Testing using SpecFlow and Gherkin scenarios in C#

Overview

In this tutorial, we will create some test scenarios in SpecFlow using C#.

Similarly to Cucumber, SpecFlow is mainly a collaboration framework used in BDD context in order to improve shared understanding within the team, usually during "3 Amigos" sessions. That's its main fit.

However, some teams use it in other contexts (e.g. after sofware has being built) for implementing automated tests and take advantage of Gherkin syntax to have visibility/abstraction of the underlying automation code and have reusable automation code.

(Test) Scenarios derived from SpecFlow are executable specifications; their statements will have a corresponding code implementation. These test scenarios are feature and more business oriented; they're not unit/integration tests.

Your specification is made using Gherkin (i.e. Given, When, That) statements in Scenario(s) or Scenario Outline(s), eventually complemented with a Background. Implementation of each Gherkin statement (i.e. "step") is done in code; the SpecFlow framework finds the code based on regular or cucumber expressions.

Source-code for this tutorial

Code is available in GiHub; the repo contains some auxiliary scripts.

Usage scenarios

Tools such as Cucumber or SpecFlow are used in diverse scenarios. Next you may find some usage patterns, even though SpecFlow usage is mostly recommended only if you are adopting BDD.

1. Teams adopting BDD, start by defining a user story and clarify it using Gherkin Scenario(s); usualy, Gherkin Scenario(s)/Scenario Outline(s) are specified directly in Jira, using Xray
2. Teams adopting BDD but that favour a more Git based approach (e.g. GitOps). In this case, stories would be defined in Jira but Gherkin .feature files would be specified using some IDE and would be stored in Git, for example
3. Teams not adopting BDD but still using SpecFlow, more as an automation framework. Sometimes focused on regression testing; sometimes, for non-regression testing. In this case, SpecFlow would be used...
   a. With a user story or some sort of "requirement" described in Jira
   b. Without any story/"requirement" described in Jira

You may be adopting, or aiming to, one of the previous patterns.

Before moving into the actual implementation, we need to decide which workflow we'll use: do we want to use Xray/Jira as the master for writing the declarative specification (i.e. the Gherkin based Scenarios), or do we want to manage those outside using some editor and store them in Git, for example?

Learn more

Please see Testing in BDD with Gherkin based frameworks (e.g. Cucumber) for an overview of the possible workflows.

The place that you'll use to edit the SpecFlow Scenarios will affect your workflow. There are teams that prefer to edit Scenarios in Jira using Xray, while there others that prefer to edit them by writing the .feature files by hand using some IDE.

Example

For the purpose of this tutorial, we'll use a simple dummy Calculator implemented in a C# class as our target for testing.

The code is structure in two projects: one that we will be targeted by testing and one with the test code and SpecFlow.

> **Try it yourself!**
>
> The code on this tutorial is available in the tutorial-csharp-specflow GitHub repository.
>
> You can fork it, adapt, and try it for youself.

Calculator.cs

```
namespace SpecFlowExamples
{
    public class Calculator
    {
        public int FirstNumber { get; set; }

        public int SecondNumber { get; set; }

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

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

        public int Divide()
        {
            if (FirstNumber == 0 || SecondNumber == 0)
            {
                return 0;
            }
            else
            {
                return FirstNumber / SecondNumber;
            }
        }

        public int Multiply()
        {
            return FirstNumber * SecondNumber;
        }
    }
}
```

Requirements

- .NET 5.0 (.NET 6.0 is not supported by SpecFlow together with the SpecFlow+ Runner)
- mono (SpecFlow reporting utility requires mono)
- Packages
    - SpecFlow <= 3.9.40
    - SpecRun.SpecFlow <= 3.9.31
    - FluentAssertions

Please note

To generate the Cucumber compatible JSON report that Xray is able to process, **we need to use the "SpecFlow+ Runner" test runner**, which supports .Net Core up to .NET 5.0, and take advantage of the possibility of generating custom reports (more detail ahead).

Meanwhile, in January 2021 the SpecFlow team decided to end up with development of the "SpecFlow+ Runner" test runner. That means that the "SpecFlow+ Runner" package is avaiable for .NET up to version 5.0 but it is no longer maintained. Additionaly, this runner is not available for .NET 6.0, meaning that for .NET 6.0 there is no way of generating Cucumber JSON reports, as the supported test runners (e.g., NUnit, xUnit, MSTest) don't support it; if using SpecFlow and integrating it with Xray, or any other tool that uses Cucumber JSON reports, is important to you, then you should reach out SpecFlow team.

