and the execution time are stored.

The following result values are used:

- Unknown = 0
- Succeeded = 100

- Ignored = 200
- Pending = 300
- RandomlyFailed = 390
- Failed = 400

The SQL database also contains a table called Events, that also contains information on the time a test was executed, the machine, the configuration etc.

9.3 Accessing the Database via HTTP

The execution statistics in the database are also partially accessible via HTTP using the following URLs:`<Server>:<Port>/read/TestItems` `<Server>:<Port>/read/Projects`

You can append queries to the URL, such as `[...]/TestItems?$filter=(ProjectId eq guid'745C3D6E-FFC9-4ED9-AE7D-F6FA2102D4AC') and (indexof(Path, ,cart‘) ge 0)`

By default, the information is displayed as an RSS field, but if you display the source code, you can view the data itself.

CHAPTER 10

Build Servers

You can set up SpecFlow+ Runner to execute your tests whenever a build is created on your build server.

Note: When running tests on a build server with SpecRun.exe (legacy), use the `buildserverrun` *command line option*.

For general information on configuring builds, refer to the vendor-specific documentation. The following sections only cover configuring the build to execute your SpecFlow+ tests, not the full configuration process:

10.1 SpecFlow+ and TFS/VSTS

By default, TFS looks for a profile named TFS.srprofile to execute your tests; if none is found, TFS uses Default.srprofile instead. If you have changed the name of your profile, you need to enter the name of your profile in your runner settings (`<Profile>` element) and enter the path to your `.runsettings` file in the **Run Settings File** field.

10.1.1 SpecFlow+ LivingDoc

If you are using TFS/VSTS to build your projects, you may want to includes a step to build your living documentation|Generating Documentation using SpecFlow+ LivingDoc.

10.1.2 TFS 2015+/VSTS

Note: General information on running tests with your builds in TFS can be found here.

To configure your build process to execute tests using SpecFlow+ Runner:

1. Open TFS and switch to your build definition.
2. Click on **Add build step** to add a new step to the build definition. Click on **Test** in the list of categories and click on **Add** next to "Visual Studio Test".
3. Configure the build step as follows:

- Enter the path to your **Test Assembly** (the DLL containing your compiled specifications project with the test bindings).
- If you are using both MSTest and SpecFlow+ Runner, enter the path to NuGet package folder in the **Path to Custom Test Adapters** field. If you do not enter the path here, only your MSTest tests will be executed as TFS cannot find the SpecFlow+Runner test adapter.
- If you have renamed your .srprofile file (i.e. the name is not `TFS.srprofile` or `Default.srprofile`), ensure the profile is specified in your .runsettings file and enter the name of the .runsettings file in the **Run Settings File** field.

4. Save your changes.
5. You may also want to add an additional build step to generate living documentation from your feature files, see Generating Documentation.

Once the test run is complete, links to the report file and logs generated by SpecFlow+ Runner are available from the test run as attachments.

10.1.3 TFS and XAML

To configure your build process in TFS with XAML:

1. Define your build definition as described here
2. In order for the tests to execute, the binaries (DLLs in the various SpecFlow packages) need to be available to the build agent. Either:
   - Enable NuGet restore, in which cases the files are downloaded automatically
   - Check in the DLLs in the corresponding SpecFlow, SpecRun and SpecFlow+ packages. These DLLs are located in the corresponding sub-folders of your solution's packages directory.
3. Enter the name of your .srprofile file in the **Run Settings File** field if the name is not `TFS.srprofile` or `Default.srprofile` (see above).

Known Issues

- The build agents cache the test adapters, which means that the last test adapter to be used is used for each build. You can thus only use a single SpecFlow+ Runner version with the same build agent, as the cached version is always used. If you want to use a different version of SpecFlow+ Runner for different builds, you need to define separate build agents.
- Upgrading SpecFlow to a newer version requires a restart to purge the cache.

10.2 SpecFlow+ and TeamCity

Information on on configuring build steps in TeamCity can be found here. Select `VSTest` as your test engine type and configure your build step as described in the TeamCity documentation.

You may also want to install the custom logger (the link is on the documentation page).

10.3 SpecFlow+ and Jenkins

Note: General information on installing Jenkins can be found here.

The following steps assume that you are using the Git plugin to handle your source files. You will also need to install the MS Build plugin. You probably also want to install the HTML Publisher plugin to handle reports.

If Visual Studio is not installed on the build server, you will also need to install nuget.exe and the Agents for Visual Studio.

To configure your build process in Jenkins to execute tests using SpecFlow+ Runner:

1. Start the Jenkins web interface.
2. Click on **New Item** to add a new project if you have not already set up a project:
3. Enter a name for your project.
4. Choose "Freestyle project" from the list.
5. On the Jenkins main screen, select **Manage Jenkins** and then **Global Tool Configuration**.
6. Add a configuration for MSBuild. Give it a meaningful **Name** (e.g. "MSBuild") and enter the **Path to MSBuild**.
7. Add a configuration for VSTest as well, including the **Path to VSTest**.
8. Select **Configure** From the project's main page.
9. Enable **Git** under **Source Code Management**.
10. Enter your **Repository URL**, e.g. "https://github.com/Me/MyProject.git". You can also enter your login credentials.
11. Specify which branch to build under **Branches to build**, e.g. "*/master".
12. Enable **Delete workspace before build starts** under **Build Environment**, otherwise your test results may be incorrect due to the existing TRX files.
13. Click on the **Advanced** button.
14. Click on **Add** next to **Patterns for files to be deleted**: Exclude the following:.git/**, TestResults, TestResult/*.html
15. Click on **Add build step** under **Build** and select **Execute Windows batch command**.
16. Add the following **Command**:"C:\PATH\nuget.exe" restore "C:\PATH\MySpecs.sln"Replace "PATH" with the full path to nuget.exe and your solution. This ensures that your NuGet packages are restored each build.
17. Select the MSBuild configuration you defined in the **MSBuild Version** field under **Build a Visual Studio project or solution using MSBuild**.
18. Enter the path to your solution in the **MSBuild Build File** field.
19. Configure your unit tests to run with VSTest.console:

- Select the configuration you defined earlier in the **VStest Version** field.
- **Test Files**: Enter the path to the assemblies to be tested here.
- **Specify a logger for test results**: Select trx.
- **Command Line Arguments**: Enter `/TestAdapterPath:"C:\PATH\packages"` (replace "C:\PATH\" with the path to your solution's directory).

10.4 SpecFlow+ Execution Reports (optional)

The SpecFlow+ execution reports are stored in the TestResults directory of your project. To be able to easily access these reports from the project overview in Jenkins:

1. Under **Post-build Actions**, click on **Add post-build action** and select "Publish HTML reports".
2. Enter the path to your project's TestResults directory under **HTML dir to archive**.
3. Set the **Index Pages** to "ProjectName_ProfileName*.html" (replace "ProjectName and ProfileName with the name of your project and profile respectively) to reflect the SpecFlow+ report naming convention.

The results of your tests are also visualized by the Jenkins interface, e. g. in the build history and graphs. This information is taken from the TRX file generated by MSTest, which needs to be published as well:

1. Under **Post-build Actions**, click on **Add post-build action** and select "Publish MSTest test result report".
2. Enter "**/TestResults/*.trx" in the **Test report TRX file** field under **Publish MSTest test result report**.

10.5 SpecFlow+ and AppVeyor

Note: General information on AppVeyor can be found here.

To configure your AppVeyor build process to execute your test using SpecFlow+ Runner:

1. Create a new `appveyor.yml` file or open your existing file.
2. Add the following `before_build` section to restore the NuGet packages:

Replace `MyProject.sln` with the full path to your solution, relative to the location of the appveyor.yml file. 3. Add the following `test_script` section to execute the tests and generate the test output for AppVeyor:

Replace `AssemblyPath` with the path to your assembly and replace `MyAssembly.dll` with the name of your assembly. Replace `PackagesPath` with the path to the packages folder of your solution. 4. Save `appveyor.yml` and use the file to build your application with AppVeyor.

A sample project is available here.

CHAPTER 11

Migration to SpecFlow+ Runner

11.1 Installing SpecFlow+ Runner

SpecFlow+ Runner works with any assertions API, which means that switching from another test runner to SpecFlow+ Runner is easy. To switch, you will need to:

1. Install the `SpecRun.SpecFlow` NuGet package.
2. Build your project

Once you have completed these steps, you can run your tests with SpecFlow+Runner. More detailed information on the steps necessary for different unit test providers are provided below.

11.2 Detailed steps

11.2.1 MSTest

1. Remove the `SpecFlow.MsTest` NuGet package from your project
2. Add the `SpecRun.SpecFlow` NuGet package to your project
3. Build your project

11.3 NUnit

1. Remove the `SpecFlow.NUnit` NuGet package from your project
2. Add the `SpecRun.SpecFlow` NuGet package to your project
3. Build your project

11.4 xUnit

1. Remove the `SpecFlow.xUnit` NuGet package from your project
2. Add the `SpecRun.SpecFlow` NuGet package to your project.
3. Build your project

CHAPTER 12

Profiles

SpecFlow+ Runner profiles (`.srprofile` file extension) are XML files that determine how SpecFlow+ Runner executes your tests. This includes the behavior when tests fail (e.g. repeat the test, abort after X failed tests), defining various target environments for your tests (e.g. different web browsers or x64/x86), enabling multi-threading, configuring folder and file paths and applying transformations to your default configuration file in order to change the transform the configuration file for different target environments.

