Testing using SpecFlow and Cucumber Scenarios in C#

Overview

In this tutorial, we will create some tests in Cucumber/Gherkin, using SpecFlow and C# and we'll import the results to Xray to have visibility of the test results.

Please note

There are some possible workflows related with Cucumber.

In this tutorial, we assume that the the tests (specification) are initially created in Jira as a Cucumber Tests and exported afterwards using the UI or the REST API; that's what we call the "standard" workflow.

If you prefer to manage the .feature and respective Scenarios outside of Jira, like in your own local dev environment/IDE or in Git/SVN, then you'll need to synchronize the specification to Jira as depicted in our VCS based workflow.

More info in Testing in BDD with Gherkin based frameworks (e.g. Cucumber).

Requirements

- Install SpecFlow and the SpecFlow+ Runner 1.7.2 or newer along with the msbuild helper package; If you're using Visual Studio, just go to NuGet's Console (**Tools | NuGet Package Manager | Package Manager Console**)
  - Install-Package SpecRun.SpecFlow
  - Install-Package SpecRun.SpecRun
  - Install-Package SpecFlow.Tools.MsBuild.Generation

packages.config

```
<?xml version="1.0" encoding="utf-8"?>
<packages>
  <package id="Newtonsoft.Json" version="9.0.1" targetFramework="net452" />
  <package id="SpecFlow" version="2.3.2" targetFramework="net452" />
  <package id="SpecFlow.Tools.MsBuild.Generation" version="2.3.2" targetFramework="net452" />
  <package id="SpecRun.Runner" version="1.7.2" targetFramework="net452" />
  <package id="SpecRun.SpecFlow" version="1.7.2" targetFramework="net452" />
  <package id="SpecRun.SpecFlow.2-3-0" version="1.7.2" targetFramework="net452" />
  <package id="System.ValueTuple" version="4.3.0" targetFramework="net452" />
</packages>
```

- Use the `CucumberJson.cshtml` report template provided in this page

Description

In this tutorial, we detail more extensively the standard Cucumber workflow (more info in Testing in BDD with Gherkin based frameworks (e.g. Cucumber)), where Xray/Jira is used as the master of information, i.e. the place where you edit/manage your Cucumber Scenarios.

An alternate approach would be using your IDE, or the feature files persisted in Git for example, as the master of information. In that case, the workflow is a bit different as we'll mention ahead.

Using Xray and Jira to manage the Scenario specification

In this use case, Cucumber Tests are written in Jira using Xray of type "Scenario" or "Scenario Outline", in Jira.

Calculator / CALC-2248

add two numbers

Edit | Comment | Assign | More | Start Progress | Resolve Issue | Close Issue | Admin

Details

Type:	Test	Status:	OPEN (View Workflow)
Priority:	Trivial	Resolution:	Unresolved
Affects Version/s:	None	Fix Version/s:	v3.0
Component/s:	None		
Labels:	None		

Description

Click to add description

Test Details

Type: **Cucumber**

Scenario Type: **Scenario**

Scenario:

```
1  Given I have entered 50 into the calculator
2  And I have also entered 70 into the calculator
3  When I press add
4  Then the result should be 120 on the screen
```

Calculator / CALC-2249

Add two positive numbers

Edit | Comment | Assign | More | Start Progress | Resolve Issue | Close Issue | Admin

Details

Type:	Test	Status:	OPEN (View Workflow)
Priority:	Trivial	Resolution:	Unresolved
Affects Version/s:	None	Fix Version/s:	v3.0
Component/s:	None		
Labels:	None		