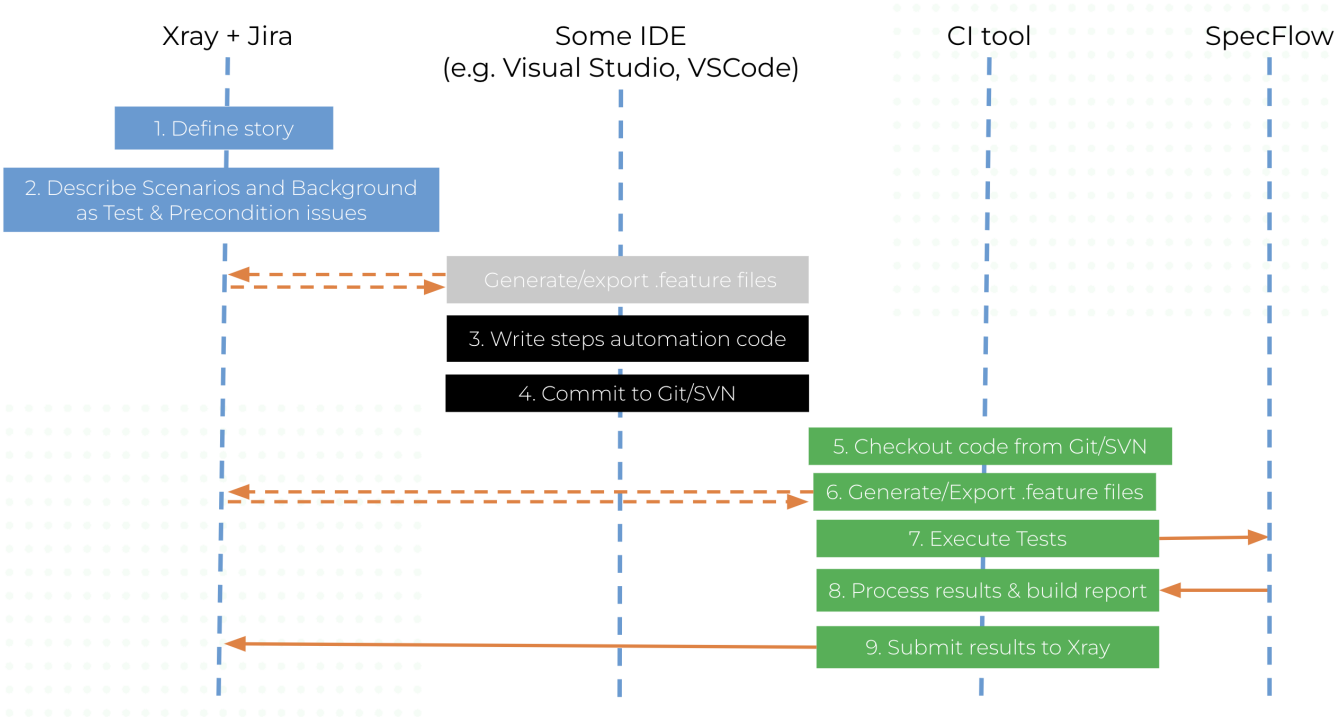
If you use NUnit, xUnit, or MSTest as test runners, then SpecFlow isn't able of generating Cucumber JSON reports.

Using Jira and Xray as master

This section assumes using Xray as master, i.e. the place that you'll be using to edit the specifications (e.g. the scenarios that are part of .feature files).

The overall flow would be something like this, assuming Git as the source code versioning system:

1. define the story (skip if you already have it)
2. create Scenario/Scenario Outline as a Test in Jira; usually, it would be linked to an existing "requirement"/Story (i.e. created from the respective issue screen)
3. implement the code related to Gherkin statements/steps and store it in Git, for example. To start, and during development, you may need to generate/export the .feature file to your local environment
4. commit previous code to Git
5. checkout the code from Git
6. generate .feature files based on the specification made in Jira
7. run the tests in the CI
8. obtain the report in Cucumber JSON format
9. import the results back to Jira

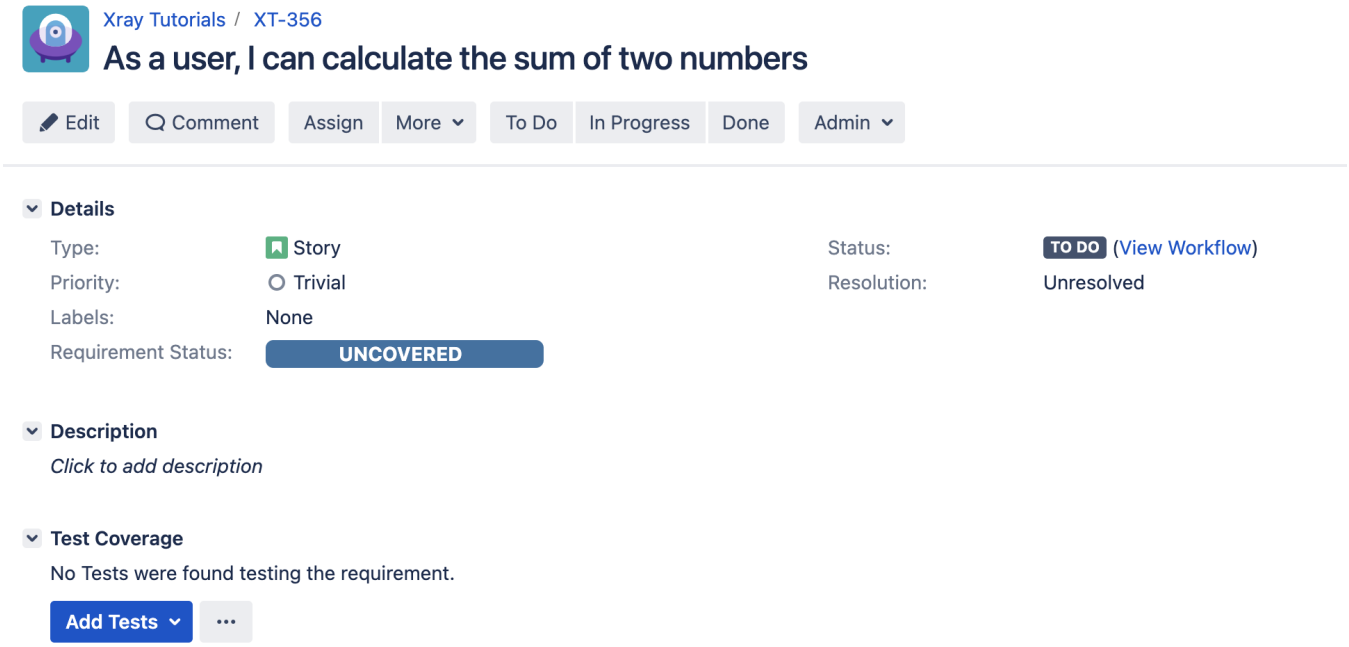


Note that steps (5-9) performed by the CI tool are all automated, obviously.