12.1 Using Profiles

The name of the profile file used by your project is defined using the `<Profile>` tag in your `.runsettings` file. By default, SpecFlow+ Runner expects a file called `Default.srprofile`.

Note: If you use a .runsettings file to specify your profile and are executing tests in Visual Studio, make sure you have set the settings file correctly in Visual Studio. Select **Test](Select Settings File** from the menu in Visual Studio and select your .runsettings file to do so. If your .runsettings file is not referenced correctly, your tests will run using Default.srprofile, and the results may not be the ones you expect.

If you need to regularly switch between profiles from the command line, you can do so by adding a .runsettings file for each profile, and specifying the profile as a parameter from the command line.

Examples:

```
vstest.console.exe MyTestAssembly.dll /Settings:MySettings.runsettings
```

```
dotnet test -s MySettings.runsettings
```

12.2 Default Profile

This section only affects SpecFlow+ Runner until version 3.0.284. For default values used in later SpecFlow+ Runner versions, see the section below.

The default profile is relatively basic, and includes your project name, ID and various default settings. It also includes a commented out section that you can use to transform the database connection string in your configuration file in order to access a different database instance.

You either need to add this file to your project manually, or it is added manually, depending on the version of SpecFlow+ you are using:

12.2.1 SpecFlow+ 3

When using a version of SpecFlow+ Runner higher than 3.0.284, you do not have to manually add a Default.srprofile to the project. This affects all project formats and target framework versions.

Until SpecFlow+ Runner version 3.0.284, you need to manually add a profile to your project. To do so:

1. Right-click your project and select **Add](New Item**
2. Browse to **Installed](Visual C# Items](SpecFlow**.
3. Select **SpecFlow+ Runner Profile (.srprofile)** from the list.
4. Change the name of the new item to `Default.srprofile`.
5. Click on **Add**.

12.2.2 Earlier versions of SpecFlow+

Prior to SpecFlow 3, a `Default.srprofile` is automatically added to your Visual Studio project when you add the NuGet package to your solution.

12.3 Default values used by SpecFlow+ Runner after version 3.0.284

The following behaviors are used by default if no Default.srprofile could be found:

- Search for tests in the base folder (i.e. `bin/Debug` or `bin/Debug/<Framework>`) when using `SpecRun.exe` for test execution
- `Execution` configuration element:
  - `testThreadCount` is `1`
  - `stopAfterFailures` is `3`
  - `testSchedulingMode` is `Sequential`

If you intend to use other values, you have to add a `.srprofile` file to the project.

12.4 Adding a Profile to Your Project (.NET Core)

The following section affects only SpecFlow+ Runner versions until 3.0.284.

When working with .NET Core projects, the default profile (Default.srprofile) is not automatically added to new SpecFlow+ projects. You need to add the file manually:

1. Locate `Default.srprofile` in the `content` folder of the specrun package you added to your project.
2. Add the file to your project.

3. Make any changes you require (see below).

12.5 SpecFlow+ Runner Profile Elements and Attributes

The `<TestProfile>` element is a container for the remaining elements.

The following XML elements and attributes are available:

-
-
-
-
-
-
-
-
-
-
-

You can also use a number of Placeholders in your profile.

CHAPTER 13

Settings

The `<Settings>` element defines general settings for your project. The following attributes are available:

13.1 Example

```
<Settings
    name="MySettings"
    projectName="MyProject"
    projectId="{7B359152-D36D-4D5A-BFDB-ED78B094D619}"
    baseFolder="C:\MyBaseFolder"
    outputFolder="..\OutputFolder"
    reportTemplate="MyTemplate.cshtml"/>
```

CHAPTER 14

Execution

The `<Execution>` element defines how your tests are executed. Use these settings to run tests in parallel by defining multiple threads, determine how often and when tests are retried, and when to abort the test run due to failing tests passing a specific threshold.

You can also schedule tests to run based on your test history stored in the *SpecFlow+ Runner server*. If you use adaptive mode, previously failing tests are prioritized over tests that have passed.

14.1 Attributes

The following **optional** attributes are available:

14.1.1 `stopAfterFailures`

- The number of failed tests after which the execution of all tests is stopped. Set this to 0 (zero) to execute all tests, irrespective of the number of failures. If **not specified**, this defaults to 10.

  Note: *The value in the default `.srprofile` shipped with the NuGet package is 3.*

14.1.2 `testSchedulingMode`

- Determines the order in which SpecRun executes your tests:
  - `Sequential` : Tests are always executed in the same order **(default)**. The order in which tests are executed is predictable and always the same, but cannot be changed manually.
  - `Random` : The order of tests is randomised. This can be useful to detect flickering scenarios that may pass/fail depending on the order in which the other tests area executed.
  - `Adaptive` : SpecRun uses the statistics available in the pass/fail history to determine which tests to run first. Previously failing tests and new tests are executed before successful and stable tests.

***Note:** If you set the `testSchedulingMode` to `Adaptive` when no previous test execution results are available (e.g. no server has been configured, or the server is unreachable), the tests are executed using the `Random` option instead.*

14.1.3 `testThreadCount`

- The number of tests threads used to execute tests **default value is 1**. While there is no maximum number of threads, setting this value too high can reduce performance as the individual threads compete for processing power.
- For `Process` test thread isolation (`testThreadIsolation` setting), it recommended to set the number of threads to one less that the number of cores available. You can generally use higher numbers for isolation using `AppDomain` and `SharedAppDomains`. See *Parallel Exeuction* documentation for more info on `testThreadIsolation`.

  ***Note:** Just increasing the number of threads on its own may cause issues. E.g. if the threads access the same database. It thus makes sense to combine this with a relocation and transformation of the .config file using the `{TestThreadId}` placeholder in the name of the new file and database instances.*

  ***Note:** Before and After hooks (e.g. BeforeTestRun, AfterTestRun etc.) are executed per thread.*

14.1.4 `retryFor`

- Determines which tests should be retried based on their status:
  - `None`: No tests are retried.
  - `Failing`: Failing test are retried **(default)**.
  - `All`: All tests are retried, including those that pass.

  ***Note:** Tests are retried immediately after returning the corresponding status and before any subsequent tests are executed.*

14.1.5 `retryCount`

- The number of times tests that their execution status matches the value defined in `<retryFor>` are retried. **Default value is 2**.

14.1.6 `apartmentState`

- Sets the apartment state used to execute the tests:
  - `STA` : Single Threaded Apartment. Use this if your application is not thread-safe.
  - `MTA` : Multi-Threaded Apartment.
  - `Unknown` : The ApartmentState is not set ; tests run in same thread as SpecFlow+. **(default)**.

Note: *Setting the* `apartmentState` *property does not set the test thread apartment state on Linux and OSX. ApartmentStates are not supported on non-Windows systems because COM interop is only available on Windows. Please refer to Microsoft's documentation for more information*

14.2 Example

The following example retries only failed tests regardless of their outcome, using all threads for execution. The order of tests is random in a multi threaded execution and a failed test is re-executed twice:

```
<Execution retryFor="Failing" stopAfterFailures="0" testThreadCount="Max"␣
↪testSchedulingMode="Random" retryCount="2" apartmentState="MTA"/>
```

CHAPTER 15

Environment

The `<Environment>` element defines your target platform environment.

15.1 Attributes

The following **optional** attributes are available:

15.1.1 `framework`

- The .NET framework in use. SpecFlow+ determines which CLR (Command Language Runtime) version to use to execute the tests based on these setting:
  - `Net5.0` : .NET 5.0 - CLR net50
  - `Net461` : .NET 4.6.1 - CLR 4.0 **(Default)**
  - `Netcoreapp3.1` : .NET Core 3.1 - CLR netcore31
  - `Netcoreapp2.1` : .NET Core 2.1 - CLR netcore21

15.1.2 `platform`

- The target platform architecture: x86, x64 or Default (the setting under **Test | Test Settings | Default Processor Architecture** in Visual Studio).

15.1.3 `testThreadIsolation`

- Determines the level of thread isolation:
  - `Process` : Default for .NET Core and .NET
  - `AppDomain` : Default for .NET Framework, available only in .NET Framework
  - `SharedAppDomain`

15.1.4 `apartmentState`

- Sets the apartment state used to execute the tests:
  - `STA`: Single Threaded Apartment. Use this if your application is not thread-safe.
  - `MTA`: Multi-Threaded Apartment.
  - `Unknown`: The ApartmentState is not set; tests run in same thread as SpecFlow+. **(default)**

  Note: *Setting the* `apartmentState` *property does not set the test thread apartment state on Linux and OSX. ApartmentStates are not supported on non-Windows systems because COM interop is only available on Windows. Please refer to Microsoft's documentation for more information.*

15.2 Example

Multi-Threaded Apartment with a Process isolation:

```xml
<Environment apartmentState="MTA"  testThreadIsolation="Process"/>
```

CHAPTER 16

Report

16.1 Template Element

Use the `<Template>` element to specify the templates used to generate reports and the behaviour of the output file. You can generate multiple reports by defining multiple `<Template>` elements, one for each report.

