# Introduction

* SpecFlow is …
* This document describes the features and the usage of SpecFlow.
* To learn about the idea behind SpecFlow and about how to integrate it into the development process, see
  + BDD/ATDD: link
  + SpecFlow WorkFlow
  + Cucumber
  + Other practices & samples
* This document uses examples from the SpecFlow BookShop sample, that can be downloaded from here: <link>

# Installation

SpecFlow is distributed as a Windows Installer MSI file. In order to install it to your machine you have to execute this installer.

SpecFlow can integrate to Visual Studio 2008 and Visual Studio 2010. During the installation process you can decide whether to install the integration to the different Visual Studio versions.

The installer deploys the necessary files to the specified folder (default: C:\Program Files\TechTalk\SpecFlow) and registers the Visual Studio Integration.

# Execution of SpecFlow Tests

With SpecFlow you can define the acceptance criteria in .feature files that can be executed. These tests are usually placed in a separate project in the solution (e.g. "BookShop.AcceptanceTests" in the BookShop sample).

SpecFlow generated executable unit tests from the defined acceptance criteria (called scenarios). These generated unit tests are in the generated sub-items if the feature files (e.g. US01\_BookSearch.feature.cs).

The execution of the tests depends on the unit test provider used by SpecFlow (currently NUnit, MsTest and xUnit is supported). The unit test provider can be configured in the app.config file of the test project:

<specFlow>

<unitTestProvider name="MsTest" />

</specFlow>

For example for MsTest unit test provider, the acceptance tests can be executed with the following steps:

1. Select the acceptance test project (e.g. "BookShop.AcceptanceTests") in solution explorer.
2. Select command from the main menu: Test / Run / Tests in Current Context (Ctrl R,T)

## Test Output and Result

When SpecFlow tests are executed the execution engine processes the test steps, executes the necessary test logic and either finish successfully or fails with some reason.

### Test Passes

While the tests are executed the engine also outputs useful information about the execution to the test output. Therefore in some cases it makes sense to investigate the test output even if the test was passing.

The test output shows by default the executed test steps, the invoked test logic methods (bindings) and the execution time of the longer operations. The information displayed in the test output can also be configured. See [<trace>](#_<trace>) configuration element.

### Test Fails because of a Test/Business Logic Error

A test can fail because the test/business logic reports an error. This is reported as a test error, you can investigate the test output for the detailed information (e.g. stack trace) of the error.

### Missing, Pending or Improperly Configured Bindings

The test can also fail because some parts of the test logic (bindings) were not implemented yet (or configured improperly). This is reported by default with the “inconclusive” result. You can change how SpecFlow should behave in this case. See [<runtime>](#_<runtime>) configuration element.

Some unit test framework does not support inconclusive result. In this case the problem is reported as an error by default.

The test output can be very useful for missing bindings as it contain a step binding method skeleton that you can copy to your project and fill-in with the test logic.

### Ignored Tests

Just like with normal unit tests, you can also ignore a SpecFlow test. This can be done by marking the scenario with the @ignore [tag](#_Tags).

## Debugging Tests

TBD

# Technical Concept

* allows adding feature files to the projects (C#, VB.NET)
* the installed SpecFlow single-file generator generates a unit test when the feature file is saved
  + you can force generation from context menu: “Run Custom Tool”
* the generated unit test can be executed with the unit test execution tools
  + unit test provider has to be configured
  + the project type might depend on the selected unit test provider (e.g. Test Project for MsTest)
* the executed tests call out to the test logic (“bindings”)
* the bindings can drive the application and implement the automation of the test steps
* SpecFlow: generator part and runtime part

# Setup SpecFlow Tests

Generally, just like for test-driven development, behavior-driven development works the best if it is integrated into the development process with a test-first approach and using and outside-in approach. Therefore it is important to setup the environment for the SpecFlow tests from the beginning of the application development.

The SpecFlow tests are usually placed into one or more separate project in the solution. Just like for unit tests, it makes sense to keep a consistent naming convention for these projects. The BookShop sample manages the acceptance criteria in a project called "BookShop.AcceptanceTests".

As mentioned in the [Technical Concept](#_Technical_Concept) section, SpecFlow uses a unit testing framework to execute the acceptance criteria defined in the feature files. Therefore setting up a project for the SpecFlow is very similar to set up a unit test project.