To generate .feature file(s) based on Scenarios defined in Jira (i.e. "Cucumber" Tests and Preconditions), we can do it directly from Jira, by the REST API or using a CI tool; we'll see that ahead in more detail.

Step-by-step

All starts with a user story or some sort of “requirement” that you wish to validate. This is materialized as a Jira issue and identified by the corresponding issue key (e.g. CALC-7931).



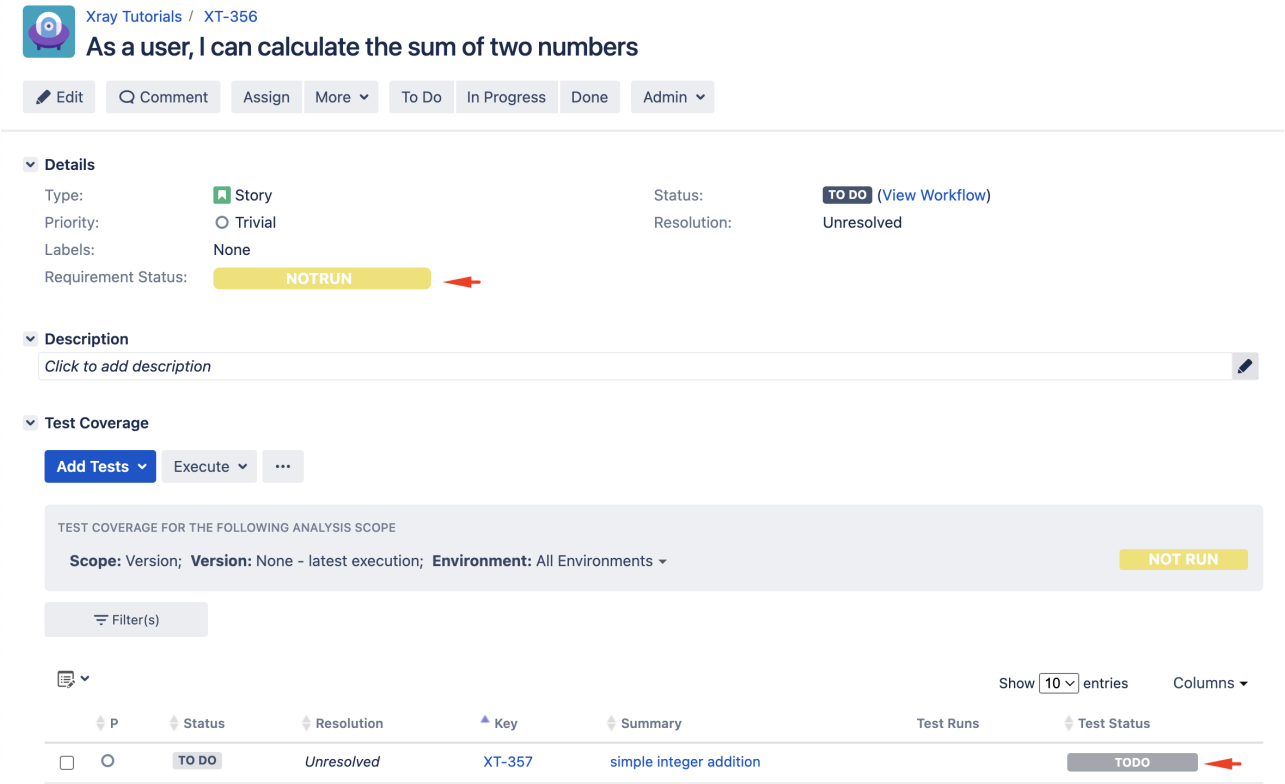
We can promptly check that it is “UNCOVERED” (i.e. that it has no tests covering it, no matter their type/approach).

In this case, we'll create a Cucumber Test, of Cucumber Type "Scenario".

We can fill out the Gherkin statements immediately on the Jira issue create dialog or we can create the Test issue first and fill out the details on the next screen, from within the Test issue. In the latter case, we can take advantage of the built-in Gherkin editor which provides auto-complete of Gherkin steps.

After the Test is created, and since we have done it from the user story screen, it will impact the coverage of related "requirement"/story.

The coverage and the test results can be tracked in the "requirement" side (e.g. user story). In this case, you may see that coverage changed from being UNCOVERED to NOTRUN (i.e. covered and with at least one test not run).



Additional tests could be created, eventually linked to the same Story.

The related statement's code is managed outside of Jira and stored in Git, for example.

Our target "application" to test, is in fact a simple Calculator class, stored and managed under the `Calculator` project directory.

The tests related code is stored under `Calculator.Specs` project directory, which itself contains several other directories. In this case, they're organized as follows:

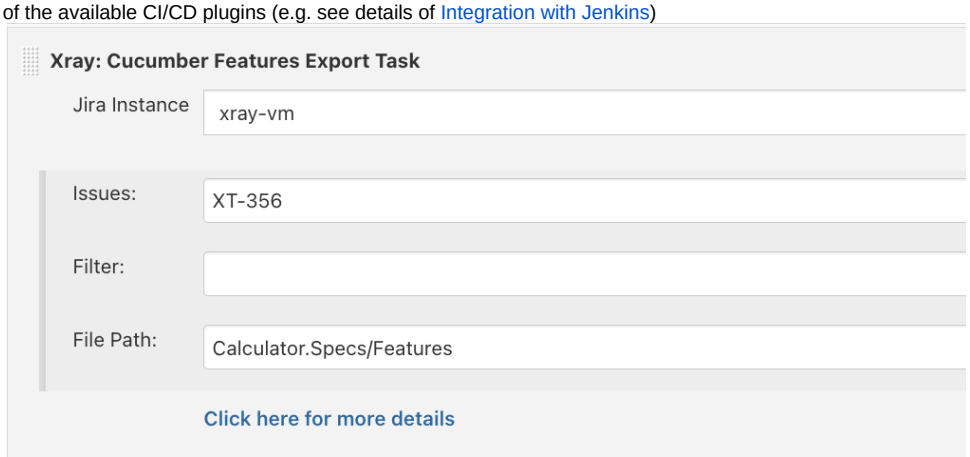
- `TestResults/`: where the test result reports will ge created.
- `Steps/`: step implementation files. The steps "glue-code" is defined in CalculatorStepDefinitions.cs class.