The following attributes are available:

Note: The report template specified in the <Settings> element (`reportTemplate`) is used in addition to the templates specified in the `<Report>` element.

16.2 Examples

To disable the report generation:

```xml
<TestProfile xmlns="http://www.specrun.com/schemas/2011/09/TestProfile">
    <Settings name="Disable Reports" projectName="SpecRun Test Project" />
    <Report disable="true"/>
    <TestAssemblyPaths>
        <TestAssemblyPath>SpecRun.TestProject.dll</TestAssemblyPath>
    </TestAssemblyPaths>
</TestProfile>
```

To output 2 additional reports and copy the reports to the base folder:

```xml
<TestProfile xmlns="http://www.specrun.com/schemas/2011/09/TestProfile">
    <Settings name="Multiple Reports" projectName="SpecRun Test Project" />
    <Report copyAlsoToBaseFolder="true">
        <Template name="CustomReportTemplate_1.cshtml" outputName="Report1.html" />
        <Template name="CustomReportTemplate_2.cshtml" outputName="Report2.html" />
    </Report>
    <TestAssemblyPaths>
```

(continues on next page)

(continued from previous page)

```
        <TestAssemblyPath>SpecRun.TestProject.dll</TestAssemblyPath>
    </TestAssemblyPaths>
</TestProfile>
```

CHAPTER 17

Filter

The `<Filter>` element allows you to define filters that are applied to your tests, allowing you to determine which tests to execute. Filters can also be defined in `<Target>` elements, in which case the filter only applies to that target. Filters defined outside of a `<Target>` element are applied globally. The **Test Explorer** window in Visual Studio only lists those tests that meet your filter criteria (after rebuilding your project). Note that global filters also apply when running tests by right-clicking on a feature file and selecting **Run SpecFlow Scenarios**.

Filters can be applied to tests based on tags (including regular expressions), or the scenario or feature name.

Note: Tags and filters are **case-sensitive**.

The following filter types can be defined:

You can combine filters using logical operators. The following operators are supported:

17.1 Examples

Filter

`<Filter>@MyTag &amp; @YourTag</Filter>` executes all tests with both the `@MyTag` and `@YourTag` tags.`<Filter>!tagmatch:Tag[1-9]</Filter>` executes all tests that are not tagged with any of `@Tag1` to `@Tag9`.`<Filter>@MyTag | tagmatch:tag[1-9]</Filter>` executes all tests with either the `@MyTag` tag or tags `@Tag1` to `@Tag9`.

Target with Filter

```
<Targets>
   <Target name="IE_11">
      <Filter>@MyTag</Filter>
   </Target>
    <Target name="FireFox">
      <Filter>@MyTag</Filter>
   </Target>
    <Target name="Chrome">
      <Filter>@MyTag</Filter>
```

(continues on next page)

(continued from previous page)

```
   </Target>
</Targets>
```

CHAPTER 18

Targets

The `<Targets>` element is a container element for `<Target>` elements. Each `<Target>` element defines a test target. Tests are executed for each target, and you can define different target environments for your test. For example, you can define a target for x64 and x86 environments. You can also apply filters to each target, e.g. to only execute tests tagged with `@cloud` in browser environments. Tests that are executed for multiple targets are listed multiple times in the Test Explorer window, once for each target. See the SeleniumWebTest sample project for an example of using different targets.

The following attributes are available for the `<Target>` element:

18.1 Example

```
<Targets>
   <Target name="32bit">
      <Filter>@32bit</Filter>
      <Environment platform="x86" apartmentState="STA" testThreadIsolation="Process"/>
   </Target>
</Targets>
```

CHAPTER 19

DeploymentTransformation

This element is used to define transformations, which you can also apply to your configuration file (`app.config`) when using the Full Framework in your project. **Note that you cannot transform the configuration file in .NET Core projects, as .NET Core does not support app.config.**

Transformations can be nested within a `<Target>` element, allowing you to define different configuration settings per target, e.g. for different platforms (x64/x86) or for different web browsers. You can also use `{Target}` as a placeholder for this value, in which case the placeholder is replaced with the target name.

Note: When running tests in in multiple threads, you need to ensure that each thread is accessing a different file, or you will encounter conflicts. Use `RelocateConfigurationFile` in conjunction with the `{TestThreadID}` placeholder to achieve this.

The following elements and attributes are available:

Each step can contain the following elements:

19.1 Example

```
<DeploymentTransformation>
   <Steps>
      <ConfigFileTransformation configFile="App.config">
         <Transformation>
            <![CDATA[<?xml version="1.0" encoding="utf-8"?>
               <configuration xmlns:xdt="http://schemas.microsoft.com/XML-Document-
↪Transform">
                  <appSettings>
                     <add key="foo" value="bar" xdt:Locator="Match(key)"␣
↪xdt:Transform="Insert"/>
                  </appSettings>
               </configuration>
            ]]>
         </Transformation>
      </ConfigFileTransformation>
```

(continues on next page)

(continued from previous page)

```
   </Steps>
</DeploymentTransformation>
```

CHAPTER 20

TestThreads

The `<TestThreads>` element is a container for `<TestThread>` elements. The following attributes are available for `<TestThread>` elements:

You can use the `{TestThreadId}` as a placeholder to reference the thread ID, e.g. to transform the name of the database instance you are accessing based on the thread ID, ensuring that each thread accesses a separate instance of the database. This prevents the threads from conflicting with one another when accessing the database, as thread 0 may manipulate data that is required by thread 1 otherwise.

20.1 Example

```
<TestThreads>
    <TestThread id="0">
        <TestAffinity>testpath:Target:FireFox</TestAffinity>
    </TestThread>
</TestThreads>
```

CHAPTER 21

VSTest

The `<VSTest>` element determines the behavior when using SpecFlow+ Runner in conjunction with VS Test.

21.1 Attributes

21.1.1 `testRetryResults`

- By default, a test that initially fails is treated as having failed, even if the retries are successful. This can result in test execution results that show the same test as both failing (the initial test) and passing (the subsequent retries). This is particularly common when testing web applications if the initial test times out. In some cases, you may want to determine that a test should be treated as passing, even if one or other of the tests fails, allowing you to discount tests that time out.
  - `Separate` : When retrying tests, separate test results are output for each execution of the test: the initial (failing) execution and all retries. **(Default)**
  - `Unified` : A single unified result is output for the test. If the percentage of successful tests (the initial test + all retries) exceeds the threshold (passRate), the test is treated as passing.

21.1.2 `passRateRelative`

- Determines the percentage threshold (default = 100%) used to determine when a test should be treated as passing when using the `Unified` option. For example, if the `passRateRelative` is set to 50%, the test is only counted as passing if at least half the tests pass.

21.1.3 `passRateAbsolute`

- Determines the absolute number of successful retries to determine when a test should be treated as passing when using the `Unified` option.

Note: *You can either specify* `passRateRelative` *or* `passRateAbsolute`*. If you specify both, you will get an error.*

21.2 Example

```
<VSTest testRetryResults="Unified" passRateAbsolute="1"/>
```

CHAPTER 22

TestAssemblyPaths

The <TestAssemblyPaths> element is a container for <TestAssemblyPath> elements, which define your assembly paths when executing tests from the command line. Paths in <TestAssemblyPath> elements are relative to the base folder.

Make sure that you specify the correct name(s) of all your assembly file(s) in their own <TestAssemblyPath> element, otherwise no tests will be found!

22.1 Example

```
<TestAssemblyPaths>
    <TestAssemblyPath>SpecRun.TestProject.dll</TestAssemblyPath>
</TestAssemblyPaths>
```

CHAPTER 23

Server

You can publish the results of your tests to a SpecFlow+ Runner server. Details on setting up the server can be found *here*.

The following attributes are available:

23.1 Example

```
<Server serverUrl="http://specrun.server:6365" publishResults="true" />
```

CHAPTER 24

Placeholders

There are placeholders available for the elements defined in your profile. You can use these placeholders in your configuration file. This allows you to use the same configuration file for various target environments, e.g. you could use the `{TestThreadId}` placeholder in the name of your database instance to ensure that each thread accesses a different instance of your database (Instance0, Instance1 etc.) .

The following placeholders are available:

- `{TestThreadId}`: The ID of the current thread (integer)
- `{Target}`: The name of the current target (string)
- `{UniqueId}`: The unique ID entered in the profile
- `{BaseFolder}`: The base folder
- `{OutputFolder}`: The output folder for test results and reports
- All environment variables (specified using the standard format, i.e. enclosed in percentage signs ('%'))

Placeholders can be used in the following elements:

- ConfigFileTransformation/Transformation
- CopyFile/target
- CopyFile/source
- CopyFile/targetFolder
- CopyFolder/source
- CopyFolder/target
- CustomDeploystep/type (in the assembly path)
- IISExpress/Port
- IISExpress/webAppFolder
- IISExpress/iisExpressPath
- RelocateConfigurationFile/target

- Relocate/targetFolder

CHAPTER 25

Parallel Execution Features

To start a parallel test run, you simply need to change the testThreadCount property in your srProfile to a number higher than 1. How your tests are executed then depends on the testThreadIsolation property.