1. Create a new project that suits to the selected unit-testing framework (usually a Class Library, but for MsTest you need to create a Test Project.
2. Add a reference to the created project for the unit-test framework library (e.g. to nunit.framework.dll for NUnit).
3. Add a reference to the SpecFlow runtime assembly (TechTalk.SpecFlow.dll).
4. Add an “App.config” file to configure the unit test provider and other options of SpecFlow (see [Configuration](#_Configuration)). This step is optional if you use NUnit.
5. Optional: Create a “StepDefinitons” folder for the step binding classes.
6. Optional: Create a “Support” folder for other infrastructural code (e.g. event bindings).

Note: SpecFlow might provide later a Visual Studio project template to accomplish these steps.

As the SpecFlow runtime assembly is not installed into the GAC, it is recommended to copy this assembly into the project source folder and store it in the source control together with your code. The other assemblies installed by SpecFlow are only required for the Visual Studio integration, reporting and other tools, so they are not necessarily required to keep together with your source.

### First Feature File

After configuring the project you can add the first feature files to your project (usually into the root folder of the project). This can be done using the Visual Studio item template (Add / New Item…), called “SpecFlow Feature”.

This item template creates you a new feature file with a sample scenario. Change the file content according to the specification of your application and save the file. SpecFlow will generate the supporting unit test that you can execute to start the outside-in development process. See also: [Missing, Pending or Improperly Configured Bindings](#_Missing,_Pending_or)

You can read more about structuring the feature files and the bindings here: <url>

### Regenerate Unit Tests

Although normally working with the feature files in Visual Studio keeps the generated unit test code consistent with the feature file, in some cases you might want to re-run the generation (e.g. because you have modified the feature file outside of Visual Studio).

Re-generation can be forced by saving the files explicitly or doing the following steps:

1. Select the feature files in solution explorer (you can select multiple files within a project).
2. Invoke "Run Custom Tool" command from the context menu.

You can also configure the project to refresh the feature files (if necessary) before each compilation. Read more about this option in the [Generate All](#_Generate_All) section. This might be useful if you regularly use another tool to edit the feature files.

# Upgrade Project to a Newer SpecFlow Version

TBD

* The cleanest way for upgrading is to install the new specflow installer (that replaces the visual studio integration so the generator part) and also replace the runtime part (TechTalk.SpecFlow.dll) in your project. After doing that re-generate the unit tests ([Regenerate Unit Tests](#_Regenerate_Unit_Tests)).
* The interface generator part and the runtime is usually compatible: the new runtime can run tests generated by the old generator and vica versa
* However, to take the advantage of the new generator features (or in case of a breaking change in the generator-runtime interface) you might need to regenerate all tests, see [Regenerate Unit Tests](#_Regenerate_Unit_Tests)

# Configuration

The behavior of SpecFlow can be extensively configured through .NET configuration files. SpecFlow processes the configuration file of the acceptance test projects (the projects that contain the feature files). The configuration has to be placed in a file called “App.Config” (the standard configuration file convention for .NET).

Unlike other runtime-only tools, SpecFlow processes the configuration file also while it generates the unit-tests from the feature files (this happens usually when you save the feature file). This means that after you have changed the configuration file, you might need to force re-generation of the unit test (if the configuration change affects the generated tests). See [Regenerate Unit Tests](#_Regenerate_Unit_Tests) for details about how this can be done.

## Default Configuration

In SpecFlow all configuration option has a default setting, so in an extreme case you don’t need to specify any configuration file.

Commonly the most important thing to configure is the [unit test provider](#_<unitTestProvider>). Therefore simple SpecFlow projects configure only this aspect. The following example shows such a simple configuration.

<?xml version="1.0" encoding="utf-8" ?>

<configuration>

<configSections>

<section name="specFlow"

type="TechTalk.SpecFlow.Configuration.ConfigurationSectionHandler,

TechTalk.SpecFlow"/>

</configSections>

<specFlow>

<unitTestProvider name="MsTest" />

</specFlow>

</configuration>

The following example shows all possible configuration option with their default values.