Calculator.Specs/Steps/CalculatorStepDefinitions.cs

```
using System;
using FluentAssertions;
using TechTalk.SpecFlow;

namespace SpecFlowExamples.Specs.Steps
{
    [Binding]
    public sealed class CalculatorStepDefinitions
    {
        private int _result;
        private Calculator _calculator = new Calculator();

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

        [Given(@"I have also entered (.*) into the calculator")]
        public void GivenIHaveAlsoEnteredIntoTheCalculator(int number)
        {
            _calculator.SecondNumber = number;
        }

        [When(@"I press add")]
        public void WhenIPressAdd()
        {
            _result = _calculator.Add();
        }

        [Then(@"the result should be (.*) on the screen")]
        public void ThenTheResultShouldBeOnTheScreen(int expectedResult)
        {
            // Assert.AreEqual(expectedResult, result);
            _result.Should().Be(expectedResult);
        }
    }
}
```

You can then export the specification of the test to a Gherkin .feature file via the REST API, or the **Export to Cucumber** UI action from within the Test /Test Execution issue or even based on an existing saved filter. As source, you can identify Test, Test Set, Test Execution, Test Plan or "requirement" issues. A plugin for your CI tool of choice can be used to ease this task.

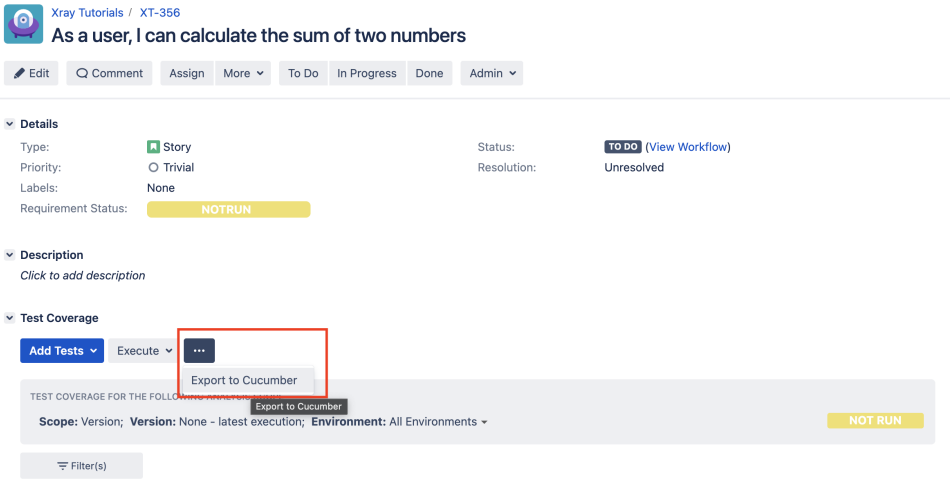
So, you can either:

- use one of the available CI/CD plugins (e.g. see details of Integration with Jenkins)
  - 

- use the REST API directly (more info here)

```
#!/bin/bash

rm -f Features/*.feature Features/*.cs
curl -u admin:admin  "http://jiraserver.example.com/rest/raven/1.0/export/test?keys=XT-356&fz=true" -o features.zip
unzip -o features.zip  -d Features
```

- ... or even use the UI (e.g. from the Story issue)



We will export the features to a new directory named `Features/` on the root folder of your C# testing-dedicated project.

After being exported, the created .feature(s) will contain references to the Test issue key, eventually prefixed (e.g. "TEST_") depending on an Xray global setting, and the covered "requirement" issue key, if that's the case. The naming of these files is detailed in Export Cucumber Features.

features/1_XT-356.feature

```
@REQ_XT-356
Feature: As a user, I can calculate the sum of two numbers


        @TEST_XT-359 @XT-355
        Scenario Outline: sum of two positive numbers
                Given I have entered <input_1> into the calculator
                And I have also entered <input_2> into the calculator
                When I press <button>
                Then the result should be <output> on the screen

                    Examples:
                            | input_1 | input_2 | button | output |
                            | 20      | 30      | add    | 50     |
                            | 2       | 5       | add    | 7      |
                            | 0       | 40      | add    | 40     |
                            | 4       | 50      | add    | 54     |
                            | 5       | 50      | add    | 55     |


        @TEST_XT-358 @XT-355
        Scenario: negative integer addition
                Given I have entered -1 into the calculator
                And I have also entered 2 into the calculator
                When I press add
                Then the result should be 1 on the screen


        @TEST_XT-357 @XT-355
        Scenario: simple integer addition
                Given I have entered 1 into the calculator
                And I have also entered 2 into the calculator
                When I press add
                Then the result should be 3 on the screen
```

Before compiling and running the tests, you have to use a proper SpecFlow report template file in order to generate a valid Cucumber JSON report and you have to configure the SpecFlow+ Runner profile (e.g., `Default.srprofile`) to use it.