The three supported modes are:

- AppDomain
- Process
- SharedAppDomain

25.1 AppDomain

This is the default mode. Each test thread is executed in a separate AppDomain. These AppDomains are created at the beginning of the test run, and are reused for the rest of the test run.

25.1.1 Pros

Executed tests are isolated by the AppDomain border, so you do not have problems with static data.

25.1.2 Cons

Limited when you have shared data on a process level (e.g. SQLite in-memory dbs)

25.2 Process

This mode has been supported since version 1.2. A separate executor process is created for each test thread and is used to execute the tests. This is necessary if your application contains entities that exists once per process, e.g. SQLite's in-memory database. These processes are started at the beginning of the test run, and are reused for the rest of the test run.

This mode is also used if you run your tests using the .NET 2.0 framework or for a different processor architecture.

To keep your test run short, I would recommend settings testThreadCount to (CPU Cores – 1). The remaining core is then kept free for the actual test runner process to manage the other executor processes.

25.2.1 Pros

Completely process-based separation of executed tests

25.2.2 Cons

Slower due to the additional cost of starting the test execution processes and inter-process communication

25.3 SharedAppDomain

This new mode takes advantage of the new parallelization support in SpecFlow 2.0, and executes all tests in the same AppDomain. This makes it very fast, but the trade-off is that you lose the isolation between the currently executed tests. However if you have tests that do not require isolation, this is the fastest way to execute them.

When using this mode, you can set testThreadCount to really high numbers and still have fast test runs.

25.3.1 Pros

Very fast

25.3.2 Cons

No isolation between currently executed tests

CHAPTER 26

Command Line Usage

You can run SpecFlow+ tests from the command line, either using `SpecRun.exe` (Full Framework) or `dotnet test` or `vstest.console.exe` (.NET Core).

26.1 Syntax Changes in V3

Note that **the syntax for command line options was changed with SpecFlow+ Runner 3**. Command line options now need to be prefixed with `--` (double dash) rather than `/`.

26.2 Running .NET Core projects from the Command Line

To run your .NET Core tests from the command line, use either `dotnet test` or `vstest.console.exe`:

- `dotnet test`: Specify the path to your solution or project, or run the command from that folder. See the documentation here.
- `vstest.console.exe`: Specify the path to your test assembly. See the documentation here.

26.3 Running Full Framework projects from the Command Line

SpecRun.exe is only compatible with projects developed using the Full Framework. You cannot use SpecRun.exe in conjunction with .NET Core projects.

You can run your SpecFlow tests via SpecFlow+ Runner from the command line. Your specification project's directory contains a batch file, `runtests.cmd` that you can start to run your tests. You can also use `SpecRun.exe` located in the `..\packages\SpecRun.Runner.{version}\tools\` directory of your Visual Studio project to run your tests.

Start `SpecRun.exe` without any parameters to display an overview of the available commands. Use the `help` command to display information on a specific command, e.g. `SpecRun.exe help run`.

The following commands are available from the command line:

26.4 run

Use `SpecRun.exe run` to run your tests using the following parameters:

Example:`SpecRun.exe run Default.srprofile --basefolder c:\MyProjectFolder --outputfolder output --report MyReport.html`Executes your tests using the `Default.srprofile` file. The base folder is set to `C:\MyProjectFolder` while the output folder is set to `C:\MyProjectFolder\output` (the `output` folder is relative to the base folder). The generated report is output to `C:\MyProjectFolder\output\MyReport.html`.

26.5 buildserverrun

Note: This option is not required when running your tests with `dotnet test` or `vstest.console.exe` (required for .NET Core projects). This option is only required when using `SpecRun.exe` to run tests on a build server.

Use `SpecRun.exe buildserverrun` to execute your tests in build server mode using the following parameters:

26.6 register

Use `SpecRun.exe register` to register your SpecFlow+ license. You only need to register your license once per user per machine. The license is valid for all SpecFlow+ components.

26.7 unregister

Use `SpecRun.exe unregister` to unregister your SpecFlow+ license.

26.8 about

Use `SpecRun.exe about` to display information such as your version number, build date and license information (licensee, upgrade until date/expiry date).

CHAPTER 27

Test Execution Result Codes

SpecFlow+ Runner returns the following result codes, which are included in the JSON output:

CHAPTER 28

Generating a TRX file

To generate a TRX file with SpecFlow+ Runner, you will need to start your tests from the command line using `vstest.console.exe` with the `/logger:trx` option to generate a TRX file.

You have to execute it in the output folder of your test project (`bin\debug\`).

28.1 SpecFlow+Runner 1.8 and Later

Execute your tests using the following syntax:`vstest.console.exe <MyTestAssembly.dll> /logger:trx`

28.2 Before SpecFlow+Runner 1.8

You have to specify the path to the testadapter, so you execute your tests using the following syntax:`vstest.console.exe <MyTestAssembly.dll> /TestAdapterPath:<PATH_TO_TESTADAPTER.EXE> /logger:trx`

Replace `<MyTestAssembly.dll>` with the name of your test assembly.<PATH_TO_TESTADAPTER.EXE> is the path to the `TechTalk.SpecRun.VisualStudio.TestAdapter.dll` (located in the `/packages/specrun<version>/tools` directory of your project).

CHAPTER 29

Reports

SpecFlow can generates reports once your tests have finished executing that include a breakdown of the results of your tests. The default report includes a statistical overview of the status of all tests, as well as information on individual scenarios, including Gherkin test cases, statistics on the total number and percentage of successful tests, and the execution time for each step. When running tests from within Visual Studio, a link to the generated report(s) is included in the **Output** window once the tests have completed.

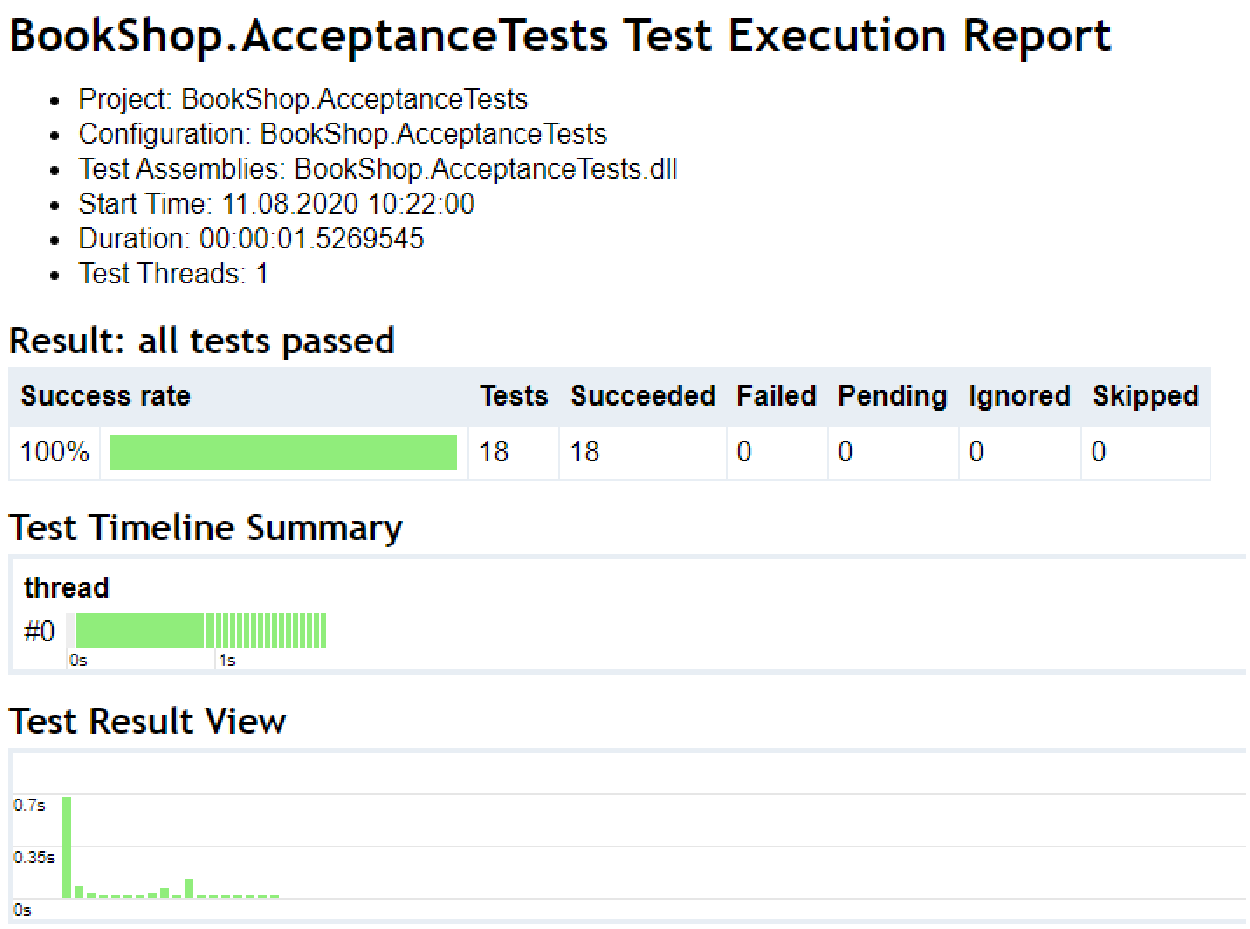
The report is output to your output folder (configured in your *profile*) or `TestResults` folder and the name of the report is generated using the `projectName` and `name` defined in your profile plus a time stamp: `<projectName>_<name>_YYYY-MM-DDTHHMMSS`

If you want to generate multiple reports from a single test run, you also need to modify your profile to include the templates and output paths for these reports. The settings for multiple reports are defined in the *<Report> element*.