Description

Click to add description

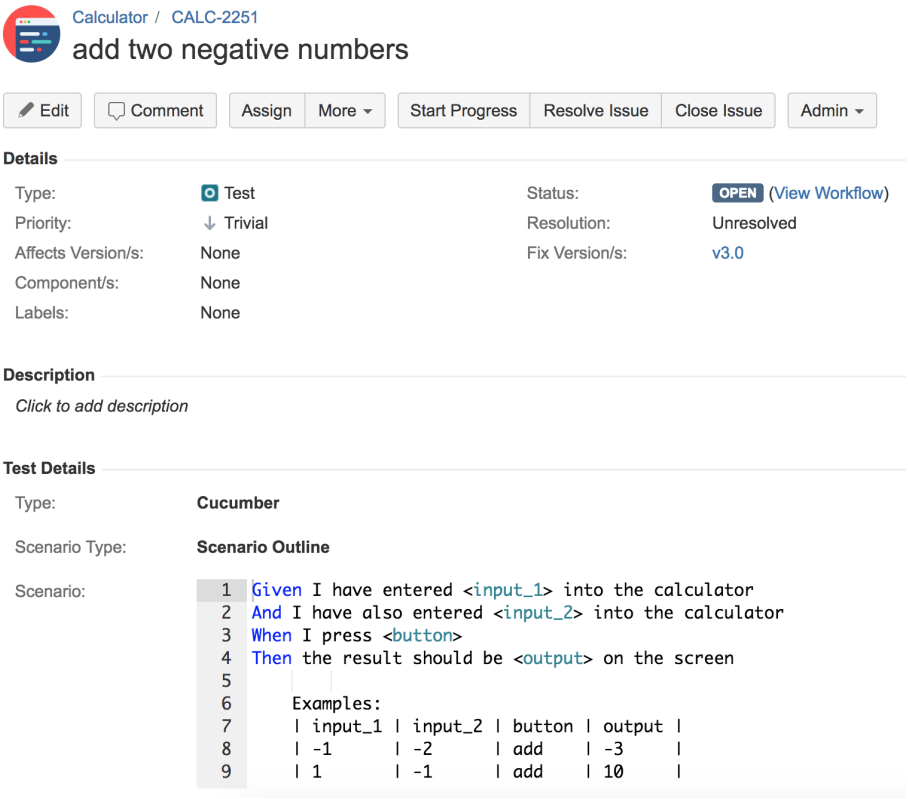
Test Details

Type: **Cucumber**

Scenario Type: **Scenario Outline**

Scenario:

```
1   Given I have entered <input_1> into the calculator
2   And I have also entered <input_2> into the calculator
3   When I press <button>
4   Then the result should be <output> on the screen
5
6     Examples:
7       | input_1 | input_2 | button | output |
8       | 20      | 30      | add    | 50     |
9       | 2       | 5       | add    | 7      |
10      | 0       | 40      | add    | 40     |
11      | 4       | 50      | add    | 54     |
```



You can export the specification of the tests to a Cucumber .feature file via the REST API or the **Export to Cucumber** UI action from within the Test Execution issue.

The created file will be similar to the following one.

1_CALC-889.feature

```
@CALC-2250
@REQ_CALC-2247
Feature: Sum Operation
        #In order to avoid silly mistake
        #
        #As a math idiot
        #
        #I want to be told the sum of two numbers


        @TEST_CALC-2249
        Scenario Outline: Add 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_CALC-2248
        Scenario: add two numbers
                        Given I have entered 50 into the calculator
                        And I have also entered 70 into the calculator
                        When I press add
                        Then the result should be 120 on the screen


        @TEST_CALC-2251
        Scenario Outline: add two negative 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 |
                        | -1      | -2      | add    | -3     |
                        | 1       | -1      | add    | 0      |
```

The actual step implementation code lives outside of Jira. Thus, you have to make the implementation for each step/sentence.

CalculatorSteps.cs

```
using System;
using TechTalk.SpecFlow;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using UnitTestProject1;

namespace UnitTestProject1
{
    [Binding]
    public class CalculatorSteps
    {
        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);
        }
    }
}
```

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 test profile to use it.

CucumberJson.cshtml

```
@inherits TechTalk.SpecRun.Framework.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))
```

Default.srprofile

```
<?xml version="1.0" encoding="utf-8"?>
<TestProfile xmlns="http://www.specflow.org/schemas/plus/TestProfile/1.5">
  <Settings projectName="UnitTestProject1" projectId="{5359f4fc-ee65-45b2-bb4e-5c0255b88806}" />
  <Execution stopAfterFailures="3" testThreadCount="1" testSchedulingMode="Sequential" />
  <!-- For collecting by a SpecRun server update and enable the following element. For using the
      collected statistics, set testSchedulingMode="Adaptive" attribute on the <Execution> element.
    <Server serverUrl="http://specrunserver:6365" publishResults="true" />
  -->
  <TestAssemblyPaths>
    <TestAssemblyPath>UnitTestProject1.dll</TestAssemblyPath>
  </TestAssemblyPaths>
  <DeploymentTransformation>
    <Steps>
      <!-- sample config transform to change the connection string-->
      <!--<ConfigFileTransformation configFile="App.config">
        <Transformation>
          <![CDATA[<?xml version="1.0" encoding="utf-8"?>
                                                    <configuration xmlns:xdt="http://schemas.microsoft.com
/XML-Document-Transform">
                <connectionStrings>
                  <add name="MyDatabase" connectionString="Data Source=.;Initial Catalog=MyDatabaseForTesting;
Integrated Security=True"
                       xdt:Locator="Match(name)" xdt:Transform="SetAttributes(connectionString)" />
                </connectionStrings>
                                                    </configuration>
                                                ]]>
        </Transformation>
      </ConfigFileTransformation>-->
    </Steps>
  </DeploymentTransformation>


  <Report>
    <Template name="CucumberJson.cshtml" outputName="data.json"/>
  </Report>

</TestProfile>
```