Default.srprofile

```
<?xml version="1.0" encoding="utf-8"?>
<TestProfile  xmlns="http://www.specrun.com/schemas/2011/09/TestProfile">
  <Settings name="dummy" projectName="Calculator.Specs" />
  <Execution stopAfterFailures="3" testThreadCount="1" testSchedulingMode="Sequential" />

  <TestAssemblyPaths>
    <TestAssemblyPath>Calculator.Specs.dll</TestAssemblyPath>
  </TestAssemblyPaths>

  <Report copyAlsoToBaseFolder="false">
    <Template name="../../../CucumberJson.cshtml" outputName="cucumber.json" existingFileHandlingStrategy="
Overwrite" />
  </Report>

</TestProfile>
```

"SpecFlow+ Runner" supports customizable reports and provides a report template tailored for the generation of Cucumber JSON reports. You can can copy it from your local packages directory (e.g., `.nuget/packages/specrun.runner/3.9.31/templates/CucumberJson.cshtml`) to your test project directory. This template has changed slightly with SpecFlow 3, so please make sure you use the proper one for your scenario.

CucumberJson.cshtml

```
@inherits SpecFlow.Plus.Runner.Reporting.CustomTemplateBase<TestRunResult>
@using System
@using System.Collections.Generic
@using System.Linq
@using System.Globalization
@using Newtonsoft.Json
@using Newtonsoft.Json.Converters
@using TechTalk.SpecRun.Framework
@using TechTalk.SpecRun.Framework.Results
@using TechTalk.SpecRun.Framework.TestSuiteStructure
@using TechTalk.SpecRun.Framework.Tracing
@{
    var serializationSettings = new JsonSerializerSettings
    {
        ReferenceLoopHandling = ReferenceLoopHandling.Ignore,
        Converters = new List<JsonConverter>() { new StringEnumConverter(false) }
    };

    var features = GetTextFixtures()
        .Select(f => new
        {
            description = "",
            elements = (from scenario in f.SubNodes
                        let lastExecutionResult = GetTestItemResult(scenario.GetTestSequence().First()).
LastExecutionResult()
                        select new
                        {
                            description = "",
                            id = "",
                            keyword = "Scenario",
                            line = scenario.Source.SourceLine + 1,
                            name = scenario.Title,
                            tags = scenario.Tags.Select(t => new { name = t, line = 1 }),
                            steps = from step in lastExecutionResult.Result.TraceEvents
                                    where IsRelevant(step) && (step.ResultType == TestNodeResultType.Succeeded
|| step.ResultType == TestNodeResultType.Failed || step.ResultType == TestNodeResultType.Pending)
                                    && (step.Type == TraceEventType.Test || step.Type == TraceEventType.TestAct
|| step.Type == TraceEventType.TestArrange || step.Type == TraceEventType.TestAssert)
                                    let keyword = step.StepBindingInformation == null ? "" : step.
StepBindingInformation.StepInstanceInformation == null ? "" : step.StepBindingInformation.
StepInstanceInformation.Keyword
                                    let matchLocation = step.StepBindingInformation == null ? "" : step.
StepBindingInformation.MethodName
                                    let name = step.StepBindingInformation == null ? "" : step.
StepBindingInformation.Text
                                    let cucumberStatus = step.ResultType == TestNodeResultType.Succeeded ?
"Passed" : step.ResultType.ToString()
                                    select new
                                    {
                                        keyword = keyword,
                                        line = 0,
                                        match = new
                                        {
                                            location = matchLocation
                                        },
                                        name = name,
                                        result = new
                                        {
                                            duration = step.Duration.TotalMilliseconds,
                                            error_message = step.StackTrace,
                                            status = cucumberStatus
                                        }
                                    },
                            type = "scenario"
                        }).ToList(),
            id = "",
            keyword = "Feature",
            line = f.Source.SourceLine + 1,
```

```
            tags = f.Tags.Select(t => new { name = t, line = 1 }),
            name = f.Title,
            uri = f.Source.SourceFile
        });
}
@Raw(JsonConvert.SerializeObject(features, Formatting.Indented, serializationSettings))
```

Tests can be run from within the IDE (e.g. Visual Studio) or by the command line using `dotnet`, for example; in the later case, make sure to specify the profile name and all the paths properly.

To run the tests and produce a Cucumber JSON report, we can use `dotnet`, for example.

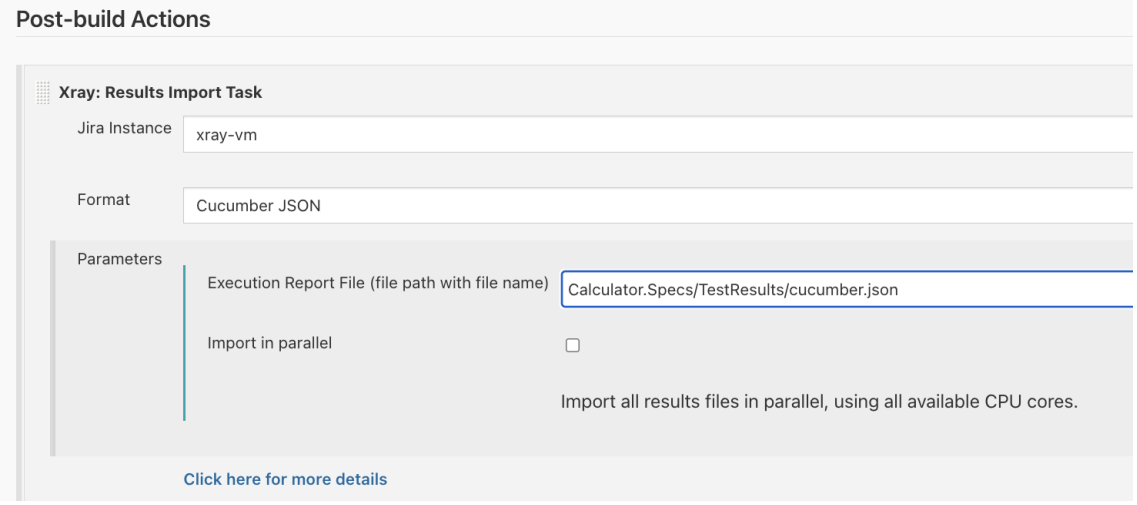
```
dotnet clean
dotnet test
```

This will produce one Cucumber JSON report with all results.