29.1 Standard Report structure

29.1.1 Test Run Summary

In the Test Run summary you find information, that are about the whole test run.



BookShop.AcceptanceTests Test Execution Report

- Project: BookShop.AcceptanceTests
- Configuration: BookShop.AcceptanceTests
- Test Assemblies: BookShop.AcceptanceTests.dll
- Start Time: 11.08.2020 10:22:00
- Duration: 00:00:01.5269545
- Test Threads: 1

Result: all tests passed

Success rate		Tests	Succeeded	Failed	Pending	Ignored	Skipped
100%	🟩	18	18	0	0	0	0

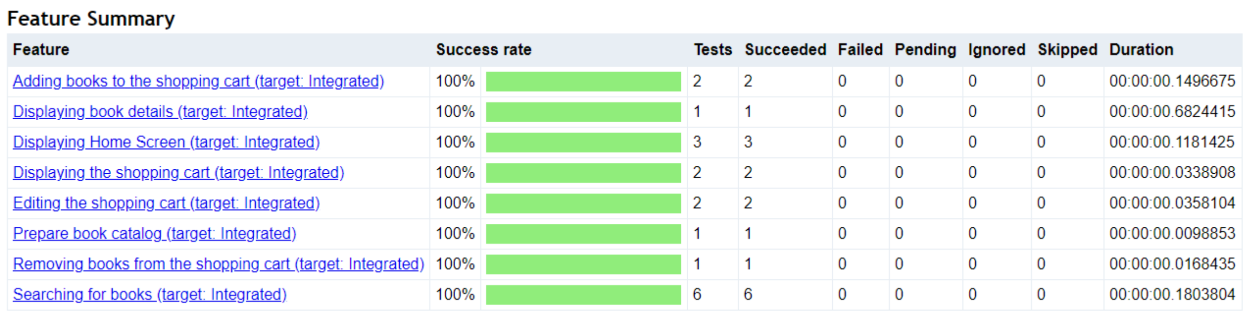
Test Timeline Summary

Test Result View

The first table displays you, how many scenarios had the different results.The Test Timeline Summary shows you, which scenario was executed on which thread.The Test Result View shows you by default the duration and result of the scenarios. You can change what you want to see and how to sort.

29.1.2 Feature Summary

The Feature Summary shows you the result of every executed feature.

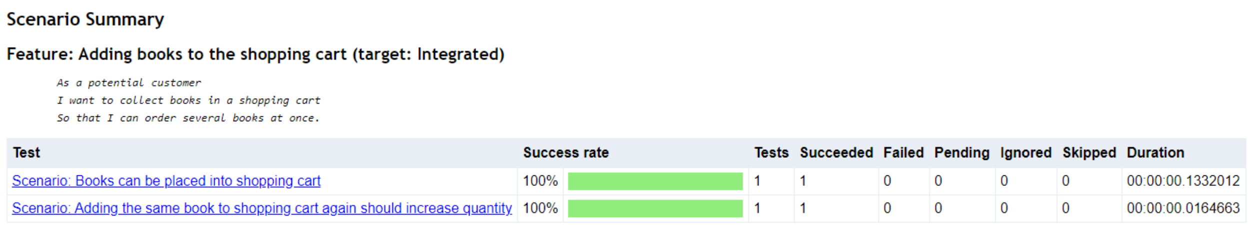


Feature Summary

Feature	Success rate		Tests	Succeeded	Failed	Pending	Ignored	Skipped	Duration
Adding books to the shopping cart (target: Integrated)	100%	🟩	2	2	0	0	0	0	00:00:00.1496675
Displaying book details (target: Integrated)	100%	🟩	1	1	0	0	0	0	00:00:00.6824415
Displaying Home Screen (target: Integrated)	100%	🟩	3	3	0	0	0	0	00:00:00.1181425
Displaying the shopping cart (target: Integrated)	100%	🟩	2	2	0	0	0	0	00:00:00.0338908
Editing the shopping cart (target: Integrated)	100%	🟩	2	2	0	0	0	0	00:00:00.0358104
Prepare book catalog (target: Integrated)	100%	🟩	1	1	0	0	0	0	00:00:00.0098853
Removing books from the shopping cart (target: Integrated)	100%	🟩	1	1	0	0	0	0	00:00:00.0168435
Searching for books (target: Integrated)	100%	🟩	6	6	0	0	0	0	00:00:00.1803804

29.1.3 Scenario Summary

As the Feature Summary, the Scenario Summary shows you the result of every executed scenario in a feature.



Scenario Summary

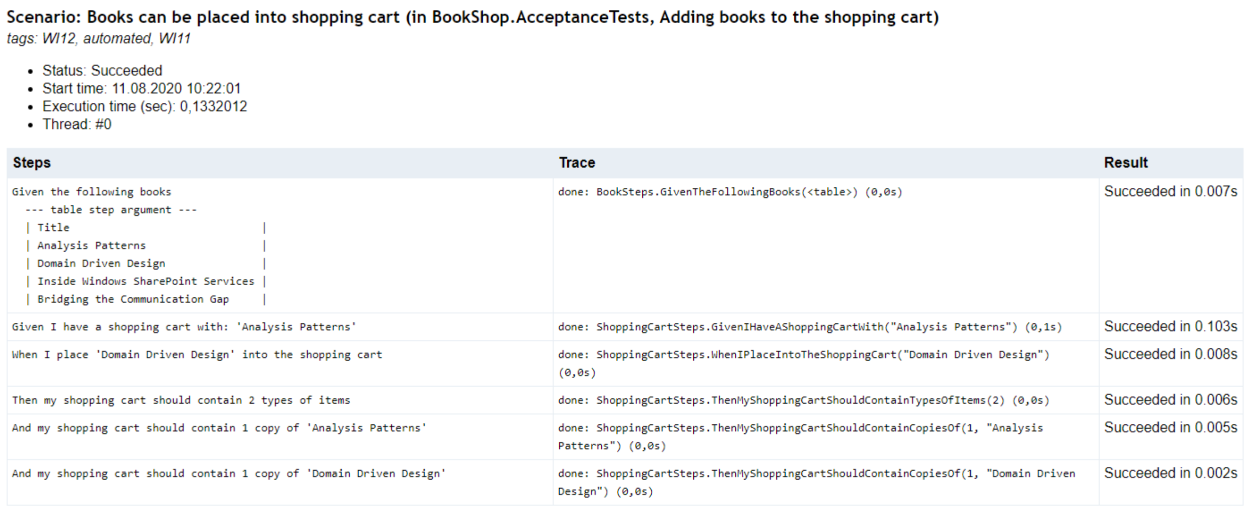
Feature: Adding books to the shopping cart (target: Integrated)

As a potential customer
I want to collect books in a shopping cart
So that I can order several books at once.

Test	Success rate	Tests	Succeeded	Failed	Pending	Ignored	Skipped	Duration
Scenario: Books can be placed into shopping cart	100% 🟩	1	1	0	0	0	0	00:00:00.1332012
Scenario: Adding the same book to shopping cart again should increase quantity	100% 🟩	1	1	0	0	0	0	00:00:00.0164663

29.1.4 Scenario Detail

In the Scenario Detail you see all the execution details for this scenario.



Scenario: Books can be placed into shopping cart (in BookShop.AcceptanceTests, Adding books to the shopping cart)
tags: WI12, automated, WI11

- Status: Succeeded
- Start time: 11.08.2020 10:22:01
- Execution time (sec): 0,1332012
- Thread: #0

Steps	Trace	Result

`Given the following books`
`--- table step argument ---`
`| Title |`
`| Analysis Patterns |`
`| Domain Driven Design |`
`| Inside Windows SharePoint Services |`
`| Bridging the Communication Gap |`
	`done: BookSteps.GivenTheFollowingBooks(<table>) (0,0s)`	Succeeded in 0.007s

`Given I have a shopping cart with: 'Analysis Patterns'`	`done: ShoppingCartSteps.GivenIHaveAShoppingCartWith("Analysis Patterns") (0,1s)`	Succeeded in 0.103s
`When I place 'Domain Driven Design' into the shopping cart`	`done: ShoppingCartSteps.WhenIPlaceIntoTheShoppingCart("Domain Driven Design") (0,0s)`	Succeeded in 0.008s
`Then my shopping cart should contain 2 types of items`	`done: ShoppingCartSteps.ThenMyShoppingCartShouldContainTypesOfItems(2) (0,0s)`	Succeeded in 0.006s
`And my shopping cart should contain 1 copy of 'Analysis Patterns'`	`done: ShoppingCartSteps.ThenMyShoppingCartShouldContainCopiesOf(1, "Analysis Patterns") (0,0s)`	Succeeded in 0.005s
`And my shopping cart should contain 1 copy of 'Domain Driven Design'`	`done: ShoppingCartSteps.ThenMyShoppingCartShouldContainCopiesOf(1, "Domain Driven Design") (0,0s)`	Succeeded in 0.002s

Each row in the table is one executed step in the scenario.