<specFlow>

<language feature="en-US" tool="" />

<unitTestProvider name="NUnit" />

<generator allowDebugGeneratedFiles="false" />

<runtime detectAmbiguousMatches="true"

stopAtFirstError="false"

missingOrPendingStepsOutcome="Inconclusive" />

<trace traceSuccessfulSteps="true"

traceTimings="false"

minTracedDuration="0:0:0.1"

listener="TechTalk.SpecFlow.Tracing.DefaultListener,

TechTalk.SpecFlow" />

</specFlow>

## Configuration Elements

### <language>

This section can be used to define the default language for the feature files and other language-related settings. Learn more about the language settings in the [Feature Language](#_Feature_Language) section.

|  |  |  |
| --- | --- | --- |
| Attribute | Value | Description |
| feature | culture name (“en-US”) | The default language of the feature files added to the project. It is recommended to use specific culture names (e.g.: “en-US”) and not generic (neutral) cultures (e.g.: “en”).  Default: en-US |
| tool | empty or culture name | Specifies the language that SpecFlow uses for messages and tracing. Uses the default feature language if empty. (Currently only English is used for messages.)  Default: empty |

### <unitTestProvider>

This section can be used to specify the unit-test framework SpecFlow uses to execute the acceptance criteria. You can either use one of the built-in unit-test providers or you can specify the classes that implement the custom unit test providers.

|  |  |  |
| --- | --- | --- |
| Attribute | Value | Description |
| name | “NUnit”, ”MsTest” or “xUnit” | The name of the built-in unit test provider. If you specify this attribute, you don’t have to specify the other two.  Default: NUnit |
| generatorProvider | class name | An assembly qualified class name of a class that implements TechTalk.SpecFlow.Generator. UnitTestProvider.IUnitTestGeneratorProvider interface. |
| runtimeProvider | class name | An assembly qualified class name of a class that implements TechTalk.SpecFlow.  UnitTestProvider.IUnitTestRuntimeProvider interface. |

### <generator>

This section can be used to specify various unit-test generation options.

|  |  |  |
| --- | --- | --- |
| Attribute | Value | Description |
| allowDebugGeneratedFiles | “true” or “false” | The debugger is by default configured to step through the generated code. This helps to debug from the feature files directly to the bindings (see [Debugging Tests](#_Debugging_Tests)). This feature can be disabled by setting this attribute to “true”.  Default: false |

### <runtime>

This section can be used to specify various unit-test execution options.

|  |  |  |
| --- | --- | --- |
| Attribute | Value | Description |
| detectAmbiguousMatches | “true” or “false” | Specifies whether SpecFlow should report an error if there is an ambiguous match of step binding or just use the first one that matches.  Default: true |
| stopAtFirstError | “true” or “false” | Specifies whether the execution should stop at the first error or should continue to try matching the subsequent steps (in order to detect missing steps).  Default: false |
| missingOrPendingStepsOutcome | “Inconclusive”, ”Ignore” or “Error” | Specifies how SpecFlow should behave if a step binding is not implemented or pending. See [Missing, Pending or Improperly Configured Bindings](#_Missing,_Pending_or).  Default: Inconclusive |

### <trace>

This section can be used to configure how and what should SpecFlow trace out to the unit test output.

|  |  |  |
| --- | --- | --- |
| Attribute | Value | Description |
| traceSuccessfulSteps | “true” or “false” | Specifies whether SpecFlow should trace successful step binding executions.  Default: true |
| traceTimings | “true” or “false” | Specifies whether SpecFlow should trace execution time of the binding methods (only if the execution time is longer than the minTracedDuration value).  Default: false |
| minTracedDuration | TimeSpan (0:0:0.1) | Specifes a threshold for tracing the binding execution times.  Default: 0:0:0.1 (100 ms) |
| Listener | class name | An assembly qualified class name of a class that implements TechTalk.SpecFlow.  Tracing.ITraceListener interface. SpecFlow provides DefaultListener and NullListener as default implementations.  Default: TechTalk.SpecFlow.Tracing.DefaultListener, TechTalk.SpecFlow |

# Gherkin Language Elements

## Features

## Scenarios

## Scenario Steps

## Table and multi-line text arguments

## Tags

## Scenario Outlines

## Feature Language

# Bindings

## Step Bindings

## Event Bindings

## Argument Conversions

### Simple Conversions

### Step Argument Transformations

## Communication between Bindings

### Instance Fields

### Static Fields

### ScenarioContext and FeatureContext

### Context Injection

# Tools

## Generate All

# Reporting

## Test Execution Report

nunit-console.exe /labels /out=TestResult.txt /xml=TestResult.xml bin\Debug\BookShop.AcceptanceTests.dll

specflow.exe nunitexecutionreport BookShop.AcceptanceTests.csproj /out:MyResult.html

## Step Usage Report

specflow.exe stepdefinitionreport BookShop.AcceptanceTests.csproj