After running the tests, results can be imported to Xray via the REST API, or the **Import Execution Results** action within an existing Test Execution, or by using one of the available CI/CD plugins (e.g. see an example of Integration with Jenkins).

example of a Bash script to import results using the standard Cucumber endpoint

```
curl -H "Content-Type: application/json" -X POST -u admin:admin --data @"TestResults/cucumber.json"
http://jiraserver.example.com/rest/raven/2.0/import/execution/cucumber
```



Which Cucumber endpoint to use?

To import results, you can use two different endpoints/"formats" (endpoints described in Import Execution Results - REST):

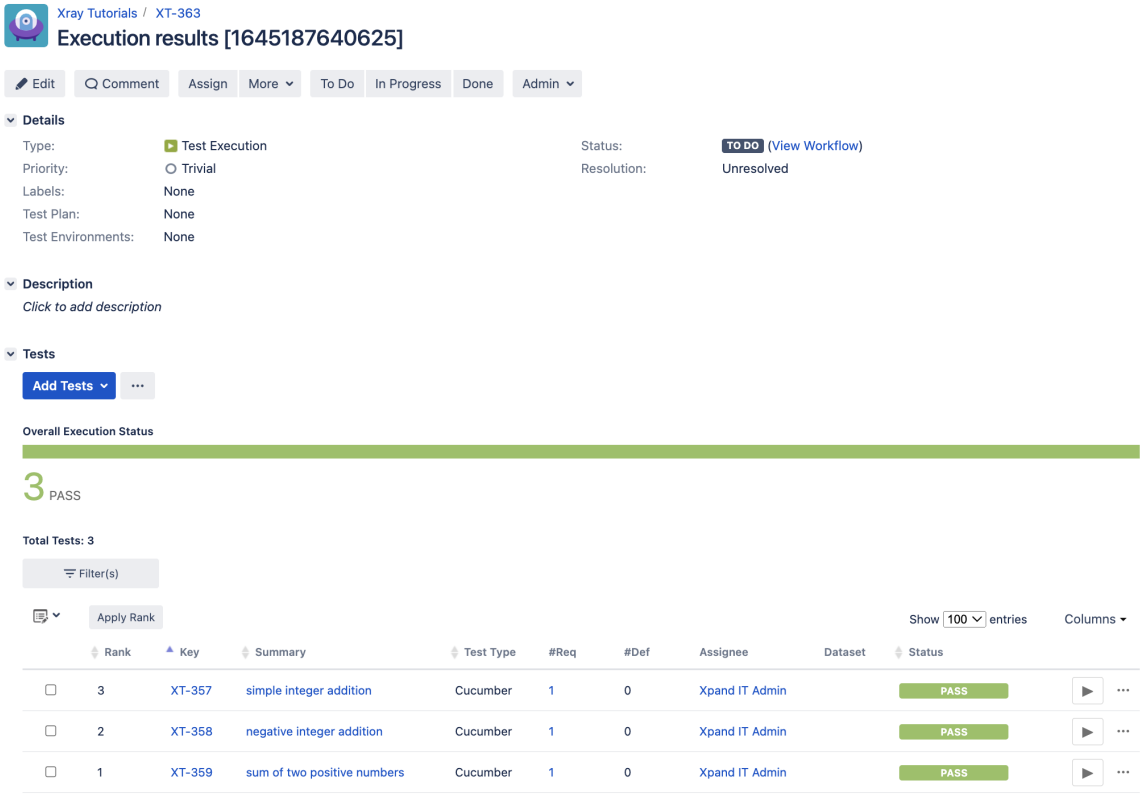
1. the "standard cucumber" endpoint
2. the "multipart cucumber" endpoint

The standard cucumber endpoint (i.e. */import/execution/cucumber)* is simpler but more restrictive: you cannot specify values for custom fields on the Test Execution that will be created. This endpoint creates new Test Execution issues unless the Feature contains a tag having an issue key of an existing Test Execution.

The multipart cucumber endpoint will allow you to customize fields (e.g. Fix Version, Test Plan), if you wish to do so, on the Test Execution that will be created. Note that this endpoint always creates new Test Executions (as of Xray v4.2).