Steps: Executed step**Trace**: Output that happened during execution of this step. You can use `Console.WriteLine(string)` to output your own text here.**Result**: The result of the step and the duration of execution

29.2 Additional Requirements for Non-Windows Users

If you are running your tests on Linux or macOS, you need Mono installed in order to generate reports with SpecFlow+. Make sure you have installed Mono on the machine you are using to execute your tests (i.e. on your development machine or build server). We have tests for the reports using the latest version of Mono, but previous versions should also work.

For more information on installing Mono, please refer to the Mono documentation:

- Installation instructions for Linux.
- Installation instructions for macOS.

29.3 Report Templates

A Razor template is used to generate the reports, and the SpecFlow+ Runner NuGet package includes several report templates. The templates are located in the `\packages\[SpecRun.Runner]\templates` directory of your Visual Studio project and include the following:

- `ReportTemplate.cshtml`: The default report template, this is a standard report that outputs information on the results of the test run as HTML, and is human-readable.
- `ReportTemplate_Json.cshtml`: Outputs the results of the test run as JSON. This format is intended for post-processing purposes.
- `ReportTemplate_Xml.cshtml`: Outputs the results of the test as XML. This format is intended for post-processing purposes.

You can customize the templates to meet your needs. If you customize a template, we recommend renaming the template file accordingly. For some examples of how you can customize your reports, refer to the Customising Reports tutorial.

Note: The old HTML report template for SpecFlow+ 1.2 can be found here. There are no XML or JSON report templates for version 1.2.

29.4 Defining Your Own Template

Please review the SpecFlow+ Runner Report API Documentation to find out about the classes and properties of the reporting model that can be referred in the reporting template.

To define your own report template:

1. Create a copy of `ReportTemplate.cshtml` (in `\packages\[SpecRun.Runner]\templates`) and rename it accordingly.
2. If desired, add the .cshtml file to your Visual Studio project so that you can edit in Visual Studio. If you do this, ensure that **Copy to Output Directory** in the file's properties is set to "Copy always" to ensure that the latest version of the template is always used to generate reports.
3. Edit your `.srprofile` file and change the value of the `reportTemplate` attribute in the `Settings` element to the name of your template file. This path is relative to your base folder (default: `\bin\`).
4. Change the `name` and `projectName` attributes in the `Settings` element to reflect your project. These values are used to name the generated report.
5. Execute your tests to generate the report.

29.5 Generating a Single Report for Multiple Specification Projects

A SpecFlow+ report is generated for the `.srprofile` file used to execute the tests from the command line. You can add multiple assemblies to your `.srprofile` file, in which case the report will combine the results of the assemblies in a single report.

To generate a single report for multiple assemblies:

1. Open the profile you want to edit and locate the `<TestAssemblyPaths>` section. This should already contain the path to the assembly linked to the project the profile belongs to.
2. Add the test assemblies you want to include in the report as elements. You need to specify the paths as relative to the location of the assembly associated with the profile's project. If all your projects are located in the same root folder, the path should be `../../../<ProjectDirectory>/bin/debug/<ProjectSpecificationAssembly>.dll`.
3. Start `runtests.cmd` located in the directory of your edited profile. Your tests are executed and the report is generated.

29.6 Generating Multiple Reports in a Single Test Run

You can generate multiple reports from a single test run. To do so, you also need to modify your `.srprofile` file to reference the .cshtml templates and output paths for each reports. The settings for multiple reports are defined in the *<Report> element* of your profile. You can use these settings to determine the name of the report files and how to handle conflicts with existing files with the same name.

CHAPTER 30

Tutorial: Customizing Reports

Note: This tutorial assumes that you have already set up a Visual Studio project with SpecFlow+ Runner and have several tests that are executed. If you have not yet set up a project, refer to the Getting Started guide first.

This section also includes a couple of examples of customising reports.

30.1 Customizing Reports

You can output HTML reports after each test execution run that include information on the tests in the test run, e.g. their status (passed/failed), the number of retries, execution time etc. SpecFlow+ Runner includes a default report template (`ReportTemplate.cshtml`) that you can use as the basis for customising your own reports. This tutorial will take you through the process of customising the default report template to include sortable lists, a graphical representation of statistical data and more.

The default template (`ReportTemplate.cshtml`) is located in your user folder under `\.nuget\packages\specrun.runner\[version number]\templates` with SpecFlow 3. If you are using an older version, it is located in the `\packages\[SpecRun.Runner]\templates` directory of your Visual Studio projects. **We recommend making a copy of this template, rather than editing it directly**.

30.2 Initial Setup

The following initial steps are optional but highly recommended:

1. Create a copy of the `ReportTemplate.cshtml` file in the `\packages\[SpecRun.Runner]\templates` directory of your VS project, and give this file a new name (e.g. `MyReportTemplate.cshtml`).
2. Add the template file to your specification project in Visual Studio so you can edit it directly in Visual Studio.
3. Set the template file's **Copy to Output Directory** property to "Copy always" to ensure that the up-to-date template is available when executing your tests.
4. Add the necessary DLLs to your project as references to enable Intellisense.

30.2.1 Configuring SpecFlow to use your Template

Before you can use your new template, you need to tell SpecFlow+ Runner where to find the template. You can also determine the name of the HTML report file that is generated. To do so:

1. Open your project in Visual Studio.
2. Open your `.srprofile` file (the default name is `Default.srprofile`).
3. Edit the `Settings` *element* as required:
    1. `projectName`: The name of your project and the first portion of the name of the report.
    2. `name`: The name of your settings and the second portion of the report's name.
    3. `outputFolder`: The folder the report is output to, relative to your base folder (`baseFolder` attribute).
    4. `reportTemplate`: The location of the report template used to generate the report, relative to your base folder. Enter the name of the template you created earlier here.
4. A time stamp (date in YYYY-MM-DD format and time in HHMMSS format) is appended to the generated report. The final name of the generated report is composed as follows: `<projectName>_<name>_YYYY-MM-DDTHHMMSS`
5. Run your tests and click on the link to the generated report in the **Output** window in Visual Studio to ensure that your report template can be located and a report is generated.

Note: You can generate multiple reports from the same test run. To generate additional reports, use the `<Report>` element as described *here*.

30.3 From Template to Report

The .cshtml html template determines the output of the final HTML report. The template can include both static HTML and dynamically generated content. You therefore have access to all the formatting options HTML provides, and you can use C# to dynamically determine the contents of the report.

If you open the default template or your renamed copy of it, you will notice it contains a series of helper functions, and an HTML section (starting from `<html>`). We will be extending both the helper functions to implement additional logic (e.g. to populate statistical charts) and the HTML to include additional information (e.g. embed the chart) in the output.

30.3.1 Referencing Test Data from SpecFlow+

You can reference the properties by SpecFlow+ to perform calculations and output information on the test run you are interested in. Use the '@' character as a prefix to reference these properties, e.g. `@Model.Configuration.ProjectName` returns the name of your project.

You can see this in action for yourself. Search for "<body>" in your template to locate the start of the HTML report's body:

```
<body>
        <h1>@Model.Configuration.ProjectName Test Execution Report</h1>
```

The second line defines the top level heading (`h1` element) at the start of the report. @Model.Configuration.ProjectName accesses the name of your project while "Test Execution Report" is plain (static) text. As mentioned before, the '@' character is used to denote code sections or variables to embed in your HTML.

Let's make a minor change to the report's heading:

1. Change `@Model.Configuration.ProjectName` to `@Model.Configuration.TestProfileSettings.ReportTemplate`
2. Change "Test Execution Report" to "is my new Test Execution Report template".The content should look like this:<body><h1>@Model.Configuration.TestProfileSettings.ReportTemplate is my new Test Execution Report template</h1>
3. Run your tests and click on the link to the report to view the output. The header should now be similar to the following:

While this is a trivial example, it demonstrates how to integrate information passed by SpecFlow+ in your report and how to include it in the HTML output. The examples below include some more interesting customizations that you can use as the basis for your own templates.

30.4 Examples

The following examples are intended to get you started and give you some ideas for how to customize your own reports:

30.4.1 Sortable Lists

By default, the tables in the report are static and the columns cannot be sorted. However you can easily make your tables sortable using JavaScript. This can be particularly useful for sorting tests by execution time, for example, in order to determine which tests take a long time to execute and may therefore require optimisation.

For simplicity's sake, this tutorial uses SortTable by Stuart Langridge to create tables whose columns can be sorted by clicking in the column header. Download the source JavaScript file (sortable.js) from Stuart's website at the previous link. Obviously you can use an alternative solution or roll your own – the principle is the same.