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

Since there is code-behind file generation, it is required to have the NuGet "SpecFlow.Tools.MsBuild.Generation" package.

```
msbuild /t:Clean;Rebuild
cd bin\debug
..\..\..\packages\SpecRun.Runner.1.7.2\tools\SpecRun.exe run Default.srprofile /outputFolder:..\..\..
\TestResults
cd ..\..
```

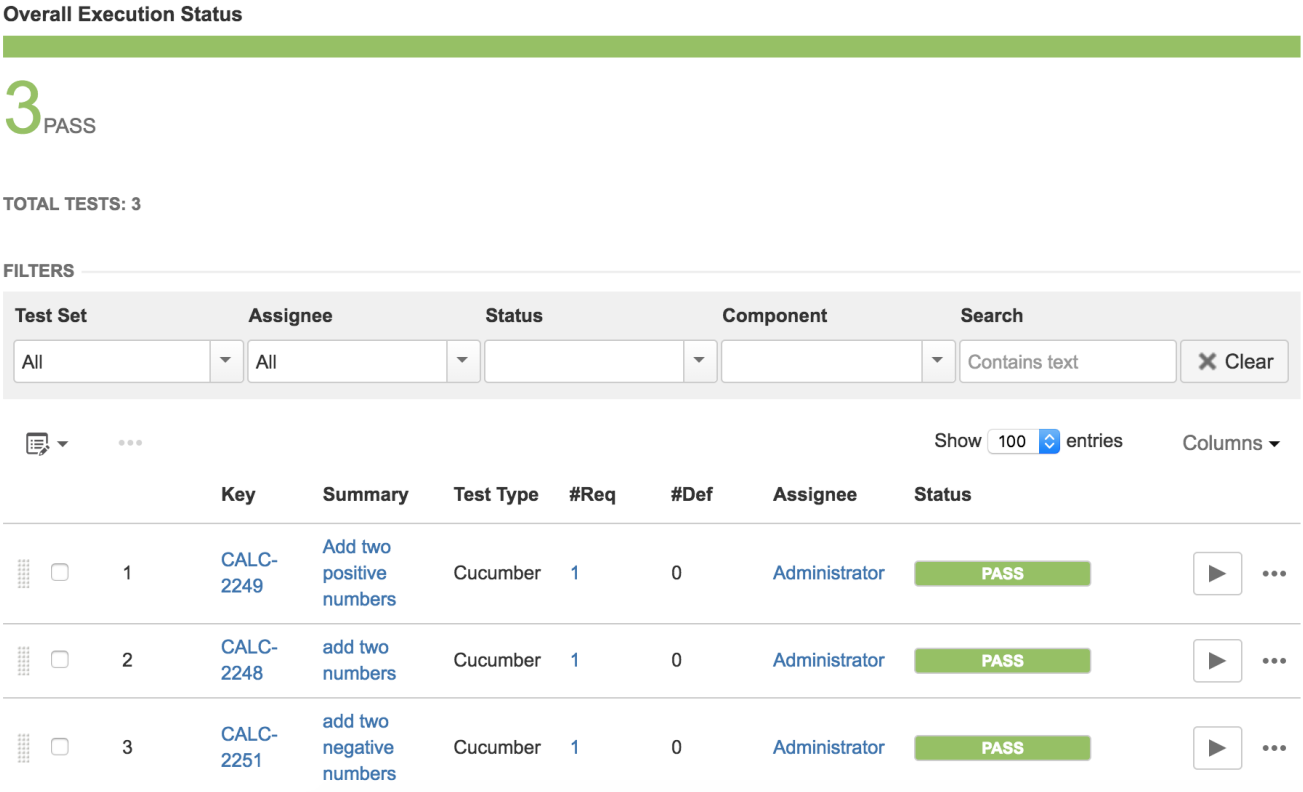
After running the tests and generating the Cucumber JSON report (e.g., data.json), it can be imported to Xray via the REST API or the **Import Execution Results** action within the Test Execution.

```
curl -H "Content-Type: application/json" -X POST -u user:pass --data @"data.json" http://jiraserver.example.com
/rest/raven/1.0/import/execution/cucumber
```