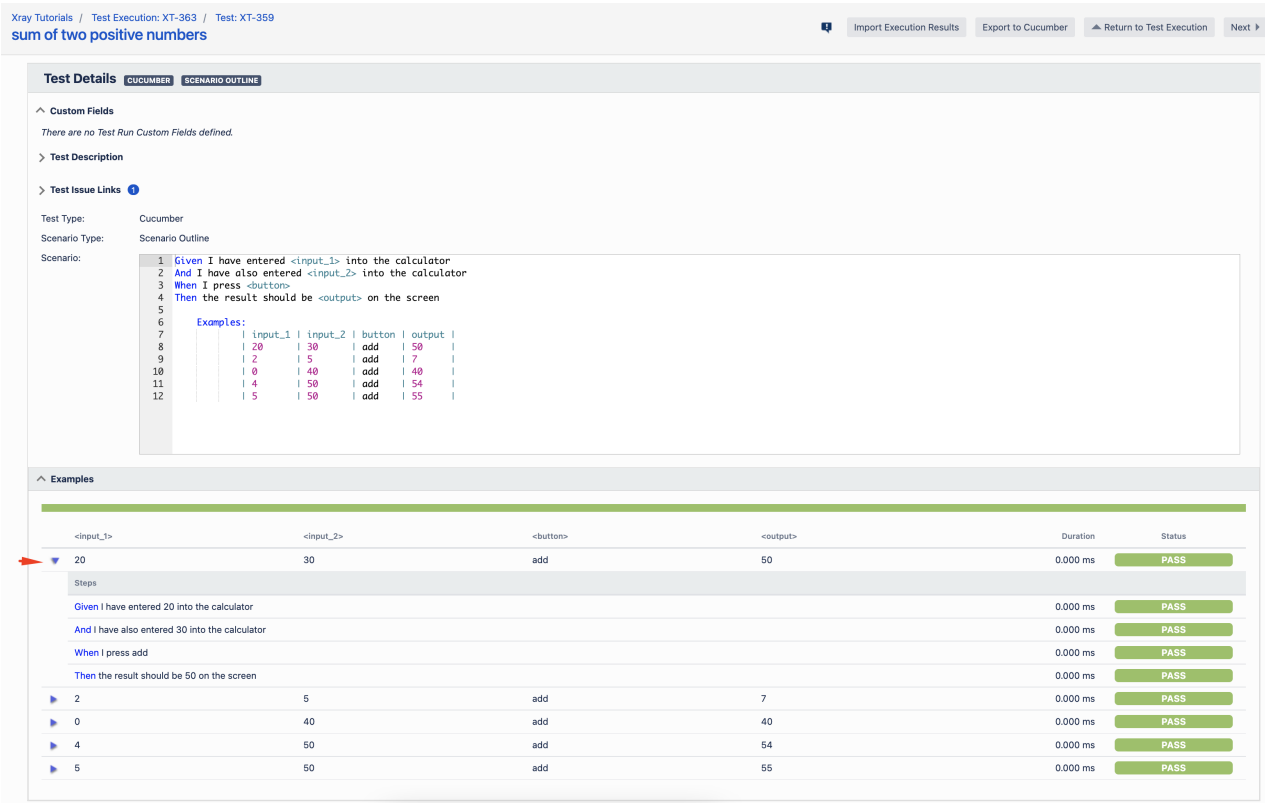
In sum, if you want to customize the Fix Version, Test Plan and/or Test Environment of the Test Execution issue that will be created, you'll have to use the "multipart cucumber" endpoint.

A new Test Execution will be created (unless you originally exported the Scenarios/Scenario Outlines from a Test Execution).



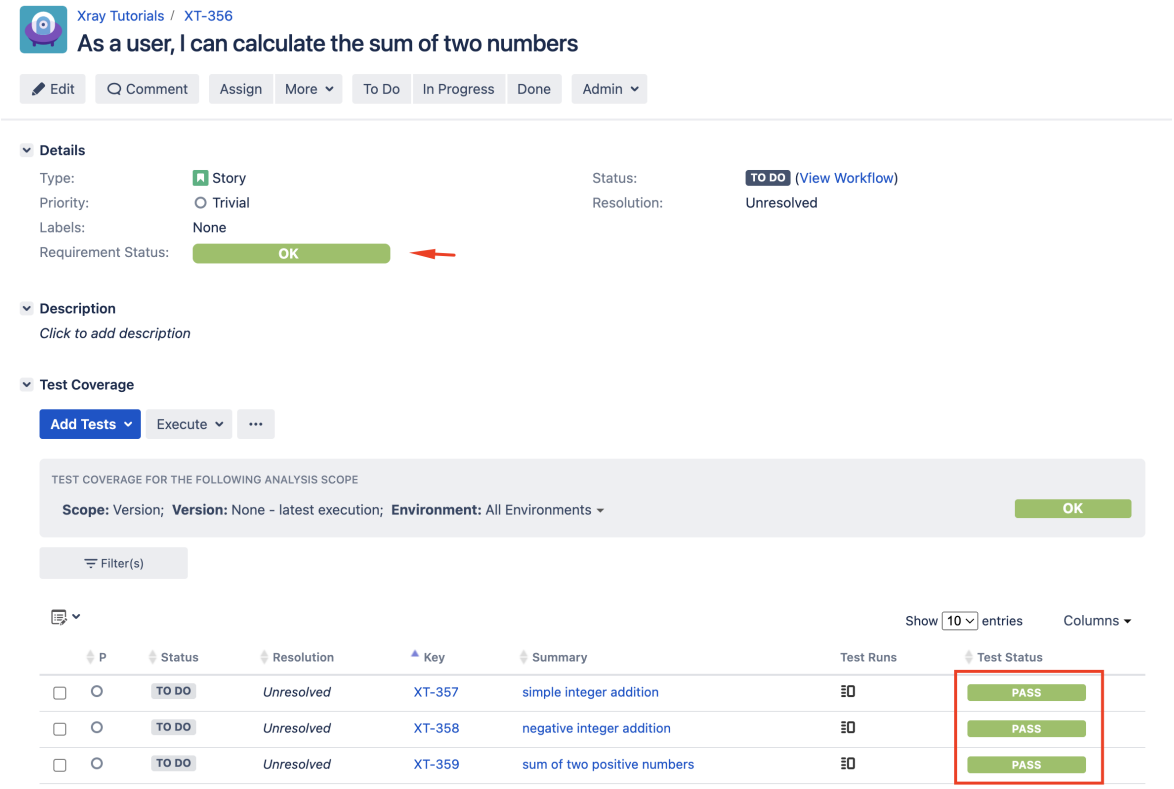
The execution screen details of the Test Run will provide overall status information and Gherkin statement-level results.