Editing the Template

Once you have downloaded the JavaScript source, open your CSHTML template file and add a reference to the script to the start of the template:

```
@inherits TechTalk.SpecRun.Framework.Reporting.CustomTemplateBase<TestRunResult>
<!DOCTYPE html>
@using System
@using System.Linq
@using System.Globalization
@using TechTalk.SpecRun.Framework
@using TechTalk.SpecRun.Framework.Results
@using TechTalk.SpecRun.Framework.TestSuiteStructure
<script src="sorttable.js"></script>
```

Adding a Sortable Table

We are going to add a sortable table to display information on the individual steps in a feature file. You can either replace the final table in the default template (`<table class="testEvents">`), or simply add this table to the end of the report (within the `<body>` section). We will include the time taken by each step in the table so we can sort the table by step duration. This makes it easy to identify any steps that are taking a long time to execute and possibly optimise them.

Defining a sortable table using sortable.js is easy: just start your HTML table declaration with the following element:`<table class="sortable">`

The remaining definition of the table is standard HTML using `<tr>` and `<td>` elements. We will also need to define an “Index” column to ensure that we can revert the table to the default sort order at any time. The following code initialises the index to 0 (you can obviously start counting from 1 if you prefer) and defines the column headers for each of the columns for our test results:

```
int Index = 0;
<table class="sortable">
<tr>
    <th>Index</th>
    <th>Steps</th>
    <th>Trace</th>
    <th>Result</th>
    <th>Test duration</th>
</tr>
```

To populate the table with data, we are going to iterate through all the trace events returned by SpecFlow, ignoring any that are irrelevant:

```
@foreach (var traceEvent in test.Result.TraceEvents)
{
    if (!IsRelevant(traceEvent))
    {
        continue;
    }
```

. . . and add a row to the table for each relevant entry, with each column’s contents matching the header . . .

```
<tr>
    <td>@Index</td>
    <td>
        <pre class="log">@(traceEvent.BusinessMessages.TrimEnd())</pre>
    </td>
    <td>
        <pre class="log">@Raw(FormatTechMessages(traceEvent.TechMessages.TrimEnd()))</
↪pre>
        @if (!string.IsNullOrEmpty(traceEvent.Error))
        {
          <div class="errorMessage">@Raw(FormatTechMessages(traceEvent.Error))</div>
           <pre class="stackTrace">@Raw(FormatTechMessages(traceEvent.StackTrace.
↪TrimEnd()))</pre>
         }
    </td>
    <td>@traceEvent.ResultType</td>
    <td>@GetSeconds(Math.Round(traceEvent.Duration.TotalSeconds, 3))s</td>
</tr>
```

Finally, we need to increment the Index by one before looping through the next entry:

```
    Index = Index +1;
}
</table>
```

Making Things Pretty

You can customize the colours used by the table in the CSS section of the template. Search for `<style type="text/css">` to locate this section. I have used the following settings, setting the header to light grey (#e8eef4) with a black font (#000000), and using alternating blues (#66aaee and #99ccf1) for the table rows:

```
        /* Sortable tables */
table.sortable thead {
    background-color:#e8eef4;    /*header bg color*/
    color:#000000;               /*Header font color*/
    font-weight: bold;
    font-size:large;
    cursor: default;
}

table.sortable tbody tr:nth-child(2n) td {
  background: #66aaee;
}
table.sortable tbody tr:nth-child(2n+1) td {
  background: #99ccf1;
}
/* SORTABLE END*/
```

Final Steps

While you can now run your tests and generate the report, you still need to ensure that the HTML output can access `sorttable.js`. For now, the just copy the file to your report's output directory. In a real-life scenario you will probably want to ensure that this file is automatically deployed to this directory whenever you run your tests.

30.4.2 The Final Output

Once you have copied the JavaScript file to your output directory and run your tests, open the resulting HTML file.

Click on a column header to sort the entries in that column:

Click in the header again to change between ascending/descending order:

Click on the Index column's header to return to the default sort order:

30.4.3 Including Charts

By default, the SpecFlow+ results output information on the number of tests that were successful, failed, are pending etc. In this example, we are going to display this information as a pie chart as well:

To do so, we are going to use Highcharts to display a pie chart. Other types of charts are also supported by Highcharts, and integrating them is very similar.

Implementing a Render Function

We will implement a function to render the pie chart that is very similar to the pie chart sample code, but we are going to populate the chart with the results data from SpecFlow+. Add the following function to your template – preferably with the other functions. I have added the function after the `doSetHeights` function and before `$(document).ready`:

```
function renderPieChart() {
  $('#chart').highcharts({
    chart: {
      plotBackgroundColor: null,
      plotBorderWidth: null,
      plotShadow: false,
      type: 'pie'
    },
    title: {
      text: ''
    },
    tooltip: {
      pointFormat: '{series.name}: <b>{point.percentage:.1f}%</b>'
    },
    plotOptions: {
      pie: {
        allowPointSelect: true,
        cursor: 'pointer',
        dataLabels: {
          enabled: true,
          format: '<b>{point.name}</b>: {point.percentage:.1f} %',
          style: {
            color: (Highcharts.theme && Highcharts.theme.contrastTextColor) || 'black'
          }
        }
      }
    },
    series: [{
      name: 'Brands',
      colorByPoint: true,
      data: [{
        name: 'Succeeded',
        y: @Model.Summary.Succeeded @*Number of successful tests*@
      }, {
        name: 'Failures',
        y: @Model.Summary.TotalFailure,   @*Number of failed tests*@
          }, {
        name: 'Pending',
        y: @Model.Summary.TotalPending  @*Number of pending tests*@
      }, {
        name: 'Ignored',
        y: @Model.Summary.Ignored   @*Number of ignored tests*@
  }, {
name: 'Skipped',
  y: @Model.Summary.Skipped   @*Number of skipped tests*@
  }]
  }]
  });
}
```

Be careful, that you add the new function not within another function. If not, you will get into problems in the next step.

The main things to note in this code:

- "#chart" refers to the ID of the `<div>` element that will contain the chart.
- The names of the individual slices are hard-coded as "Succeeded", "Failures" etc.

- The data (`series`) used by the chart is specified using `@Model.Summary.XYZ`, where `XYZ` is the value for each slice of the pie chart. You can obviously pass other numeric data from SpecFlow+ (e.g. execution times).

Rendering the Chart

We still need to render the chart by calling the renderPieChart function once the document is ready. Locate `$(document).ready(function ()`, and add the following line at the end of the function:

`renderPieChart();`

It should then look like this:

```
$(document).ready(function () {
  $("input[name='barSortOrder']").click(function () {
    doSort(true);
    return true;
  });
  $("input[name='barSortDesc']").change(function () {
    doSort(false);
  });
  $("input[name='barHeight']").change(function () {
    doSetHeights(true);
  });

  doSort(false);
  doSetHeights(false);

  $("div.scrollable").css({ overflow: "auto" });

  renderPieChart();
});
```

Including the Chart in the Report

The final step is to include the chart in your report at the appropriate position. In this case, we want to display the pie chart immediately after the overall summary of the test results. Search for `<h2>Result: @Model.Summary.ConcludedResultMessage</h2>` and add the following code after the `</table>` tag:

```
    <script src="https://code.highcharts.com/highcharts.js"></script>
    <script src="https://code.highcharts.com/modules/exporting.js"></script>
    <div id="chart" style="min-width: 310px; height: 400px; max-width: 600px; margin:␣
↪0 auto"></div>
```

Note that the ID of the `div` element is "chart", which matches the value defined in `renderPieChart`. You can move the script references to elsewhere in the document, but make sure they are included!

Note: Because the referenced JavaScript files are available online, you do not need to copy the files to your output directory.

Result

You can download the resulting report template from here.

30.5 Including Screenshots

A common requirement is to include screenshots in the report. SpecFlow+ Runner does not currently include functions for taking an embedding screenshots. However, you can still include screenshots in your report. The process is a little complicated, and involves passing information on the screenshot to the report using `Console.Write()`. The easiest way to understand how the process works is to look at an existing example. You can download a sample project here. In this project, a screenshot is taken after each step. While this project uses Selenium as the testing framework, the principle is the same for other frameworks as well:

1. Take a screenshot at the right moment during your test run.
2. Save the screenshot to file.
3. Pass the screenshot's file path to the report using Console.Write()
4. Parse the information received by the report to strip out the information relating to the screenshot, and embed the screenshot in the HTML.

Note: In order to run Selenium tests with FireFox, you may need to download the gecko driver. Copy `geckodriver.exe` to your `SpecFlow.Plus.Examples\SeleniumWebTest\TestApplication.UiTests\bin\Debug` directory or add it to your system's PATH environment settings.

Once you have downloaded the solution and installed the gecko driver:

1. Open the solution (`TestApplication.sln`) in Visual Studio.
2. Visual Studio will download the NuGet packages required by the solution.
3. Once the solution has finished loading and installing the packages, build the solution.
4. Switch to the Test Explorer in Visual Studio. You should have 7 tests:
5. Run the tests to verify that everything is set up correctly.
6. Once the tests have completed, open the report file (the link is in the **Output** pane in Visual Studio).