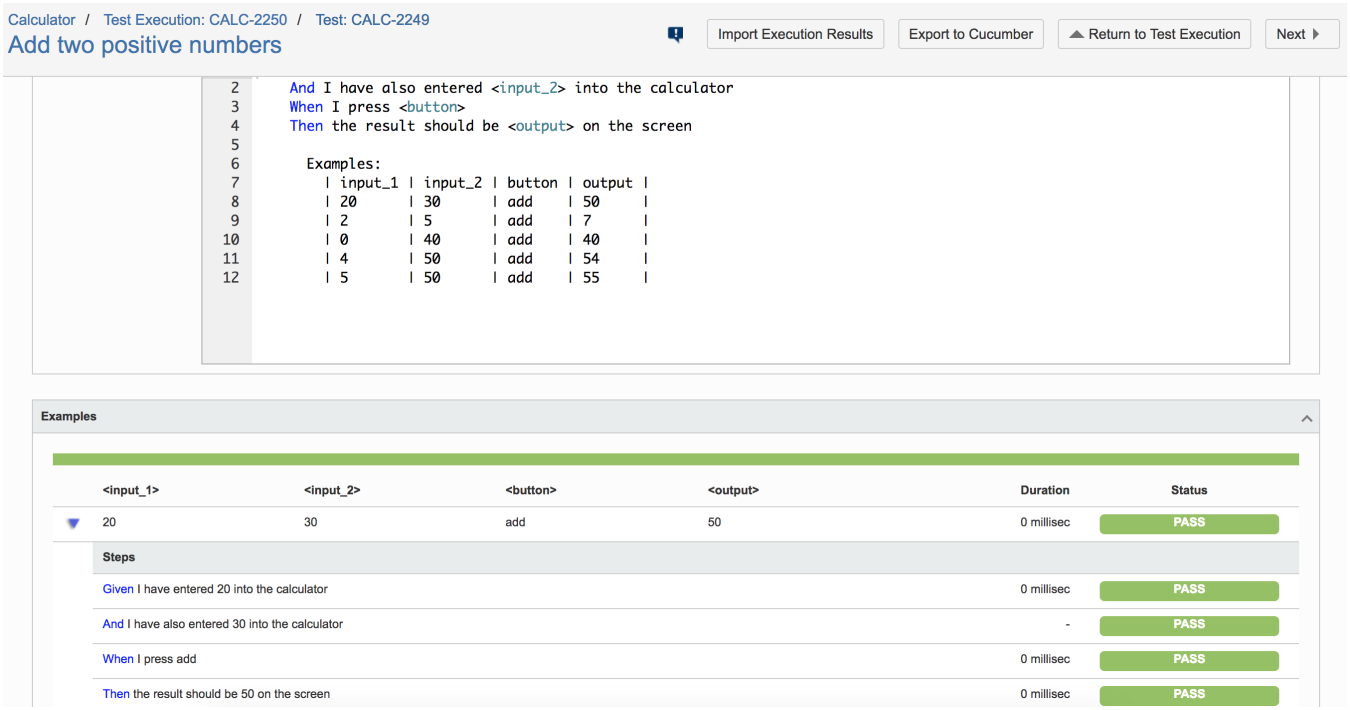
Since the original feature was extracted from a Test Execution, the results will be updated on it (this happens because the .feature file contains the Test Execution's issue key as a tag).

Please note

If the .feature was created by hand directly on your IDE, or managed elsewhere outside of Jira, and it didn't contain the Test Execution's key, then a brand new Test Execution would be created. This would also happen in case it was extracted using the REST API based on Test /requirement issue keys.



The execution screen details will not only provide information on the overall test run result, but also of each of the examples provided in the Scenario Outline and on the respective steps.



Managing the Scenario specification in your IDE, in Git or in other VCS

In this case you are using your IDE as means to write/edit the Scenarios and eventually persist them in the VCS (e.g. Git, SVN, other) so they can be run during Continuous Integration.

In this case, you'll need to regularly synchronize the specification to Jira as depicted in our VCS based workflow.

We also recommend that the .feature contains some auxiliary tags using the syntax `id:xxx` in each Scenario/Scenario Outline, to better guarantee that Scenarios are always mapped against the same Tests in Xray.

Before running the Scenarios, in order to produce a Cucumber JSON report that can be properly processed by Xray, we need to use the features extracted from JIRA instead of the ones we edit, because they will contain:

- tags corresponding to Test issue keys
- tag corresponding to the related Test Execution key, in case we want to use an existing Test Execution as the criteria to select the Tests to be run
- tags corresponding to the related requirement(s)

> **Learn more**
>
> Please see Testing in BDD with Gherkin based frameworks (e.g. Cucumber) for an overview on how to use Cucumber based Tests with Xray, and the VCS based workflow for the later example.

References

- https://specflow.org/getting-started/
- https://specflow.org/plus/documentation/SpecFlowPlus-Runner-Command-Line/
- Testing in BDD with Gherkin based frameworks (e.g. Cucumber)
- Exporting Cucumber Tests - REST