Results, including for each example on Scenario Outline, can be expanded to see all Gherkin statements.



Results are reflected on the covered items (e.g. Story issues) and can be seen in ther issue screen.

Coverage now shows that the addition related user story (e.g. XT-356) is OK based on the latest testing results.



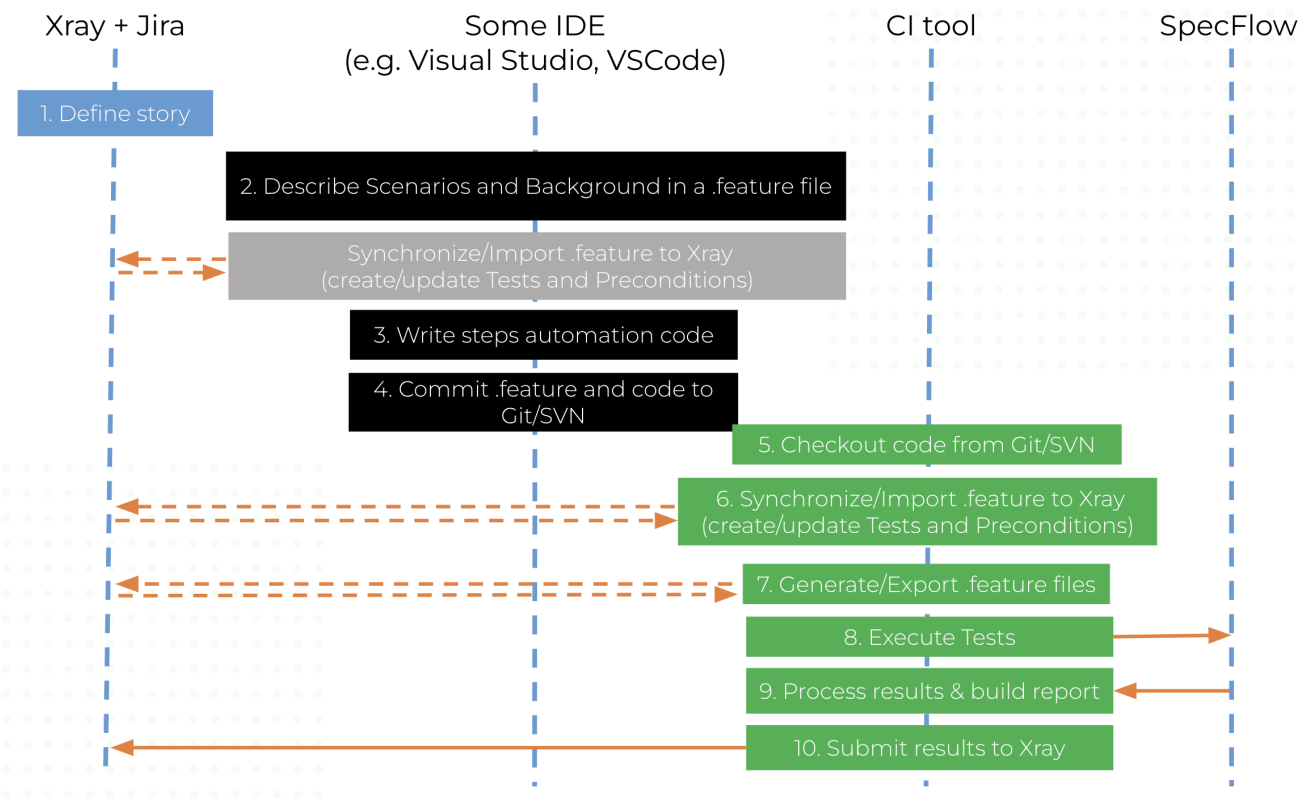
Using Git or other VCS as master

You can edit your .feature files using your IDE outside of Jira (eventually storing them in your VCS using Git, for example) alongside with remaining test code.

In any case, you'll need to synchronize your .feature files to Jira so that you can have visibility of them and report results against them.

The overall flow would be something like this:

1. look at the existing "requirement"/Story issue keys to guide your testing; keep their issue keys
2. specify SpecFlow/Gherkin .feature files in your IDE supporting SpecFlow/Gherkin and store it in Git, for example. Meanwhile, you may decide to import/synchronize them Xray to provision or update corresponding Test and/or Precondition entities
3. implement the code related to Gherkin statements/steps and store it in Git, for example.
4. commit code and .feature file(s) to Git
5. checkout the code from Git
6. import/synchronize the .feature files to Xray to provision or update corresponding Test and/or Precondition entities
7. export/generate .feature files from Jira, so that they contain references to Tests and requirements in Jira
8. run the tests in the CI
9. obtain the report in Cucumber JSON format
10. import the results back to Jira



Note that steps (5-10) performed by the CI tool are all automated, obviously.

To import .features to Jira we can either use the REST API or a CI tool. To export tagged .features from Jira, we can do it directly from Jira, by the REST API or using a CI tool.

> **Learn more**
>
> Please check the tutorial Testing using Cucumber in Java which showcases this flow for Cucumber+Java and adapt it accordingly to the SpecFlow use case.

FAQ and Recommendations

Please see this page.

References

- Code used in this tutorial, along with some auxiliary scripts
- SpecFlow documentation
- Testing in BDD with Gherkin based frameworks (e.g. Cucumber)
- Automated Tests (Import/Export)
- Exporting Cucumber Tests - REST
- https://github.com/SpecFlowOSS/SpecFlow-Examples