30.5.1 The Report

Scroll down in the report until you reach a **Steps** section. For each successfully completed step, you should see a screenshot in the **Trace** column:

30.5.2 How it Works

Taking screenshots is handled in `Screenshots.cs` in the `TestApplication.UiTests` project's Support folder:

```
    [Binding]
    class Screenshots
    {
        private readonly WebDriver _webDriver;

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

        [AfterStep()]
        public void MakeScreenshotAfterStep()
```

(continues on next page)

(continued from previous page)

```
        {
            var takesScreenshot = _webDriver.Current as ITakesScreenshot;
            if (takesScreenshot != null)
            {
                var screenshot = takesScreenshot.GetScreenshot();
                var tempFileName = Path.Combine(Directory.GetCurrentDirectory(), Path.
↪GetFileNameWithoutExtension(Path.GetTempFileName())) + ".jpg";
                screenshot.SaveAsFile(tempFileName, ImageFormat.Jpeg);

                Console.WriteLine($"SCREENSHOT[ file:///{tempFileName} ]SCREENSHOT");
            }
        }
    }
```

The function `MakeScreenshotAfterStep()` is responsible for taking a screenshot once each step is completed. The `[AfterStep()]` hook before the function declaration ensures that this function is executed once each scenario step has been completed.

In this case, the screenshot is taken using the Selenium API:

```
var screenshot = takesScreenshot.GetScreenshot();
```

Note: If you are using a different testing framework, replace this with the appropriate screenshot function for your framework.

Once the screenshot has been taken, we need to save the screenshot to the current directory. The file name is stored in `tempFileName`, and used to both save the image file (as a JPG) and to pass the path to the screenshot to the report.

Passing the file path to the report is handled using the console. Any data written to the console is available to the report template. In this case, we are writing the file name with some extra padding (`SCREENSHOT[ <PATH>] SCREENSHOT`) to allow us to uniquely identify file paths in the report template.

The path passed to the report template needs to be processed by the report template so that we can embed the specified image. Open `ReportTemplate.cshtml` (in the `Report` folder of the `TestApplication.UiTests` project). Scroll down to the bottom of the file. The following code is responsible for the content of the **Trace** column in the table:

```
<td>
   <!-- [@traceEvent.Type: @relatedNode.Type - @relatedNode.Title] -->
   <pre class="log">@Raw(FormatTechMessages(traceEvent.TechMessages.TrimEnd()).
↪Replace("SCREENSHOT[ <a href=", "<img width='1000' src=").Replace("</a> ]SCREENSHOT
↪", "</img>"))</pre>
   @if (!string.IsNullOrEmpty(traceEvent.Error))
   {
      <div class="errorMessage">@Raw(FormatTechMessages(traceEvent.Error))</div>
      <pre class="stackTrace">@Raw(FormatTechMessages(traceEvent.StackTrace.
↪TrimEnd()))</pre>
   }
</td>
```

By default, the file path is passed to the report as a hyperlink (`<a href>`). We want to embed the image in the document (using an `<img>` tag), and remove the padding used to identify the screenshot path (`SCREENSHOT [<PATH>] SCREENSHOT`).

This is handled by the following line:

```
class="log">@Raw(FormatTechMessages(traceEvent.TechMessages.TrimEnd()).Replace(
↪"SCREENSHOT[ <a href=", "<img width='1000' src=").Replace("</a> ]SCREENSHOT", "</
↪img>"))</pre>
```

(continues on next page)

(continued from previous page)

The `SCREENSHOT [` opening padding and the opening hyperlink tag are replaced by the HTML image tag including formatting. The file name is left unchanged. The closing hyperlink tag and `] SCREENSHOT` padding is replaced by a closing image tag. The result is to embed the image in the generated report using the file path passed to the report via the console.

30.5.3 Other Environments

While this example uses Selenium, it should be easy to edit the function that takes a screenshot after each step to use the appropriate screenshot function for you target environment. What is important to remember is that you can abuse the console to pass extra information to your reports. However, be aware that by default, the console output is simply written to your report "as is". You will need to parse the data passed by the console in your report template, and remove any information (such as padding) that should not appear in the report. This trick can be used to pass any kind of information, not just paths to screenshots.

CHAPTER 31

Available API documentations

31.1 TestRunContext

The TestRunContext supplies you with information about your current test run. You can get an instance of it through context injection.

31.1.1 Public properties

string TestDirectory {get;}

Provides the path to the folder of the test assembly

CHAPTER 32

Troubleshooting

This page covers some of the most common issues that you may encounter when using SpecFlow+.

- I'm trying to run my SpecFlow+ Runner tests in Visual Studio test window, but they fail with an assembly load error ("System.IO.FileNotFoundException: Could not load file or assembly")

In some cases the cache folder of Visual Studio Test Adapter gets corrupted. You need to clear the cache to resolve this:

1. Close all Visual Studio instances
2. Open the folder `%TEMP%\VisualStudioTestExplorerExtensions\`
3. Delete all folders named `*SpecRun*`

- I'm trying to use 32-bit assemblies on a 64-bit machine and receive a System.BadImageFormatException despite setting the environment to x86 in my .srprofile

This is a known issue. As a workaround, you need to set SpecRun.exe to run as a 32-bit process. Do this via the Visual Studio/Developer Command Prompt as follows: corflags YourFolder\SpecRun.exe /32BIT+.

- The Test Explorer is not showing any feature scenario tests

- If you have added SpecFlow+ Runner to your project or updated the installed package, close and restart Visual Studio. This step is necessary to allow Visual Studio to load the SpecFlow+ Runner test adapter and locate your tests.
- If cleaning your solution and rebuilding does not fix this issue, try deleting the contents (all folders and files) in your `%temp%\VisualStudioTestExplorerExtensions` folder.

- My tests are not displayed in the Test Explorer in Visual Studio 2015

This is an issue caused by a change in how Visual Studio handles solution-level packages. You can fix this issue by reinstalling the SpecFlow+ Runner NuGet packages or by adding the dependency on the `SpecRun.Runner` package to `packages.config` (`<package id="SpecRun.Runner" version="1.2.0" />`).

- I have updated SpecFlow+ and my test are no longer running

If you have updated SpecFlow+ with a new NuGet package, and are having issues, please check whether the previous versions have been removed completely. Visual Studio does not always remove previous packages when updating. If this happens, the wrong (earlier) version of SpecFlow+ may be used by Visual Studio which can cause version conflicts.

1. Switch to the folder containing your solution in Windows Explorer.
2. Open the `packages` directory.
3. If you see directories with an older version number in the name, delete those directories until you only have 1 directory for each NuGet package.

- I receive a message that I do not have permission to execute install.ps1 when installing SpecFlow+

The installation executes a powershell script. If you do not have permission to execute this script on the machine, the installation will fail. If you do not have the necessary privileges, try entering the following command in the NuGet console:`PM> Set-ExecutionPolicy RemoteSigned`

Once the command has executed, restart the installation process.

- I am receiving an Exception Code 0xc00000fd when executing my tests. What is causing this?

0xc00000fd is the code for a stack overflow exception. This exception can occur if you access recursive code from your bindings where the recursion is not terminated (or not terminated before a stack overflow exception occurs). The error is most likely in your recursive code itself.

- Why are my failing tests flagged as failing when they pass on a retry?

This is working as intended. If at least one test fails, the test has not passed successfully. If the initial test fails, there is at least one circumstance where the test fails. Were SpecFlow to flag these tests as passing, it would be ignoring the case where the test fails. There can be many reasons why the first test fails and the second passes - the first step is to determine why this is the case.

If you are absolutely sure that the reason the first test fails and all retries pass is not an issue with your code, but with your architecture, you can run your tests with *VSTest* and configure a minimum pass threshold.

Setting the test results to "Unified" with a minimum threshold allows you to treat a test as passing if a certain amount of the tests (initial test+retries) pass. This option should be used with great caution. Unless you are 100% certain that the random pass/fail behaviour is not related to your code, **do not use this option**. The option was added for a specific use case: the initial test would **always** fail due to a timeout on a remote server; this was not an issue for retried tests.

Using this option will generate a TRX file where tests will be marked as passing above the pass threshold. This option is not available for SpecFlow+ reports, as the results are only unified when running tests with VSTest.

Remember that a test that sometimes passes and sometimes fails can point to deeper issues with your tests or code, and should not simply be dismissed.

- I am receiving Could not load file or assembly Microsoft.AspNetCore.Hosting.Abstractions

The issue occurs in the HTML report generation which is executed using mono on Linux.

If you do not need the HTML report, you can simply disable the report generation in your .srprofile by adding `<Report disable="true"/>` to `TestProfile`. In the docs, you can see an *example* of how to disable the report generation.

If you need the HTML report, you might have to update both .NET Core SDK and mono. .NET Core SDK version **3.1.403** and mono version **6.12.0** are confirmed to be working properly with the HTML report generation on Linux.

CHAPTER 33

Known Issues

The following are known issues with SpecFlow+, and are scheduled to be fixed:

- "Object reference not set to an instance of an object" when using NUnit's Assert command in conjunction with NUnit tests executed with SpecFlow+ Runner.**Note:** This issue is **fixed in NUnit 3.8**. If you are experiencing this issue, please update your version of NUnit.
- Only a blank page is displayed when viewing the living documentation generated by SpecFlow+ LivingDoc in Chrome on OSX.

The following issues are known, and no fix is scheduled:

- When using **R# Build**, the test discovery process is not started for the Test Explorer. R# Build is thus incompatible and cannot be used with SpecFlow+ Runner.
- SpecFlow+ Runner does not support .NET6